Conceptual and comparative analysis of application metrics in microservices

Grant #2019/26702-8, São Paulo Research Foundation (FAPESP)

Lucas Eduardo Gulka Pulcinelli, Diego Frazatto Pedroso and Sarita Mazzini Bruschi
ICMC - Institute of Computer Sciences and Mathematics
USP - Universidade de São Paulo
São Carlos, Brazil
{lucasegp, diegopedroso}@usp.br, sarita@icmc.usp.br

***Abstract*—Cloud computing represents an extensively implemented paradigm for provisioning distributed services, offering a significant degree of scalability for global applications. Nonetheless, when confronted with the necessity to scale, the system encounters monitoring challenges, as it must contend with an increased volume of requests while simultaneously accommodating fluctuating demands across various geographic regions. Aside from that, detecting errors in such a model becomes increasingly difficult, because of the many abstraction layers and interconnected microservices a cloud solution has. In that context, metrics can be used to identify errors and monitor the system's state. The substantial diversity in the types of services and the metrics themselves introduces a formidable complexity to the analysis of an entire cluster. Therefore, it is important to identify the essential metrics in microservices that can be used to recognize issues or bottlenecks. In pursuit of this objective, a cloud-based solution was implemented within an Amazon Web Services Kubernetes cluster to emulate the functionality of an online retail store and an automated testing framework was made to inject errors in different parts of this application while its metrics were obtained. In that way, it was possible to identify the effects that errors have on the metrics of components in the system, rendering the monitoring of the cluster a more direct process and reducing the amount of data to be analyzed in order to identify the presence of errors in a cloud environment.**

***Index Terms*—Cloud computing, Chaos engineering, Microservice, Observability**

I. Introduction

Cloud computing is a recent infrastructure provisioning model, which bases itself on the availability of virtual machines, storage, networking, and software platforms in an elastic and dynamic way. Such architecture has the purpose of being highly scalable, besides having high-cost efficiency, easy provisioning ability to many data centers around the world, and many other characteristics [1]. Because of that, it is possible to quickly create cloud-native services with thousands of interconnected replicas in heterogeneous computing nodes.

Moreover, it is seen that cloud applications have a high volatility, given that the amount of replicas needed to keep the services stable is variable [2]. Considering this, it is perceptible that remaining in control of the system's state is a challenge because of that unpredictability, stating the need for tools capable of providing observability of a cluster as a whole with high granularity.

In this context, observability is the characteristic of being able to determine the internal state of a system by analyzing only output data from it [3]. In the case of cloud applications, metrics are numerical data collected with a timely frequency which can be used to generate observability in clusters [4]. Each microservice instance can generate its own metrics based on computational resources, such as CPU or memory usage, network packages sent or received, and many others.

However, there exists a high variety in the number of metrics that can be collected in a cluster. This is seen in their categorization by two parameters: granularity and limiting resources. Granularity divides metrics into system-based, including data of the cluster as a whole; and application-based, considering only an instance of a running microservice [5]. In regards to limiting resources, CPU and I/O bound metrics can be seen [6], depending on the hardware resource related to it.

Furthermore, it is also of interest to consider that each service inside a cluster has a special purpose, and the metrics generated by an instance depend on their behavior. In that case, the same distinction of limiting resources can be applied to applications themselves, CPU or I/O bound [6].

Therefore, it is clear that, in field deployments, the amount of metrics in union with the amount of microservice replicas makes the identification of errors and their causes a difficult task [7]. Hence, it is of interest to analyze which of the many metrics obtained from a service are in fact relevant to identify different kinds of errors inside a cloud-native application.

II. Objectives

This project is part of a bigger one, at Ph.D. level [8], which proposes a system to detect the root cause of incidents using probabilistic inferences via Bayesian networks. Such a system consumes logs and microservice metrics and is able to automatically find correlated access points of log anomalies and metrics. This results in the reliable detection

979-8-3503-8160-3/23/$31.00 ©2023 IEEE
DOI 10.1109/SBAC-PADW60351.2023.00028

of critical software errors and potential interruption trigger [9].

In this context, the project aims to identify, of the many application and system metrics in a cloud environment, which are the most relevant to be monitored in order to find errors in service delivery. With that, the main project is able to reduce the amount of data to be analyzed and use the conclusions reached with this research in order to implement its root cause detection.

