Incident Mitigation

Andrew Owens
University of Minnesota - Twin Cities
College of Science and Engineering
Minneapolis, MN, USA
owens518@umn.edu

Jonathan Meshesha
University of Minnesota - Twin Cities
College of Science and Engineering
Minneapolis, MN, USA
meshe005@umn.edu

***Abstract*—As organizations increasingly rely on cloud services to support critical operations, ensuring the reliability and resilience of these cloud services becomes vital to maintaining uptime and the user's quality experience. Production incidents can range from throttling to total service failure; nevertheless, their impact on customers can be expensive due to the time and resources required for effective mitigation. Through the lens of a key-value storage system deployed in the Google Cloud Platform (GCP), we analyze the factors that contribute to production incents, and subsequently, we evaluate the efficacy of various automation techniques in enhancing system resilience. This project introduces a set of automation techniques, namely automated deployment, automated rollback, and automated monitoring, implemented to mitigate incidents quickly and effectively. The core focus of this project lies in assessing the impact of these automation techniques on the key-value service's behavior during and after the incidents. We then analyze whether these automation strategies contribute to increased system resilience, thereby minimizing downtime and enhancing the overall service reliability.**

***Index Terms*—MGI, SRE, GCP**

I. Introduction

A. Production Incidents

A production incident is defined as any "event that causes a disruption or reduction of quality to a service." [3] These events can range from code errors that cause production service failure or throttling to even simply a large spike in user traffic. Incidents can be disastrous as they erode user trust and can cost millions of dollars in lost revenue. In 2021, Facebook and its associated services experienced an outage for 6 hours which is estimated to lose the company over 65 million dollars in revenue. [4] Incidents are unfortunately inevitable in complex, large-scale cloud services. Thus, responding to and mitigating these incidents as fast as possible is key to minimizing the loss in revenue and user trust.

B. Goals

Given a containerized key-value storage system deployed as a cloud service:

1) Identify points of failure.
2) Create a set of metrics to monitor service performance and detect potential incidents.
3) Create a CI/CD workflow environment that enables automated deployment.
4) Add testing stage to deployment pipeline to mitigate code bugs.
5) Using the defined metrics, create a set of policies to trigger the mitigation techniques such as automated rollback or autoscaling.
6) Assess the service's performance and uptime during and after incident mitigation procedures.

C. Motivation

The motivation behind this project is rooted in the recognition that production incidents are an inherent and inevitable aspect of managing large-scale cloud services. As organizations increasingly transition towards cloud-based applications, the complexity of these systems introduces new challenges related to service reliability. Likewise, the growing field of Site Reliability Engineering (SRE) emphasizes the importance of proactively addressing reliability challenges in cloud environments. This project aligns with this growing emphasis on SRE principles by aiming to contribute empirical insights into incident mitigation for cloud-based key-value storage services. By exploring the interaction of automation, monitoring, and error management within the context of a key-value storage service, this project aims to elucidate the practical implications of mitigating and preventing production incidents. We seek to not only advance our understanding of service resilience but also contribute to the broader discourse on the evolving landscape of cloud services and the importance of mitigation techniques in sustaining these cloud applications.

II. Methodology

A. Key-Value Storage Cloud-based Application

The cloud application we will be using for testing is a minimal key-value storage service built with Flask-Python. The users can send requests over HTTP to create, retrieve, and delete key-value pairs. The Flask application also connects to a backend PostgreSQL database for persistence memory. Finally, we used Docker to containerize the application and use it for version control.

B. Production Environment Design

This project was implemented using the Google Cloud Platform. We utilize GCP's cloud instance templates to define the desired configurations for our virtual machine instances. This ensures that there is consistency between the instances

Identify applicable funding agency here. If none, delete this.

in our deployments. These virtual machine instances are all part of a Managed Instance Group (MIG), which enables us to implement automation in the deployment and management of identical instances. The MIG also enables us to implement autoscaling, allowing for dynamic adjustments to the number of instances based on the service demand. The current configuration specifies a minimum of 2 instances and a maximum of 3 instances, governed by the autoscaler policies. These instances all produce logs that we can then use to build log-based metrics as part of our monitoring. Using the metrics we specify, we

