# Chaos Monkeys

Authors:
Ntense Obono, Catherine Bregou, Angel Ortiz, Sam Lengyel, Khizar Qureshi, Bryan Yang

## 1. Summary

### Abstract

As Netflix blog puts it, imagine “unleashing a wild monkey with a weapon in your data center (or cloud region) to randomly shoot down instances and chew through cables”.  If they could handle such random anomalies during the workday without customer impact, they could be much more confident that their procedures would work.

A vast array of challenges arise when attempting to build a software service at scale. Users may access the service from all over the world, and should see small latencies no matter their location.  Data centers or web endpoints may go down unexpectedly, but traffic must flow.  High traffic from individual users should not impact other users.  And when something goes wrong (and it will), monitoring and alerting will be in place to be able to detect the problem and recover as quickly as possible.

In order to maintain high availability (measured in percentages, such as 99.9999%, with as many “9s” as possible), we will incorporate automated failure handling into their regular workflows with products such as Chaos Monkey.  Instead of hoping the engineering was done right the first time, We will deliberately trigger failures to practice recovering from them and further improve their tooling to automatically handle such failures or send alerts when manual intervention is needed with the aid of status pages made available to the customers

### Problem

We cannot assume that cloud services will always work all of the time.  It’s much better to practice handling failures in a safe environment rather than when you least expect it.

## 2. Background

Our chess game service creates an environment where users can play a single move, and the system calculates the best possible response. The game architecture and structure consists of several components designed to handle user requests, allocate chess boards based on other user's current game states and compute optimal moves.

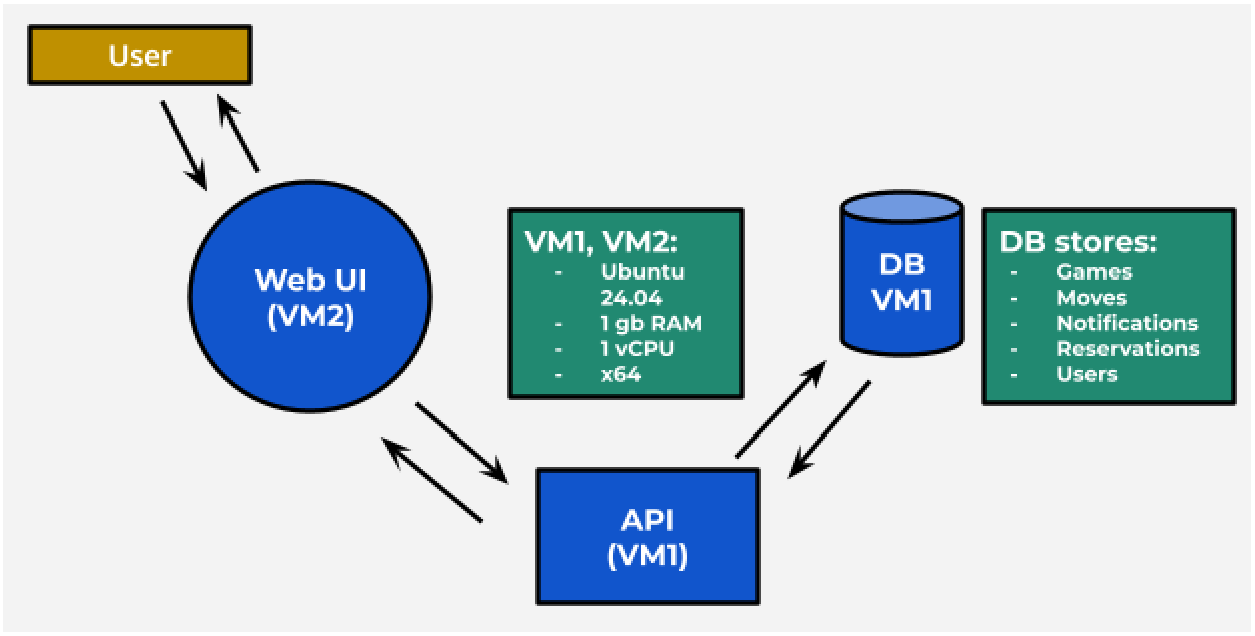


Fig. 1: How OneMoveChess Works

### The service architecture consists of:

- **Web UI (VM2)** for user interaction.
- **API (VM1)** for handling game logic, authentication, and board management.
- **Database (VM1)** that stores game data, including moves, notifications, board states, and user accounts.
- VM1 and VM2 are hosted on Ubuntu 24.04 instances with 1 vCPU and 1 GB RAM.