III. Related Work

Numerous systems have been developed with the primary objective of error detection within microservices architectures. An approach that uses metrics is [10], however it does not differentiate between CPU and I/O metrics, and does not seem to consider some application level metrics as separated by containers, such as CPU and RAM usage. Additionally, it employs metric analysis exclusively in response to high latency incidents between applications, which may not encompass the entirety of potential anomalies that could be promptly detected.

A system that selects only useful metrics and differentiate between kinds of microservices for root cause analysis is [11], using traces and historical metric data between each pair of services to determine which metrics are useful for analysis. However, it is essential to note that this approach does introduce a degree of system overhead, particularly if continuous system analysis were to be implemented. Consequently, the ultimate solution necessitates manual initiation in response to system faults.

IV. Methodology

A. Research and tool setup

1) Definition of a cloud test system: Initially, it was needed to organize a cloud system architecture to develop the following activities. The first step was obtaining the cloud provider platform, for that, Amazon Web Services (AWS) was chosen, being considered by many as the biggest cloud platform used in the market, showing itself as the ideal environment for the execution of tests that simulate a real service. Besides that, for the provisioning of dynamic compute, Kubernetes[1] was used in the form of the Amazon Elastic Kubernetes Service (EKS) system. The AWS cloud creation was possible thanks to the USP-AWS partnership[2].

Kubernetes is a cloud orchestration platform widely used in the market due to its high fault tolerance, easy workload replicability, and simple interconnection between its components. Particularly, the following concepts are of utmost importance in the development of all cluster applications created:

- Container: a unit of software bundled with all its dependencies for the bootstrap.
- Pod: a computing unit in the Kubernetes system, composed of one or more containers with many configurations, such as CPU and memory limits or persistent storage associated.
- Node: a machine (physical or virtual) provided by EKS (via the EC2, Elastic Compute Cloud, service) capable of executing pods. As the amount of nodes increases, the possibility of system outage becomes smaller, especially if the nodes are in different availability zones.
- Deployment and ReplicaSet: constructs for implementing replicability and redundancy in a pod workload. The first is based on a configurable number of replicas of a pod distributed in the cluster as a whole, such that if a pod fails or gets deleted another instance is created; while the second determines the amount of replicas of a pod that should be present in every node.
- Service: a DNS entry accessible within the cluster that connects to a load balancer of all pods in a deployment or replicaset, such that every connection to this service may go to different pods.

B. Application for test execution

Having the main cloud principles used defined, it was necessary to search for a cloud-native application architecture to resemble a real system, although smaller for the need to make controlled tests. In that regard, an application developed by Weaveworks called Sock Shop was found, which simulates an online shop with a login, cart, orders, and shipping systems. This environment example is "intended to aid the demonstration and testing of microservice and cloud-native technologies"[3], being ideal for the use case of this research.

Sock shop is composed of a total of 14 services, with 14 related deployments, each with two replica pods. The created services are:

- front-end, providing web pages directly accessible by users;
- session-db, a memory key-value Redis database, storing the session for registered and non-registered users.
- users-db, containing a non-relational MongoDB database with registered user data (including cards and addresses);
- users, controlling the access to users-db via a REST API;
- carts-db, containing a non-relational MongoDB database with cart data for each user;
- carts, controlling the access to carts-db via a REST API;
- payment, simulating a payment system that accepts or denies purchases (it is important to note that this system is not real, it just accepts purchases with a simple logic based on the value purchased);
- orders-db, containing a non-relational MongoDB database with all user product orders;

[1]https://kubernetes.io

[2]https://www.usp-aws.org/homes

[3]https://microservices-demo.github.io

- orders, which creates new orders, connect to the payment system to approve purchases, and sends them to the shipping service;
- catalog-db, containing a relational MySQL database with all items that can be purchased;
- catalog, controlling the access to catalog-db via a REST API;
- RabbitMQ, being a message queue with all shipping orders already paid to be processed;
- shipping, which includes orders to the queue;
- queue-master, that receives approved orders from the queue and processes them, ending the purchasing flow.

It is important to see the tool variability used in the sock-shop application, because some deployments use interpreted languages with complex frameworks (such as NodeJS in JavaScript for front-end), while others are created in compiled languages such as go or java. Moreover, there is variability in the software architecture used, such as relational and non-relational databases, messaging queues, and key-value cache systems.

Figure 1 is a diagram of services and their connections, differentiated by limiting resources.