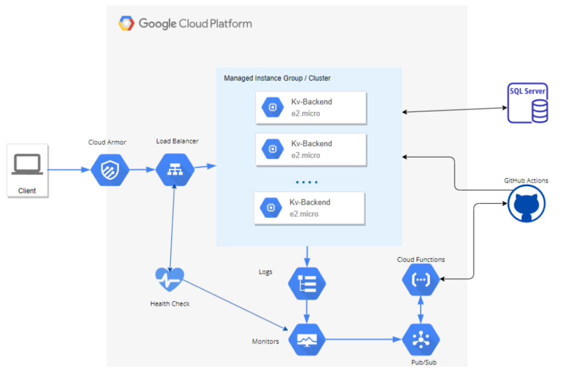

Fig. 1. Diagram of the infrastructure's design.

can set up a PubSub channel to distribute notifications and trigger certain mitigation techniques when an alerting policy is activated.

The load balancer is another critical component of our design as it is responsible for distributing incoming requests evenly across the deployed instances. We use consistent hashing as the load-balancing policy to ensure that requests operating on the same key are consistently routed to the same instance. This enhances the efficiency in handling related operations and creates a more predictable and stable system behavior.

To ensure memory persistence between sessions, the instances can read/write key-value pairs to a PostgreSQL backend database. GitHub is used for our code repository and CI/CD workflow.

C. Testing

Automating tasks prone to human error can reduce the likelihood of incidents occurring as developer error is a common root cause for incidents. Testing is one candidate that this project and many other production services implement automation for. Typically, automated tests run right before every code release is deployed and on failure it prevents the deployment process from starting up. Assuming high test coverage, this process ensures that code changes themselves are correctly implemented and that they don't break other systems. This allows automated tests to serve as the first line of defense against code bug based incidents.

In the context of our key-value store, both unit tests and integration tests were employed to create this first line of defense. Firstly, unit tests were developed to test the underlying

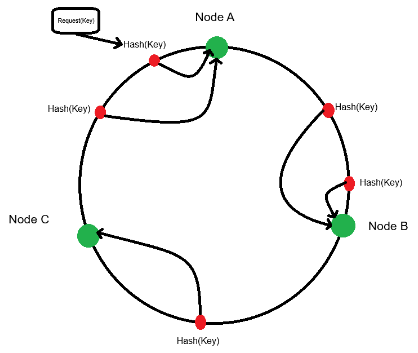

Fig. 2. Consistent Ring Hash: In Ring Hash, each point on the ring represents a possible hash value for a key. The node whose position is closest to the key's hashed position in a clockwise direction is responsible for all operations related to that key. When a new node is added or an existing node is removed, only the keys that are within the range of responsibility for that node need to be remapped. This minimizes the amount of data that needs to be moved.

data structure used to store key-value pairs for each operation including accessing, creating, deleting, and updating. Next, unit tests were developed to test query generation for SQL queries sent to the PostgreSQL database. Finally, integration tests were developed to test the interactions between these systems by running the request lifetime for all paths and using a mock connection to the database to avoid test flakiness.

D. Automated Deployment

The deployment of new source code to production is another candidate for automation that many use to prevent developer error and mitigate incidents. Manual deployment is prone to configuration errors as there are many steps that could be easily performed incorrectly. For example, a developer might mistakenly deploy the wrong container image version to production or might incorrectly input application secrets like a database password. Automated deployment facilitates quick and consistent system updates, minimizing the potential for configuration errors common in manual deployments.

In our project, GitHub Actions was utilized to implement automated deployment. This was the best option compared to other services like Circle Ci and Jenkins as it is directly integrated into GitHub, easy to use, and low cost. The deployment action executes every time a new commit is pushed and is split into two parts. First, once all tests pass, a docker image of the application is built and uploaded to docker hub which serves as the image repository. Each image is tagged with the release version using a simple versioning system that increases the version number by 1 for every new release. Second, a new instance template with this new image is created within gcp and is named after the image version tag. The MIG is notified to switch to this new instance template which kicks off the process of creating new instances from the template. Figures 3 and 4 show both parts of the deployment action respectively.

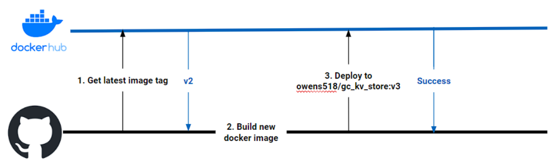
docker hub
1. Get latest image tag
v2
2. Build new docker image
3. Deploy to owens518/gc_kv_store:v3
Success

Fig. 3. First half of the deployment GitHub Action where the application is built and deployed to Docker hub with a tag named after the release version.