## Key service functionalities include:

- **User Login:** The system authenticates users and assigns a random password.
- **Bots:** Simulation of human behavior by playing the game.
- **Best Move Algorithm:** The core of the service calculates the optimal chess move based on the user's input.

To ensure this architecture is scalable, robust and capable of handling unexpected disruptions, we are applying chaos engineering principles. The chaos engineering process allows us to identify potential weak points in our system by deliberately introducing failures in a controlled and documented manner.

## Chaos Engineering Steps

1. **Hypothesis:** We identify potential problem statements
2. **Experiment:** We introduce failures to test the hypothesis. We will disable key components such as the production environment to induce failure and observe the consequences.
    a. Board Allocation Failures
    b. Database Connection Drops
    c. Sudden High Traffic Simulation
    d. Database Write Failures During Move Submissions
3. **Evaluation:** We assess the internal and external impact of these disruptions, determining what was affected and how the system responded.
4. **Solution:** Based on the results, we fine-tune the system to improve failure handling, automation, and scalability.

## Goal

We aim to achieve graceful degradation, where the system continues to function under stress without crashing. Real-time monitoring of logs, metrics, and error rates helps validate whether the system self-heals or requires intervention.

## 3. Definitions

a. Status Page Definitions
    i. **External Monitoring**: The assessment of a web service's performance and availability
    ii. **Incidents**: Disruptions affecting a web service's normal operation.
    iii. **Failures**: Instances where a web service fails to operate normally due to errors or outages.
    iv. **Downtime**: Duration of a web service is unavailable, due to planned maintenance or unexpected issues.
    v. **Real-time**: Immediate updates about service status as events occur.
    vi. **Healthy**: The web service functions normally, with no issues or failures reported.
    vii. **Availability**: Whether or not a service can be accessed and utilized by a user.

b. Database Sharding Definitions
    i. **Shards**: A smaller part or fragment of a larger database that operates independently. Multiple shards together represent the entire dataset.
    ii. **Scalability**: The ability of a system to handle growing amounts of data or traffic
    iii. **Vertical Scaling**: Increasing the capacity of a single server or machine by upgrading its hardware (e.g., adding more RAM or a faster CPU) to handle more data or traffic.
    iv. **Horizontal Scaling**: Increasing the capacity of a system by adding more servers or machines to distribute the load. Sharding is an example of horizontal scaling.

v. **Hot Spots**: A situation where certain shards become overloaded because data is unevenly distributed. This can lead to performance issues if one shard receives a disproportionate number of queries or data.

vi. **Hash Function**: A function that converts input data (such as user ID) into a fixed-size value that determines which shard the data will be stored in.

vii. **Latency**: The delay between a user action (such as a query) and the system's response. Reducing latency improves the speed at which users can access data.

c. Throttling and Rate Limiting

i. **Infrastructure as a service (IaaS):** a pay as you go cloud service that buyers can use, largely for computing, storage, and networking.

ii. **Throttling**:  when requests are intentionally delayed so that they are more spread out over time, decreasing load on servers.

iii. **Rate Limiting**: preventing abuse and ensuring equitable access to resources by putting a hard cap on the number of requests a user can make within a certain period of time.

iv. **Over-provision**: when more than enough infrastructure is bought to meet users' requests, leading to resource wastage

v. **Under-provision**: when not enough infrastructure is available to meet users' requests, leading to poor performance.

d. Load Testing

i. **Bottlenecks**: points in the system where performance slows down or fails, usually due to limited capacities or inefficiencies
ii. **Throughput**: the amount of data processed by the system within a specific time period, often used as a performance metric
iii. **Virtual Users:** In load testing, the simulated users that mimic real users on the system in order to test the systems capacity
iv. **Telemetry**: the automated collection and transmission of data from remote sources for monitoring and analysis
v. **Azure Monitor**: a service from Azure that helps monitor the the performance of applications and infrastructure of the system
vi. **ARM templates**: Azure Resource Manager templates used to define the resources needed for a system

e. Fault Injection

i. **Hardware Fault:** A fault simulating a type of hardware failure or malfunction (loss of power, electrical surge, etc).
ii. **Software Fault:** A fault simulating a type of software failure malfunction (call stack error, race condition, systemd/server software failure, etc