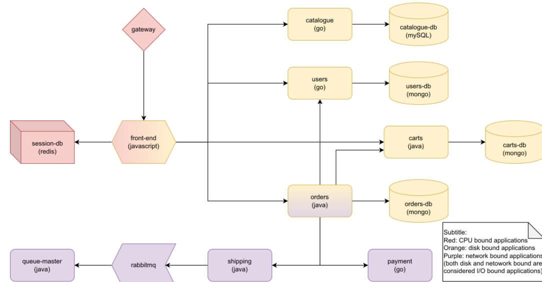

Fig. 1: Sock-shop application diagram

As it can be seen, the application is sufficiently close to a real distributed system, ideal for the purposes of the research, as it was also used by other academic works in the area [10].

1) Definition of the metric collection system: Besides the creation of the cloud application itself there was a need to plan the tools to collect and store metrics. It was identified that, because metrics are of high interest for the observability of any cloud system, solutions were especially developed for this scenario. In this context, Prometheus [4] is a time series database created for the storage and access control of metrics of all types described, using a standard format called openmetrics.

openmetrics[5] is a specialized protocol for the description and communication of metrics between databases and generators (which will be described later), basing itself in the usage of unique identifiers for each metric, having labels to increase granularity, and numeric values for each data point with certain labels. In the standard, each application or generator makes its metrics available in a certain path (such as "/metrics") and obtains them in the moment of a GET request for that specific time. Importantly, it is Prometheus' responsibility to make requests with a certain interval for all analyzed services and keep control of timestamps for each value.

In this context, there are metric generators that analyze a whole scope of metrics in the moment of a request and obtain important values. Between these generators, two were identified which, besides being widely used, have a high variety of metric types: the node-exporter and Cadvisor services. Node-exporter samples data for each node and some data regarding the whole cluster, being a part of prometheus; while Cadvisor obtains metrics for each pod and is not part of the prometheus stack directly.

2) Definition of a test automation system: Having systems for metric collection and the application to be analyzed, the next step is the definition of a testing system and the tests to be executed.

To simulate a web solution with an approximately real workload, it was necessary to create user traffic in an automated way. For that, locust[6] was used, a system for scalable and easily programmable load testing for cloud applications. Locust generates by default data such as response time distributions, mean response size, amount of responses with a given HTTP return code, all of that for each request path accessed by the load test. Besides that, the number of fake users to be created is fully configurable, as are the probabilistic distributions of which pages will be accessed.

Furthermore, it was necessary to determine which kinds of errors can be caused in a cloud system, and how to generate them in a systematic and replicable way. For that, the chaos-mesh[7] service was seen as ideal, as it is capable of introducing many types of exceptions inside in application or in the connection between microservices to analyze its fault tolerance.

3) Replicabilidade of Experiments: Finally, it was seen the need to execute the tests without human interference and in a reproducible way, while also adding cluster components orderly. For that, the continuous integration and continuous deployment (CI/CD) named CircleCI[8] was used, being able to, via configuration files and scripts, create Kubernetes resources, execute tests and introduce errors, all described in a code repository.

C. Developed Activities

1) Creation of the whole system: Initially, the AWS cloud system and the Kubernetes cluster were instantiated, besides the initialization of the code repository with all the project's files. In order to facilitate a heightened level of reproducibility, the infrastructure as code system, the infrastructure as code system Terraform[9], together with its own CI/CD called Terraform cloud, were used to declare all of the AWS service instances. Aside from the usual users,

[4]https://prometheus.io
[5]https://openmetrics.io
[6]https://locust.io
[7]https://chaos-mesh.org
[8]https://circleci.com
[9]https://terraform.io

a special one was created for use with CircleCI, allowing it to change the cluster's state systematically.

After that, an instance of CircleCI was generated outside of the project's AWS environment and associated with the base repository, with an initial deployment workflow for the creation and updating of Kubernetes applications: every time a tag of the code versioning system git is created with a specific syntax (that identifies which application is to be acted upon), a CircleCI machine would test the code for that application, create a container for it, and deploy the application in the cluster via a Kubernetes resource description file. If errors happen at any step, the previous state of the system is restored.

Using the CI/CD system described, besides the use of other tools such as helm[10], it was possible to instantiate prometheus, node-exporter and the Cadvisor metrics generators in a Kubernetes namespace, such that they would not influence the sock-shop services that will be included, but would still collect a total of 356 unique metrics from it. Other monitoring services were included too to allow real-time analysis of the metrics, such as Grafana[11], which only reads prometheus data and shows them in dashboards, without interfering with the metric collection, sock-shop services, or tests.