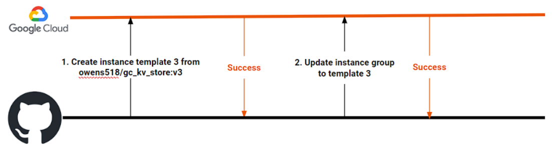
Google Cloud
1. Create instance template 3 from owens518/gc_kv_store:v3
Success
2. Update instance group to template 3
Success

Fig. 4. Second half of the deployment GitHub Action where the MIG is updated to a new instance template using the newly built Docker image.

The process of creating new instances from a new instance template was implemented with GCP's rolling update so that service interruptions are minimized during deployment. During a rolling update, older instances with the previous release are slowly drained while new instances with the new release are created. This means that the load balancer sends new requests to the new instances and once old instances complete all their currently running requests, they are removed. Rolling updates ensures that during deployment, running requests are still fulfilled and new requests still can be served.

E. Automated Rollback

Rolling back to a previous release version when an issue arises is a hugely important candidate for automation. When issues slip through both testing and deployment, rollback acts as a last safety net to catch them. However, by the time rollback is needed, the incident has already started and the production service is being disrupted. Thus, time is of the essence to mitigate the incident. Manual rollbacks can be slow and prone to developer error just like deployment. This is further worsened by the fact that a developer might not see an alert triggered by the incident right away. Automated rollback procedures allow this last safety net to mitigate the incident as fast as possible.

Just like with deployment, this project implements rollback with GitHub Actions. However, instead of being triggered by a commit being pushed, it should get triggered whenever an issue occurs in GCP. Specifically, whenever an "instance failed to deploy" alert occurs, a cloud function gets triggered that simply sends an HTTP request to GitHub that executes the rollback action. The rollback action itself is very similar to the deployment action. However, since the previous release version already exists on docker hub, the rollback action only needs to run the second part of the deployment action. Figure 5 shows this action in more detail.

F. Automated Monitoring

Automated continuous monitoring enables proactive detection of potential incidents, thus allowing for timely intervention and mitigation. However, monitoring is a complex endeavor that can quickly spiral out of control if you try to micromanage every small detail. Monitoring that has become too complex becomes fragile and difficult to change. For example, creating multiple alert policies for different latency thresholds at different percentiles adds unnecessary levels of complexity to your monitoring system as latency can easily fluctuate for various reasons. In [1], SREs at Google define the four golden signals of monitoring as latency, system traffic, error rate, and saturation. A common trap that should be avoided when building a monitoring system is to base it upon the mean values of some unit. Using latency as an example, if the average latency is about 50ms, then 1% of the requests may take several seconds to complete. If we create a monitor that aggregates the backend latencies by the 99th percentile, meaning the values by which 99% of the latencies fall below, this monitor becomes fragile as it is susceptible to falsely alerting when an unproblematic fluctuation in latency occurs. Such an example is seen in Figure 6, where the 99% backend latency spikes to 154ms while the 50% backend latency sits at only 56ms. It's also important to ensure that your monitors are measuring over a large enough period as periods of 1-2 minutes may not be large enough to sufficiently capture the behavior of the service.

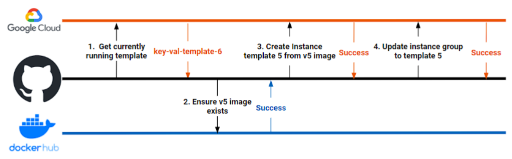
Google Cloud
1. Get currently running template
key-val-template-6
2. Ensure v5 image exists
Success
3. Create Instance template 5 from v5 image
Success
4. Update instance group to template 5
Success
docker hub

Fig. 5. Rollback GitHub Action where the MIG is updated to a new instance template containing the previous version of the Docker image.

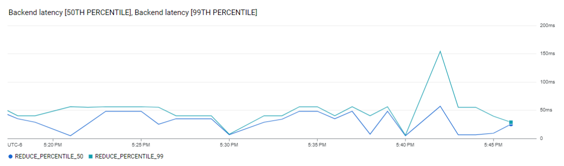

Fig. 6. Graph displaying differences in latencies when grouped by 99th and 50th percentile.

With these challenges in mind, we can revisit the four golden signals and define what exactly they'll be measuring.

