# Implementing Shadowing for a Guestbook Application: Ensuring Correctness in Google Cloud

Mohammad-Al Fattah, Sai Vineeth Karakavalasa, Md. Ariful Islam, Irvan Assiakoley

***Abstract*— This report presents the development and implementation of a shadowing mechanism for a guestbook application in the Google Cloud environment. Shadowing is utilized as a testing strategy to ensure the correctness and reliability of new system revisions before deployment to production. The report discusses the objectives, scope, requirements, and methodology involved in effectively implementing shadowing in the Google Cloud platform.**

***Index Terms*— Shadowing, Google Cloud, Kubernetes, Read replica, Docker, Guestbook Application, Redis, Google Container Registry (GCR), Istio, Grafana.**

## I. INTRODUCTION

The guestbook application is a critical system we have chosen, where new revisions must be thoroughly tested before deployment to production. To achieve this, a shadowing mechanism is proposed and implemented in the Google Cloud environment. This mechanism allows for the testing of new versions in a controlled environment that mirrors production, ensuring that potential issues are identified and addressed before impacting end-users.

## II. RESEARCH QUESTIONS

### *A. Motivation*

The motivation behind this research question stems from the critical importance of ensuring the correctness and reliability of the guestbook application, especially in environments where system failures or errors can have significant consequences. By implementing shadowing in the Google Cloud environment, we aim to proactively identify and mitigate potential issues before they impact end-users, thereby enhancing the overall stability and performance of the system.

### *B. Questions*

1. How can shadowing be implemented effectively in the Google Cloud?
2. How do we treat the production database in the shadowed environment?
3. How can we detect bugs from differences in the responses of a new version and/or its behavior?

### *C. General Approach*

*Setting Up the Shadow Environment with Google Cloud:*

To establish an effective shadow environment for our guestbook application, we begin by provisioning a Google Kubernetes Engine (GKE) cluster. Once the GKE cluster is in place, we deploy Redis as our database, which offers high performance and scalability. Virtual machines (VMs) or Kubernetes clusters, ensure that the same instance types, disk sizes, and network configurations are used in both production and shadow environments. The application itself is containerized using Docker to facilitate seamless deployment, with all necessary Docker containers stored in Google Container Registry (GCR). Dockerizing the application involves creating a Docker image that encapsulates all the necessary components and dependencies. This Dockerized application is then deployed to the GKE cluster using Kubernetes manifests, which define the deployment configurations and service specifications.

*Treatment of the Production Database in the Shadowed Environment:*

To ensure the shadow environment accurately reflects production, use Redis for data replication, the master-slave replication method is employed to synchronize data between a production environment and a shadow environment without disrupting the production workload. This replication allows us to mirror the production environment's data without affecting the primary database's performance. Traffic mirroring in Istio allows you to duplicate incoming requests to a secondary service or version of your application for testing or monitoring purposes without impacting the primary traffic flow. To safeguard sensitive data, data anonymization techniques can be applied to the replication.

*Detection of Bugs and Differences:*

To ensure application reliability, we utilize Google Cloud Monitoring, Logging, and Trace to detect discrepancies and bugs between production and shadow environments. Google Cloud Monitoring tracks response times, error rates, and critical metrics, alerting us to potential issues. Cloud Logging captures detailed logs for thorough analysis, while Cloud Trace provides performance insights through request traces. These tools are integrated into our testing strategy, ensuring new application versions are rigorously tested in the shadow environment before deployment to production. This approach minimizes disruptions and enhances overall application stability by identifying and addressing issues early in the development process.

### *D. Use cases*

*Feature Toggle Testing:*

Feature toggles can be tested in a shadow environment to ensure that new features can be turned on or off without affecting the overall application performance or user experience.

*Disaster Recovery drills:*

A shadow environment can be used to simulate disaster recovery scenarios. By replicating the production environment,

teams can practice their response to various failure scenarios and ensure their disaster recovery plans are effective.

*Security testing and Vulnerability scanning:*

Security testing in a shadow environment allows for thorough vulnerability scanning and penetration testing without compromising the security of the production environment. This helps in identifying and fixing security issues before they can be exploited.