Moreover, using CircleCI, it was possible to deploy the sock-shop application. Besides the Kubernetes resources, an EKS ingress gateway was created to expose the front-end service in a public URL, such that it can be accessed by the test system directly (without the use of the Kubernetes client port-forward capability, because this is not a recommended ingress method by the developers, so it was considered that its use could change test results).

Moving on to the test framework, another CircleCI workflow was created for load testing: code using locust is developed with a certain page distribution and added to the repository; then, after the user creates a git tag that indicates the user amount and a total test time, the CircleCI machine executes the test and sends all results, for data centralization, to the cluster prometheus database as metrics. Such metrics are not used in the analysis, differently from the node-exporter and Cadvisor ones, because the results would only be seen by the final user, not by the cluster in a real environment. They were primarily collected for the confirmation of test execution properties and to detect procedural errors in the test itself. Figure 2 shows a Grafana dashboard with data related to locust

[10]https://helm.sh
[11]https://grafana.com

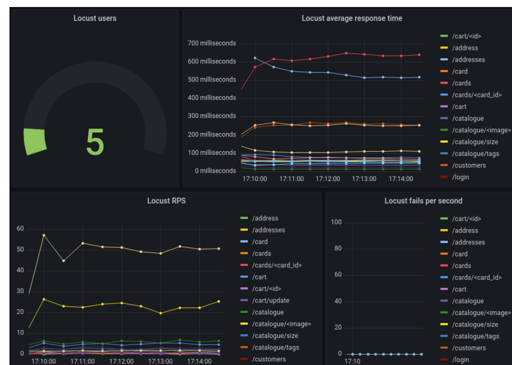

Fig. 2: Locust metric visualization dashboard via Grafana during a load test with 5 users

Finally, the creation of a cluster error generator using chaos-mesh was developed. For its use, the previous workflow git tag was modified to include an optional parameter with the error that should be generated after a certain time of normal load test execution, defined as a custom Kubernetes resource read by the chaos-mesh daemon.

D. *Execution and extraction of important results*

With the complete infrastructure, a load test program was developed with Locust capable of simulating the normal use of the application, applied in all scenarios described ahead. Regarding the kinds of errors generated, initially, the analysis considers the failure of pod provisioning for a certain microservice. To ensure the correctness of results, it is needed to repeat the tests with different services. The final list of errors generated is presented below:

- Provisioning error for front-end pods;
- Provisioning error for users pods;
- Provisioning error for payment pods;
- Provisioning error for shipping pods;
- Provisioning error for carts-db pods;

Furthermore, another test was included without the execution of errors, as a base case. Lastly, the execution of each test is as follows: the load test would be initiated, a metric stabilization time of fifteen minutes would pass, then the chaos-mesh error would be introduced with a duration of five minutes, and, after the error time passes, another ten minutes are needed to stabilize the metrics again and finish the load test.

After the execution of all tests, its Cadvisor and node exporter metrics were collected via a single program to generate the results.

V. Results

For the front-end application, it was seen that I/O bound metrics related to the network from both Cadvisor (in Figure 4) and node-exporter (in Figure 3) are strongly correlated with the failure. Such a fact can be explained by the centrality of the application, being connected to all other microservices directly or indirectly and acting as the entrypoint for users. This centrality causes the whole system

to stop responding to user requests, and therefore the data transfer suffers heavly.

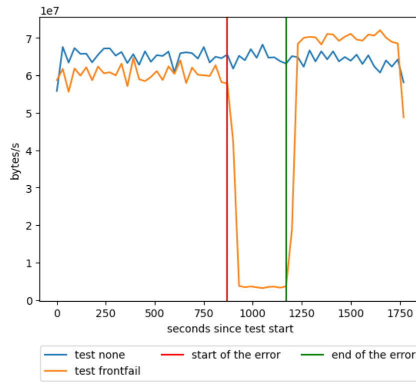

Fig. 3: Network data downstream per second for all nodes during tests

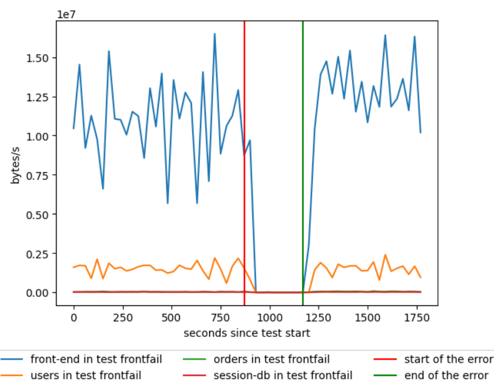

Fig. 4: Network data downstream per second for sock-shop microservices during front-end fail

Furthermore, because of the same centrality, CPU bound metrics also had a direct impact due to error, as seen in Figure 5 and 6, because without access to the gateway, no requests could be processed. However, this kind of metric in node-exporter was not conclusive, this can be explained by two factors: the influence of the kubernetes runtime itself, which, specially in error scenarios, may consume more CPU time; and the fact that the majority of applications in the sock-shop system are I/O bound, with the exception of the caching layer.