1) Latency: To differentiate between a slow average and a very slow tail of requests, we created a monitor that aggregates request counts into buckets grouped by latency and protocol response codes. This enables us to not only keep track of the average time it takes for each request to complete but also distinguish the latencies between successful and failed requests.

2) Traffic: For a key-value storage application, traffic is measured by the request and response throughput. To measure this, we fetch the two tables of time series involving request

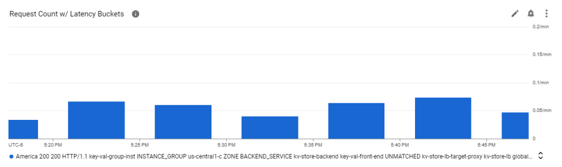

Fig. 7. Improved latency graph with protocol response codes specified. Notice how the graph covers the same time span as Figure 6 but lacks the large spike in latency. The protocol response codes also provide an additional level of detail to describe the overall service behavior.

and response read bytes from the load balancer, join those results, and then add them to get the total throughput.

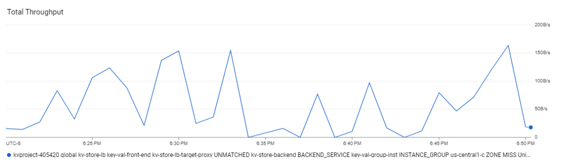

Fig. 8. Total throughput.

3) Error Rate: The error rate measures the rate of requests that fail. We can explicitly define this as the rate of HTTP 500s, however, protocol response codes are not always sufficient enough to capture the full failure condition. It is important to always have secondary measures that can

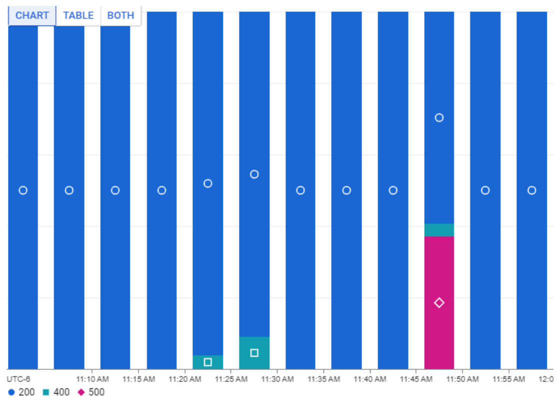

Fig. 9. Protocol Response Code Monitor

supplement your error rate monitors. As mentioned in the Production Environment Design section, each instance is constantly producing logs detailing information about the behavior of its key-value application. On top of this, we also have a health check that is constantly sending requests to the load balancer to monitor and log its health. We can use these logs to create log-based metrics that will give us the full picture regarding the overall performance and behavior of the system. To parse out the meaningful data from the lags, we can use queries with regular expressions (regex) to extract only the logs that we want to monitor. One such example can be seen in the following query:

```
jsonPayload.message =~".*(\[ERROR\]|\[CRITICAL\])|
    FATAL:.*"
```

This query extracts any log that contains the severity tag Error, Critical, or Fatal. One such example of an error log would be the "Worker failed to boot" error which only occurs when a new instance is being created. A new instance is created when either the auto scaler scales up the number of instances or a new instance template has been deployed to the MIG. If the rate of "Worker failed to boot" errors rapidly spikes, this can be indicative of a failed deployment, and thus mitigation techniques should be employed.

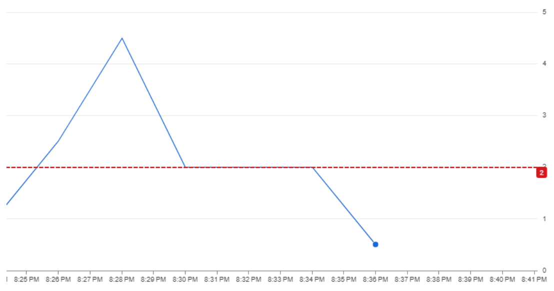

Fig. 10. Worker Failed to Boot log-based metric. We use a time window of 2 minutes and aggregate the error logs by summing them together. When the autoscaler creates a new instance, it can fail even when nothing is wrong with the key-value service itself, so we set the threshold to 2 to avoid false positives.

4) Saturation: Saturation is a measurement of how full your service is. This can be represented as resource utilization, such as CPU utilization. Increases in latency can also act as an indicator of saturation. Measuring the 99th percentile in latency over a short period can provide an early warning signal of incoming saturation. It's important to note that most systems will begin to degrade in performance well before they reach 100% utilization, so creating utilization targets is essential in maintaining system performance.