## 4. User Stories

a. Something in our database goes down causing our downtime for our web service. Our developers are immediately notified that the web service is down, and they refer to the status page to see what specifically is wrong. They see the red visual indicator on the database section, so they proceed to click the “+” symbol on the DB section. They can find the exact error causing the issue and come up with possible solutions to fix it promptly.

b. Imagine you're a developer for One Move Chess. People are really enjoying playing the game and it starts to grow globally, millions of users start logging in simultaneously to make their move. Each game, with its unique state and user data, is stored in a central database, but the growing number of users begins to overload the system. Database queries to retrieve game states, validate moves, and update the board slow down, especially during peak hours when many users are trying to make their moves concurrently. To handle the load, you decide to implement database sharding. You shard the database by game ID, distributing different games across multiple servers. This ensures that when a user makes a move, the query only affects the specific shard containing that game’s state, significantly reducing the load on each individual server. The result is a smooth, lag-free experience, where users can quickly make their move, and pass the game along, even during high-traffic periods. Database sharding allows OneMove Chess to handle many players while keeping performance intact.

c. Using load testing tools, we simulate high traffic onto the server. This stress on the server causes the machine hosting the API to crash. Once the status page notices that it cannot reach the API, it marks it as “down.” A developer is then notified that the API is down, and the developer can go in and see exactly what crashed within the status page and solve the issue promptly.

d. Our chaos engineering monkey drops a database containing a game that was supposed to be assigned to Alice. Our system automatically assigns a game from another sharded database while notifying our developers to fix the dropped database with an already viable ready clone.