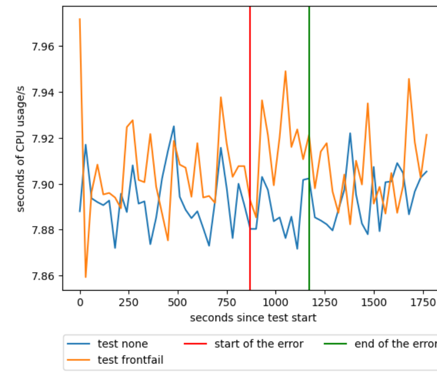

Fig. 5: total CPU usage for all nodes during tests

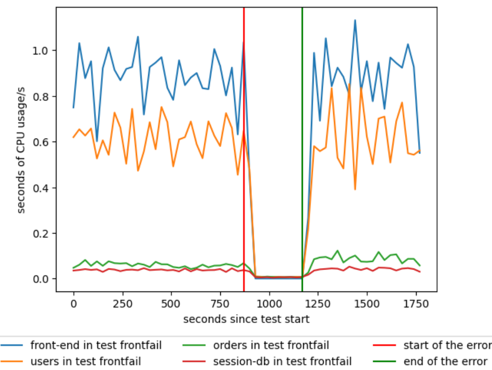

Fig. 6: total CPU usage in sock-shop microservices during front-end fail

An interesting result is that, during the various creation retries during the failure, the front-end application had high disk reading metrics. This happens because, during the initialization phase, HTML files and code were brought from secondary memory to primary many times while the application failed to start repeatedly. It is important to note that this only happened until the Kubernetes runtime noted this repeated scheduling failure and stopped trying to create front-end pods (a process called crash loop backoff). The disk reading metrics are seen in Figure 7.

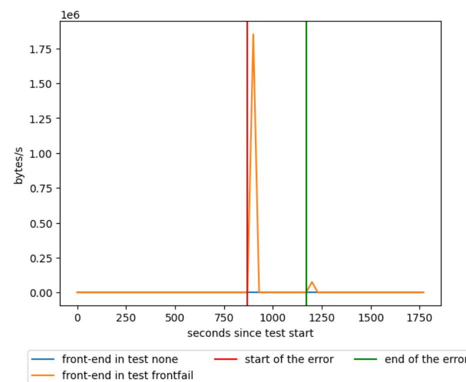

Fig. 7: Bytes read from disk per second for front-end during various tests

For the user's application, a compatible result was obtained, with the same heavy downfall for the network and a high disk reading rate, as seen in Figures 8 and 9. This is due to a panic that happened in the front-end as a result of the user failure, visible in Figure 10, which caused a crash in the system as a whole as well.

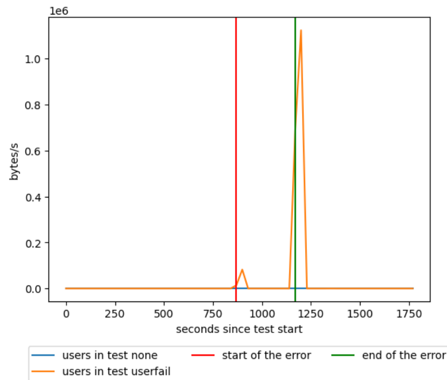

Fig. 8: Bytes read from disk per second for front-end and users during various tests