G. *Autoscaling*

The dynamic nature of cloud services demands that you take an adaptive approach to resource provisioning. Autoscaling is thus a crucial component of our design as it helps ensure the reliability and scalability of the key-value storage service to meet demands. Autoscaling ensures that our system always operates with the optimal number of instances, avoiding underutilization during spans of low demand and preventing degradation in performance during periods of high demand. By dynamically scaling resources in response to demand, autoscaling also reduces cost overhead by keeping your infrastructure in line with your actual service demand, eliminating the need for overprovisioning. Using the four golden signals as our metrics, we can configure our autoscaling policies to respond to varying workloads and adapt to unexpected traffic surges. Our autoscaling policy is configured to create a new instance under the conditions that an instance has surpassed 60% CPU utilization for more than 2 minutes or the 99th percentile latency exceeds 135ms for more than 2 minutes. The combination of CPU utilization and latency metrics provides a comprehensive view of the system's performance, enabling proactive scaling before the user experience is negatively impacted.

H. PubSub

PubSub serves as a crucial communication backbone for our system, enabling the different components to exchange information in a decoupled and asynchronous manner. This asynchronous manner allows the components to operate independently, enhancing the system's overall fault tolerance. PubSub also guarantees that messages are delivered to subscribers even when faced with intermittent failures or large fluctuations in system load. PubSub follows an event-driven architecture, meaning that it enables our system to react to events in real time, facilitating rapid and automatic responses to incidents, thus contributing to improved system agility.

Our PubSub implementation is configured to receive messages generated by our alerting policies. Our "instance failed to deploy" alert uses log-based metrics to track the behavior of our instances as they spin up for the first time. It is then triggered when a new instance template fails to deploy to the MIG. This alert will push a message to PubSub, which then triggers the subscribed cloud functions responsible for initiating the rollback procedures. This integration allows for a swift and automated response to fatal issues, minimizing downtime and the impact on service reliability. These cloud functions act as event-driven serverless components that execute predefined mitigation techniques such as rollback in response to the specified incident scenario.

III. Fault Injections

A. Code Bugs

Code bugs, especially those that manage to slip through the testing stages of the deployment pipeline, pose a unique challenge due to their often subtle and elusive nature. Typically, these code bugs would be detected by observing frequent and prolonged periods of abnormal behavior. Thus, detecting such bugs becomes crucial for maintaining the integrity of a cloud-based application. In our evaluation of automated monitoring rollback, we intentionally injected code bugs to assess the system's response and the effectiveness of automated rollback strategies.

Error rate and mean latency over a longer period can be good indicators of code bugs that made it through the testing process. One illustrative example involved a code bug that prevented a boolean variable responsible for signaling a successful connection to the database from being updated when the server startup procedures were completed. In this example, the key-value storage service would still deploy, and the health check on the load balancer would indicate that all nodes are alive and healthy since the front end would not be impacted by this code bug. The latency monitors would also not detect anything problematic as the response time isn't significantly impacted by this code bug. However, the monitors for the error rates would detect an issue since each request from the load balancer to the backend service would result in an HTTP 500 error. In this situation, if the HTTP 500 response ratio surpasses 15% for more than two minutes, this would be indicative of an issue within the key-value storage service and a rollback to the last stable version would be initiated.

Another example of a code bug that was injected into the system, involved delaying the response time for each request by two seconds. This type of code bug would also make it through the testing stages and would not trigger any response from the error rate monitors. Such code bugs can only be detected through monitoring latency and throughput. However, for this code bug, we do not initiate rollback and instead push a notification to the system administrator. Prolonged periods of higher-than-average latency are not always indicative of a code bug. This could also be a result of an infrastructure failure. If an instance fails during peak times, the latency will spike and remain in this heightened state until the autoscaler creates a new instance in its place. We would not want to initiate a rollback in such scenarios. Instead, we alert the administrator when an abnormality in latency is detected and allow them to decide whether or not to initiate a rollback.