*Training and Development:*

Shadow environments provide a safe space for training new team members and testing new development tools or techniques without risking the stability of the production environment

## III. Design

### *A. Key Design Decisions*

One of the key design decisions in our project is the adoption of a microservices architecture to enable scalability, flexibility, and maintainability of the system. We chose Kubernetes as the orchestration platform for deploying and managing containerized microservices on the Google Cloud Platform due to its robust features for automating deployment, scaling, and monitoring. Redis was selected as the database service for storing application data due to its compatibility with Kubernetes and its managed, scalable, and high-performance features. Docker containers are stored in Google Container Registry (GCR) for seamless integration and deployment.

### *B. System Context*

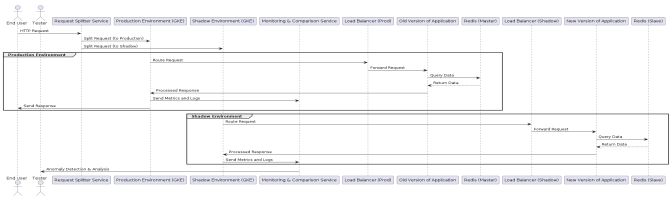

Fig.1 Architectural UML diagram for Shadowing

The approach outlined involves implementing a shadowing mechanism to test new versions of a chat application within a Google Kubernetes Engine (GKE) environment. In this system, an "End User" sends HTTP requests to a "Request Splitter Service." This service is responsible for duplicating or splitting the incoming requests, directing one set to the production environment and another set to the shadow environment. The production environment processes the requests using the old version of the chat application and routes them through a load balancer to the appropriate application and database services. The production environment then returns the processed responses to the end user while simultaneously sending metrics and logs to a monitoring service.

In the shadow environment, the split requests are processed by the new version of the chat application, following a similar routing and database querying process through a separate load balancer. However, the shadow environment queries a read replica of the production database to avoid impacting the live data. The processed responses in the shadow environment are sent to the monitoring and comparison service but not back to the end user. This service, used by testers, analyses metrics and logs from both environments to detect anomalies and compare the performance and behaviour of the new application version against the production version. This setup ensures rigorous testing and validation of the new version without disrupting the live application.

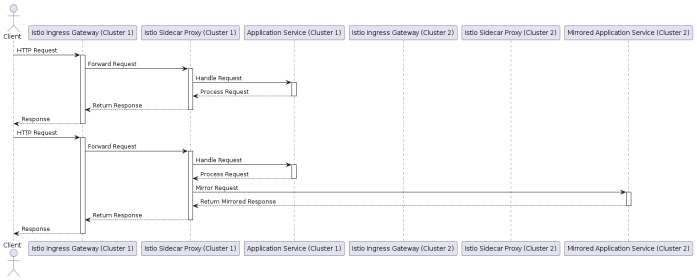


Fig.2 Sequence diagram for traffic splitting using Istio

### *C. Component View*

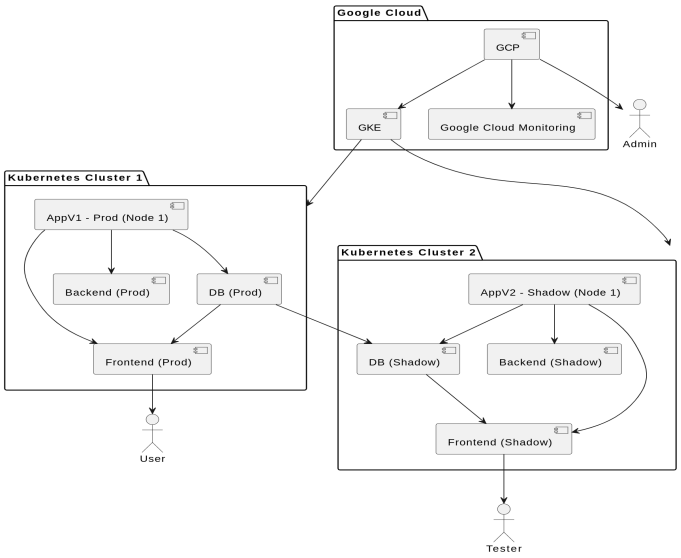


Fig.3 Deployment Diagram for Containerized Application

## IV. Findings

*How can shadowing be implemented effectively in the Google Cloud?*

During the implementation of shadowing in Google Cloud, I leveraged several key components to achieve effective results. Setting up a GKE cluster provided a robust foundation due to