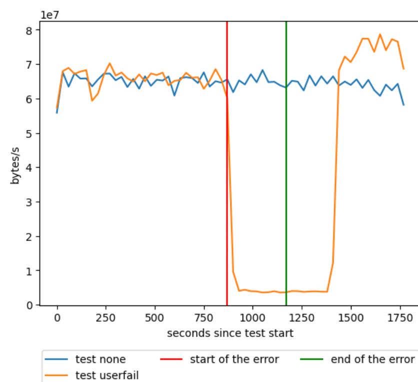

Fig. 9: Network data downstream per second for all nodes during tests

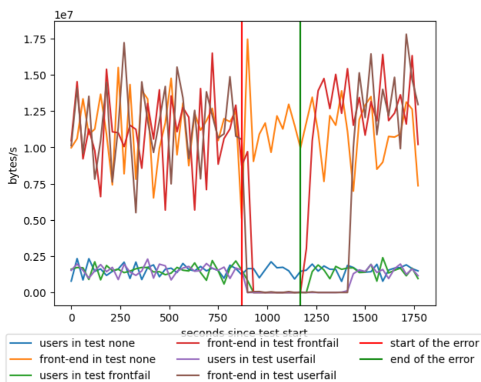

Fig. 10: Network data downstream per second for users and front-end during various tests

It is notable that the normal application workload only returned after five minutes after the error ended. This does not happen because of the user's microservice, but because of the front-end crash loop backoff, because, by default, Kubernetes only allows recreation of applications in that state after ten minutes.

On the other hand, it was seen that, because the user's pods are made in a compiled language, there was a deterministic fall in the memory usage in their pods after the error, as seen in Figure 11, because the restarts of the service before the crash loop backoff made the application free all it's allocated memory during execution. Such behavior was not seen in any other scenario with users, being an effective way to detect the errors in these kinds of applications.

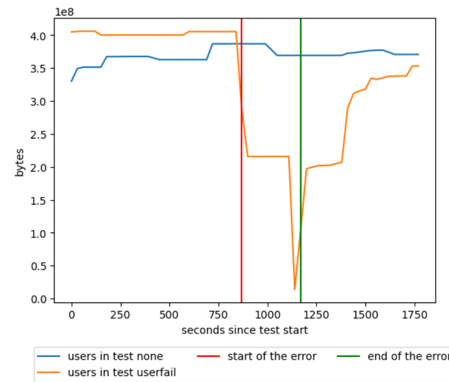

Fig. 11: Memory usage by users during tests

It is important that such behavior of memory freeing happened with front-end, as seen in Figure 12, but, because of its interpreted language with a complex runtime, it is harder to see this, especially during the users fail that triggered front-end crashes.

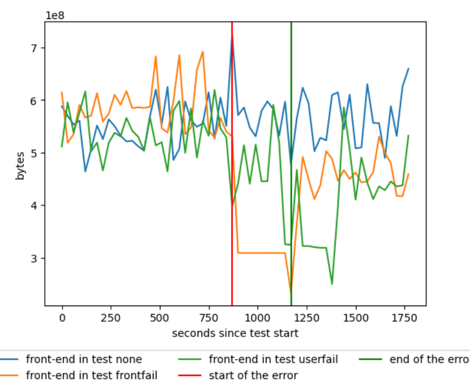

Fig. 12: Memory usage by front-end during tests

For the payment application, the error could not be detected by CPU or I/O bound metrics from node-exporter, this is because the failure did not trigger a whole system panic different from the first two tests, so the processing of other services (and consequently their resource usage) was not affected. Only two metrics were useful in the

detection of the error: the memory freeing as in the user's case, and a small increase in the CPU usage of the orders microservice, which is the only application that directly connects to payments. The disk usage was not seen in the pre-crash loop backoff stage, differently from the other cases. The payments' service memory, orders' CPU usage and payments' disk readings can be seen in Figures 13, 14 and 15, respectively.