B. Infrastructure Disruptions

To test our system's response to infrastructure disruptions, we deliberately injected disruptions by manually terminating instances during periods of high traffic. This simulation allowed us to assess the real-time effectiveness of the autoscaler and load balancer in a controlled environment, mimicking the unpredictability of actual infrastructure challenges. The key to handling infrastructure disruptions lies in the proactive measures taken by the autoscaler. The autoscaler continuously monitors the health of the instances and if an instance were to abruptly fail, the autoscaler would automatically create a new instance to replace it. The load balancer complements the autoscaler by gracefully rerouting traffic in the event of an instance failure. All traffic that was previously routed to the failed instance is automatically rerouted to the remaining healthy instances using consistent hashing. This dynamic redistribution ensures that the system maintains its responsiveness and availability, even during unexpected infrastructure disruptions.

C. Authentication Errors

Authentication errors pose a critical threat to system functionality, however, they offer clear indicators when the system fails due to incorrect credentials. Our key-value storage service relies on the proper credentials to establish connections with the persistent storage database. Failure to authenticate will result in a subsequent failure in deployment, making authentication errors readily discernible in the system logs. We can set up a log-based metric that queries system logs and detects any time an authentication failure arises. When the log-based authentication metric detects an authentication failure, it initiates an automated rollback. This rollback ensures that the system is reverted to the last stable version, addressing the authentication error immediately and without manual intervention.

IV. Results

This section provides a comprehensive analysis of the key findings from our systematic testing of several mitigation techniques within the context of our key-value storage service deployed through GCP. To evaluate the system's resilience and behavior, we intentionally injected a variety of faults, encompassing code bugs, infrastructure disruptions, and authentication errors. These faults act as simulated challenges to assess the efficacy of our mitigation techniques, such as automated rollback and autoscaling. The subsequent analysis further explores the specific behaviors exhibited by the system during and after the application of these mitigation techniques. By scrutinizing the system's response to various fault scenarios, this section provides insight into the overall reliability and robustness of our cloud-based key-value storage service.

A. Time To Detect

The time it takes to detect faults is a key metric that impacts the overall agility and responsiveness of the system to anomalies and performance degradation. However, the time it takes to detect is inherently dependent on how these monitors are configured to alert. The decision to restest before alerting creates a variable in the time to detect faults. While retesting can contribute to reducing false positives, which is vital for rollback, it comes at the cost of extending the time it takes to detect and respond to actual incidents. For the sake of clarity, we define the time to detect as the time starting from when the fault is injected into the production environment to the time when the alert is triggered.

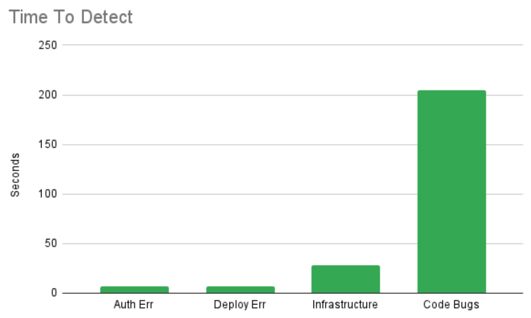

Fig. 11. Graph depicting the average time to detect (A) authentication errors, (B) deployment errors, (C) infrastructure disruptions, and (D) code bugs. (A) Authentication Errors: Incorrect credentials for backend database connection. (B) Deployment Errors: Key-value storage containers fail to build the new image. (C) Infrastructure: Capacity issues due to high traffic, high CPU utilization, and instance downage. (D) Code Bugs: Boolean bug and delayed response time.

For authentication and deployment errors, these faults are detected almost immediately at only 6 seconds. We suspect this small time delay between the fault injection and alert triggering comes from the time it takes for the monitors to update with the new data. For infrastructure disruptions, when an instance goes down, this is immediately detected by the autoscaler. However, this isn't the case for disruptions such as high traffic and CPU utilization. To test the infrastructure monitors, we simulate high-traffic loads by creating numerous clients each sending 1000 requests in a read/write intensive workload. We then record the time from when the high-traffic simulation begins to the time the infrastructure monitors trigger an alert. This results in a slight delay as we wait for the system to become saturated enough to trigger an alert. The code bugs take the most time to detect, at an average of about three minutes. This is a result of how we configured our monitors to retest before alerting. For the error rates monitor, if the error response ratio passes 15%, the alerting policy will retest after two minutes before triggering the alert. In the scenario of a slow latency due to code bugs delaying the response times, the alerting policy will retest after 5 minutes and trigger if the mean latency is above 150ms.

B. Automated Deployment

Automated deployment not only ensures consistency between each deployment but also maintains uptime while the system updates. When testing the automated deployment, we measured the latency over throughput during and after the deployment procedure was completed.