its scalability, automated updates, and seamless integration with other Google Cloud services. Dockerizing the guestbook application ensured consistent deployment across various environments, simplifying versioning and updates. Storing Docker images in Google Container Registry (GCR) facilitated reliable and efficient image management, crucial for deploying applications in GKE. Additionally, implementing Redis with master-slave replication allowed for data synchronization between production and shadow environments, maintaining data consistency for testing purposes without impacting the production environment. These strategies collectively enhanced the shadowing process, resulting in a reliable, scalable, and maintainable environment for testing and deploying applications in Google Cloud.

*How do we treat the production database in the shadowed environment?*

In the shadowed environment, we treat the production database by using traffic mirroring with a service mesh like Istio. This approach allows us to duplicate incoming requests to a secondary service or version of the application for testing or monitoring purposes without disrupting the primary traffic flow. Additionally, we ensure data consistency and validate changes by setting keys (such as usernames and session IDs) in the production environment and verifying their retrieval in the shadow environment using GET operations. This process helps confirm that changes propagate correctly during testing and shadow deployment scenarios.

*How can we detect bugs from differences in the responses of a new version and/or its behaviour?*

To detect bugs arising from differences in responses or behavior of a new version, we utilize Google Cloud Monitoring. This tool enables us to monitor response times, error rates, and other critical metrics, helping identify discrepancies and potential issues. While we await dynamic data traffic for comprehensive implementation, we currently rely on the Google Cloud monitoring tool Grafana to facilitate this process.

## V. Conclusion

Our approach to traffic mirroring with Istio and data validation techniques provided a comprehensive method for treating the production database within the shadowed environment, ensuring minimal disruption to the primary traffic flow while enabling thorough testing and monitoring. Furthermore, the utilization of Google Cloud Monitoring and Grafana has been instrumental in detecting bugs and discrepancies, thereby enhancing the stability and performance of the application.

This shadowing mechanism not only improves the reliability of our deployments but also offers a versatile framework that can be adapted for various use cases, such as feature toggle testing, disaster recovery drills, security testing, and training. The insights gained from this implementation underline the importance of rigorous pre-production testing in minimizing disruptions and enhancing overall system stability. Future extensions of this work could explore more advanced traffic mirroring techniques, automated anomaly detection, and the integration of additional cloud-native monitoring tools to further refine the shadowing process.

## VI. References

1. Professor René Wörzberger. (2024). “lcss.asciidoc” , Large Cloud based Software systems, TH Köln. Available at https://gitlab.nt.fh-koeln.de/gitlab/common/exercises/-/blob/master/lcss.asciidoc
2. Google Cloud Platform Documentation. (2023). "Google Kubernetes Engine (GKE) Overview." Retrieved from Google Cloud
3. Docker Inc. (2023). "Docker Overview." Retrieved from Docker.
4. Google Cloud Platform Documentation. (2023). "Google Container Registry (GCR)." Retrieved from Google Cloud
5. Redis Labs. (2023). "Redis Documentation: Master-Slave Replication." Retrieved from Redis.
6. Google Cloud Platform Documentation. (2023). "Google Cloud Monitoring." Retrieved from Google Cloud
7. Grafana Labs. (2023). "Grafana: The Open-Source Observability Platform." Retrieved from Grafana.
8. Kumar, A., & Srivastava, P. (2020). "Microservices Architecture and its 10 Most Important Design Patterns." International Journal of Software Engineering & Applications (IJSEA), 11(2), 67-81.
9. Namiot, D., & Sneps-Sneppe, M. (2014). "On Micro-services Architecture." International Journal of Open Information Technologies, 2(9), 24-27.
10. Santos, J. M., & Parreira, B. (2018). "Scalable, Flexible, and High-Performance Database Management with Redis." Journal of Database Management, 29(1), 1-20.