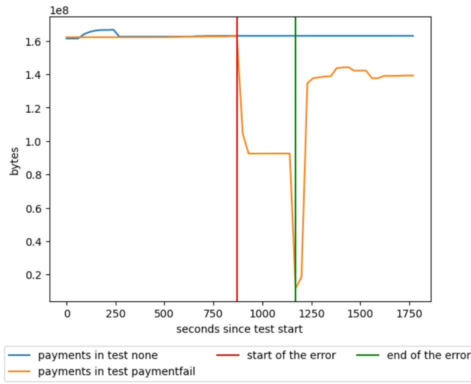

Fig. 13: Memory usage by payments during tests

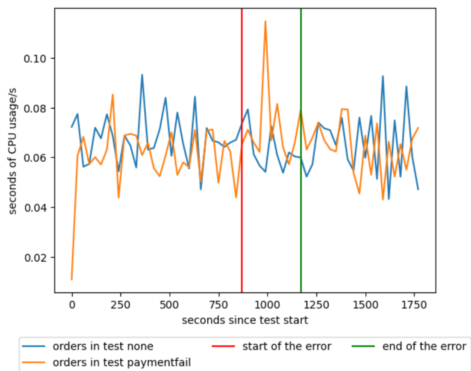

Fig. 14: CPU usage of orders during tests

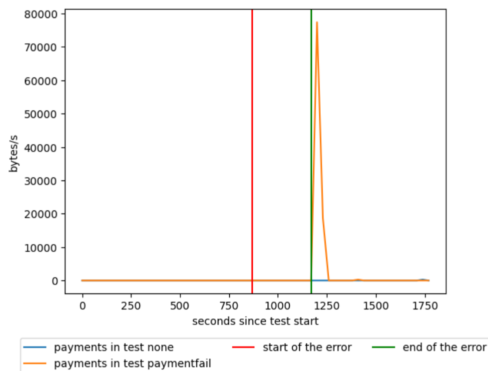

Fig. 15: Bytes read from disk per second for payments during tests

For shipping, a visible change in memory and disk readings was seen in its own metrics, as shown in Figures 16 and 17, but not in the applications that connect to it (also orders in that case, seen in Figure 18).

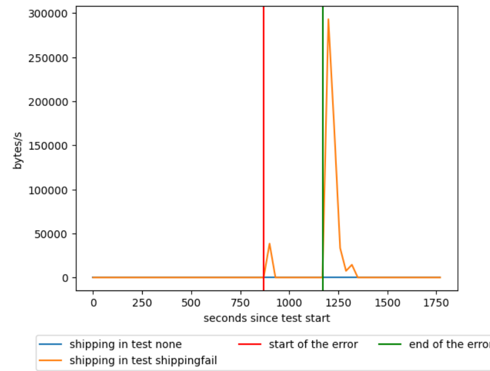

Fig. 16: Bytes read from disk per second for shipping during tests

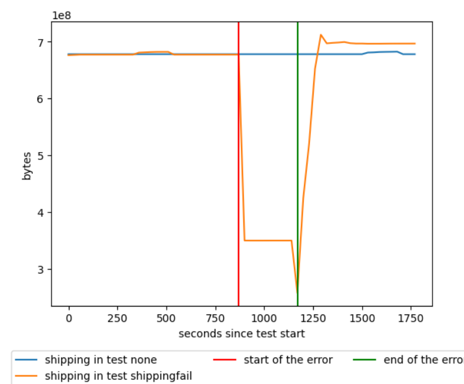

Fig. 17: Memory usage by shipping during tests

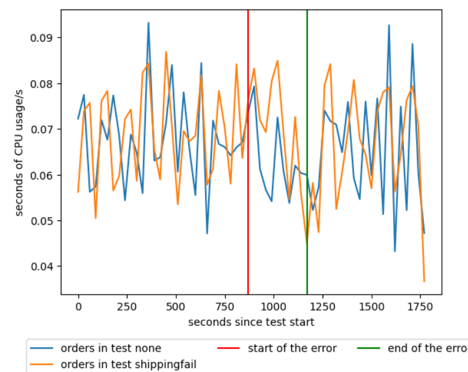

Fig. 18: CPU usage of orders during tests

Once more, as the error did not generate a complete system failure, node-related metrics were not affected, as seen in Figure 19.

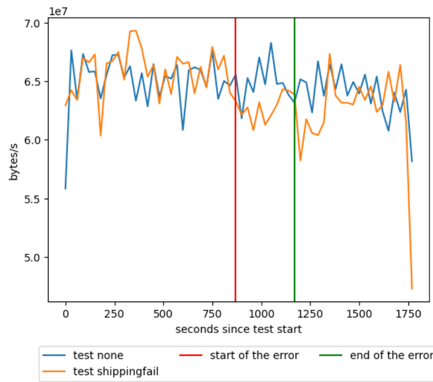

Fig. 19: Network data downstream per second for all nodes during tests

Lastly, it was of interest to analyze the behavior of a database, especially its disk reading metrics as a factor in identifying errors. The results are near the ones from other compiled microservices, visible in Figures 20 and 21.