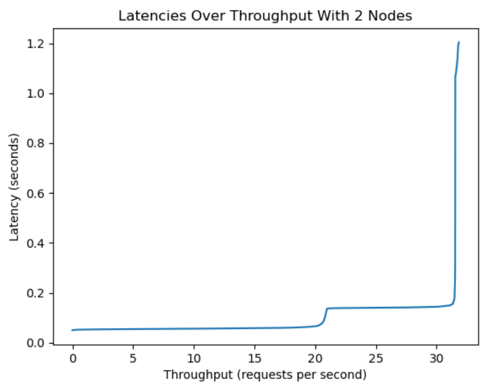

Fig. 12. Latency over throughput during automated deployment procedures with 2 nodes active.

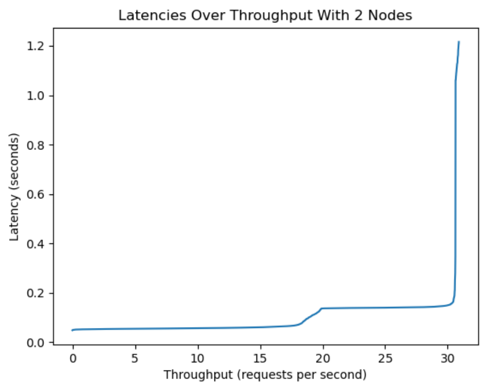

Fig. 13. Latency over throughput after the deployment process has been completed.

In Figure 12, the first spike in latency occurs at only 5 operations per second and approaches the limit at about 25 operations per second. However, in Figure 13, the latency remains about the same but the first spike in latency occurs at about 20 operations per second. The throughput limit also increases from 25 to around 30 operations per second. The difference between the two is a result of how the automated deployment procedures work. Since only two nodes were active at the time of the deployment, the MIG drains one instance from the load balancer pool and replaces it with the previous template used to create instances. This means that only 1 node is active while the deployment is occurring, reducing the application's ability to serve requests.

The throughput could be maintained by increasing the number of instances the deployment procedure is allowed to create. Rather than drain each instance one by one, we can allow the MIG to temporarily surge past the defined instance limit and create all the new instances at once. Once the new instances are running and healthy, the load balancer will then gracefully reroute all traffic to the new pool of instances. This means that we can maintain the same number of instances throughout the entire deployment process. However, doing so means you could potentially be running twice the maximum number of instances you have defined for your MIG. This can be expensive, so the cost of hosting instances should be taken into account when defining your deployment procedures.

C. Automated Rollback

Automated rollback can mitigate authentication errors, deployment errors, and code bugs. To better understand the behavior of the system as a rollback is occurring, we measured the latency over throughput during the rollback process and after the rollback finished.

Since the procedures for automated rollback only differ slightly from those of the automated deployment, we can see that the results are about the same. Similar to the automated deployment, the rollback drains each node one at a time, replacing the instances with an older version rather than a newer version of the system.

D. Autoscaling

Autoscaling is a crucial component of service reliability. To test how the service behaves as the number of instances scale up, we simulated a high-traffic load and measured the latency over throughput during and after the service scaled from 2 to 3 instances.

As we increased the number of instances from 2 to 3, the observed latency remained relatively consistent. The stability in latency is indicative of the efficiency of the load balancer in seamlessly incorporating additional instances without introducing significant delays in processing requests. A notable improvement was observed in the throughput limit. With the introduction of a third instance, the throughput increased from approximately 30 to 35 operations per second. This improvement in throughput highlights the scalability benefits

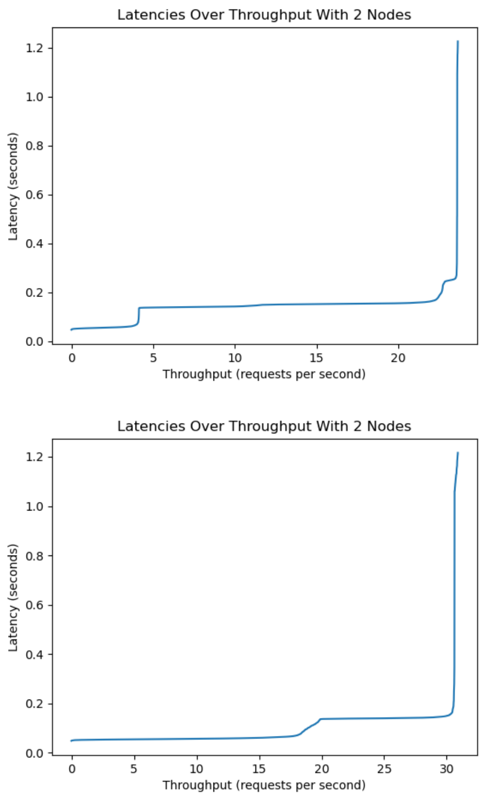

Fig. 14. Latency over throughput during rollback procedures with 2 nodes active. During the rollback procedures, the nodes are drained 1 at a time and then replaced with the new instance template. This rollback was triggered by the HTTP 500 error rate monitor. The second graph illustrates the latency over throughput after the rollback process has been completed.

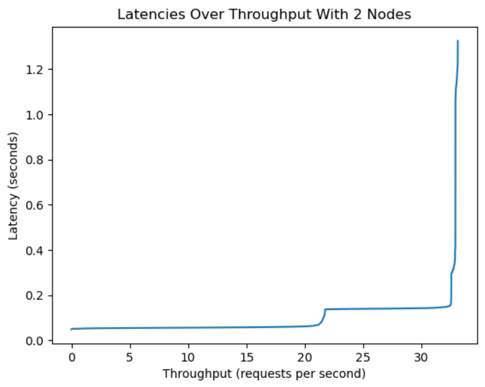

Fig. 15. Latency over throughput with 2 instances.

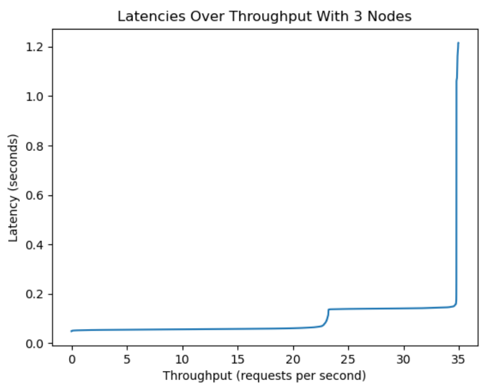

Fig. 16. Latency over throughput with 3 instances.

afforded by autoscaling, allowing the system to handle a greater volume of operations in response to increased demand.

V. Relevant Work

In [2], S. Ghosh et al. present an empirical study on large-scale cloud services and provide valuable insights into the challenges and solutions associated with production incidents in industry cloud environments. The paper specifically focuses on Microsoft Teams as a case study and analysis of recent high-severity incidents and their postmortems. The overarching objectives of the study include answering critical questions such as why incidents occur and how they are resolved, identifying gaps in existing processes that contribute to delayed responses, and determining the role of automation in bolstering service resilience. The paper serves as a foundational reference for our project, providing a context-rich understanding of the challenges and solutions associated with production incidents in large-scale cloud services.

VI. Conclusion

To conclude, this project implemented several mitigation techniques including automated testing, automated deployment, automated rollback, automated monitoring, and autoscaling. These techniques allowed for the mitigation of several injected faults including code bugs, authentication errors, deployment errors, and infrastructure errors. Further, our production service remained reliable during key areas of stress including deployment, rollback, and scaling. Future experiments should be conducted to explore further automated rollback techniques and their impacts on reliability.

References

[1] Betsy Beyer, Chris Jones, Jennifer Petoff, and Niall Richard Murphy. 2016. Monitoring Distributed Systems. In Site Reliability Engineering: How Google Runs Production Systems. O'Reilly.

[2] Supriyo Ghosh, Manish Shetty, Chetan Bansal, and Suman Nath. 2022. How to fight production incidents? an empirical study on a large-scale cloud service. In Proceedings of the 13th Symposium on Cloud Computing (SoCC '22). Association for Computing Machinery, New York, NY, USA, 126–141. https://doi.org/10.1145/3542929.3563482

[3] Atlassian. 2019. The Atlassian Incident Management Handbook. (2019). Retrieved December 14, 2023 from https://www.atlassian.com/incident-management/handbook

[4] Abram Brown. 2022. Facebook lost about $65 million during hours-long outage. (April 2022). Retrieved December 14, 2023 from https://www.forbes.com/sites/abrambrown/2021/10/05/facebook-outage-lost-revenue/?sh=45ebc2d9231a