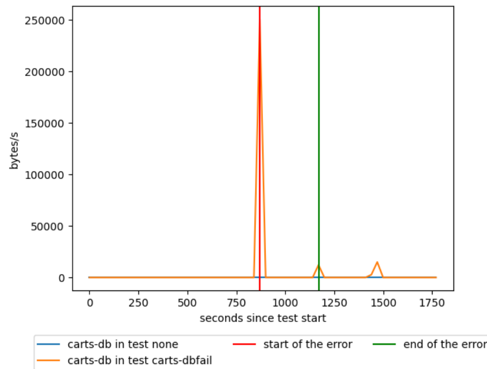

Fig. 20: Bytes read from disk per second for carts-db during tests

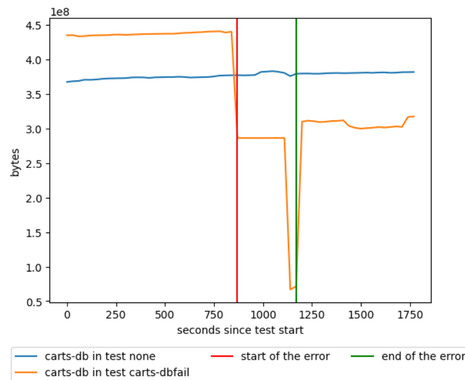

Fig. 21: Memory usage by carts-db during tests

VI. Conclusion

It is clear that, when dealing with pod provisioning failures, a high amount of applications present an increase in I/O bound metrics related to disk considering its own containers due to the pre-crash loop backoff phase. Besides that, microservices developed in compiled languages had a deterministic fall in memory usage, while interpreted languages had this behavior, but it was not as clear. However, it was not possible to correlate node metrics to this kind of error, except in the case of a complete application failure. Finally, pod CPU data was not consistent during the analysis of this error, once more with the exception of a complete crash of the system.

Therefore, to identify pod failure, it is ideal to analyze an abnormal fall in memory usage, together with an increase in disk readings. If the application is compiled, memory is a more reliable metric, while if the application has many files associated (such as HTML pages or databases), disk is essential for the detection.

A. Future works

Considering the direct correlation of disk I/O and memory metrics in compiled languages from Cadvisor with the presence of errors, it is of interest to expand this analysis to other kinds of errors using the test system developed. Especially, tests related to I/O failure in relational or non-relational databases and connection failures between pods or nodes are directly possible to do in chaos-mesh.

Furthermore, a future objective is to develop a microservice capable of using the analysis provided in this project to identify such failures in a cloud environment.

References

[1] S. Li et al., “Understanding and addressing quality attributes of microservices architecture: A Systematic literature review”. 2021.

[2] S. Merkouche and C. Bouanaka, “A Hybrid approach for containerized Microservices auto-scaling”. 2022 IEEE/ACS 19th International Conference on Computer Systems and Applications (AICCSA), Abu Dhabi, United Arab Emirates, 2022, pp. 1-6, doi: 10.1109/AICCSA56895.2022.10017677.

[3] C. Majors and L. Fong-Jones, “Observability Engineering”. O’Reilly Media, 2022.

[4] S. Newman, “Building microservices”. O’Reilly Media, 2021.

[5] B. Li et al., “Enjoy your observability: an industrial survey of microservice tracing and analysis”. Empir Software Eng 27, 25 (2022). doi: 10.1007/s10664-021-10063-9.

[6] M. Ma, W. Lin, D. Pan and P. Wang, “Self-Adaptive Root Cause Diagnosis for Large-Scale Microservice Architecture”. In IEEE Transactions on Services Computing, vol. 15, no. 3, pp. 1399-1410, 1 May-June 2022, doi: 10.1109/TSC.2020.2993251.

[7] A. Brandón et al., “Graph-based root cause analysis for service-oriented and microservice architectures”. 2020.

[8] D. F. Pedroso, “Análise automática de causa raiz de incidentes em ambientes multi-cloud utilizando redes bayesianas”. 2022.

[9] Y. Izrailevsky and C. Bell, “Cloud reliability”. IEEE Cloud Computing, v. 5, n. 3, p. 39–44, 2018.

[10] L. Wu, J. Tordsson, E. Elmroth and O. Kao, “MicroRCA: Root Cause Localization of Performance Issues in Microservices”. NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium, Budapest, Hungary, 2020, pp. 1-9, doi: 10.1109/NOMS47738.2020.9110353.

[11] Z. Li et al., “Practical Root Cause Localization for Microservice Systems via Trace Analysis”. 2021 IEEE/ACM 29th International Symposium on Quality of Service (IWQOS), Tokyo, Japan, 2021, pp. 1-10, doi: 10.1109/IWQOS52092.2021.9521340.