## 5. Timeline

### Fall Term

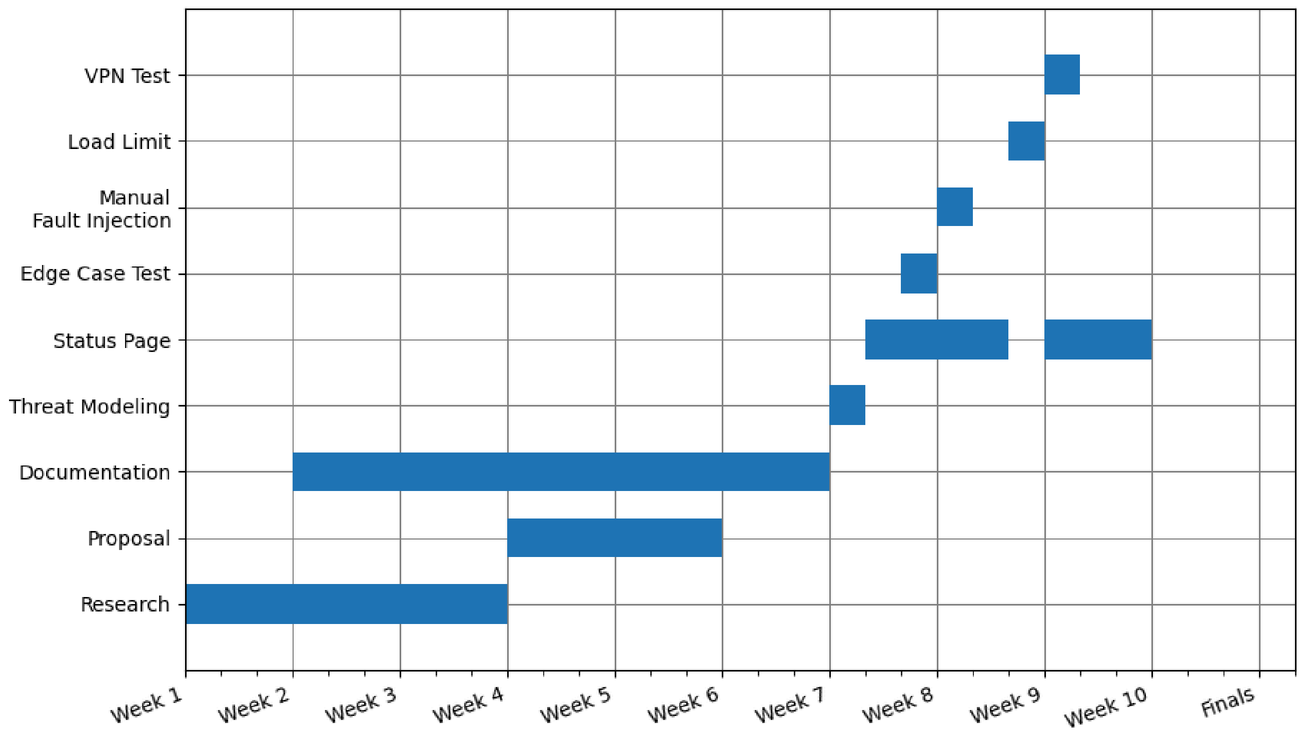


Fig. 2: Timeline of Work

| **Goals for this Term** | **Description** | **Week** |
|---|---|---|
| Research | Familiarize ourselves with the code databases, virtual machines, and learn about Chaos Engineering | Weeks 1 - 3 |
| Documentation for important functions in the codebase | Create a google doc that has an overview of each essential function within our API, comment the whole codebase. | Week 2-6 |
| Proposal | Design mock-up for status pages. Write-up on research findings  and plans on implementations | Weeks 3 - 5 |
| Threat modeling | Identify places where things can go wrong with the different components of our software | Monday of Week 7 |
| Status page: very simple prototype to ping VMs and SSH | Create a simple front end and a simple back end to ping the VMs and check if SSH works. | Begin on Wednesday of week 7 and create smaller teams to work on that. |
| Edge case testing | A session where we all log on and test edge cases: log in on different browsers, etc. | Friday of Week 7 |
| Manual fault injection | Manual fault injection: turning things off in Azure, etc. | Monday of Week 8 |
| Status page: hosting | Experiment with hosting the status page on a Raspberry Pi | Wednesday of Week 8. |
| Load limit | Have a session where we see how much load we can all produce (trying to get a sense of the limit of our VMs) | Friday of Week 8 |
| Test Geographic Differences | To determine if Geographic Sharding makes sense | Monday of Week 9 |
| Status page | Finish up the simple backend implementation | Rest of Week 9 |
| Flexible time | We may need extra time for the status page and want to set ourselves up for winter, so this time will be used to tie up loose ends. | End of fall term. |

Table 1: Descriptive Timeline of Fall Term Work

## Winter Term

| **Goals for this Term** | **Description** | **Week** |
|---|---|---|
| Familiarize | | Weeks 1 |
| | | |
| | | |
| Check-In presentations | | Mon/Wed Week 5 |
| | | |
| Feature Freeze | | Thursday, Week 7 |
| | | |
| Code Freeze | | Thursday, Week 8 |
| | | |
| Gala Presentation | | Thursday, Week 9 |
| Code Thaw | | Friday, Week 9 |
| Oral Presentation | | Week 10 |

### Fault injection

### Automated Healing

Status Page requires intervention.

## 6. Design and Content

### Status Page

a. Objective:

   Provide timely updates about the performance and availability of the One Move Chess web service.

b. How will this help us?

   i. **Incident Management**: By allowing a place to centralize sources of information, incidents, and failures, we will be able to coordinate and resolve the issue.

   ii. **Scalability & Productivity**: By improving our ability to resolve issues, our team will be able to identify patterns to scale and improve our software to prevent incidents and be more productive.

   iii. **Chaos Engineering**: External monitoring and bug detection are essential to chaos engineering. Ensuring that our software is in a good and healthy state before launching our chaos monkeys to break everything is critical to chaos engineering. A status page will allow us to define our ability to account for any unplanned disruptions and incidents.

c. Necessary Components of Our Status Page:

   i. **Current System Status**: Implement an intuitive label or visual indicator to show us the current status of our web services. This needs to be easy and clear to read at a glance.

   ii. **Incident History and Response Time:** Integrate analytics to display historical performance metrics of our web services. This will help assess our scalability improvements over time, highlighting response times and incident frequency trends.

   iii. **Hosted on a separate server**: Ensure the status page is hosted on a dedicated server, allowing continuous access at all times.

iv. **Monitoring and Alerts:** Develop a system to automatically notify our team via email when the status page indicates an error that could lead to service downtime.

v. **Real-Time Updates**: Ensure that the status page updates in real time when an incident occurs, providing immediate visibility to developers.

d. How will we implement this page:

i. **GitHub Repos Exploration:** 100s of companies with examples of their open source code for status pages are available for free. We plan to leverage some of these examples and incorporate their ideas on our status page. Repo: https://github.com/ivbeg/awesome-status-pages

ii. **Integration of Existing Tools:** Another approach that we are interested in exploring is rather than developing our own status page, we can look into incorporating an existing tool to monitor our web services. For instance, Gremlin is a tool that specializes in chaos engineering and service monitoring.

e. Below are our mock-ups for some of the designs we have considered for our status page.

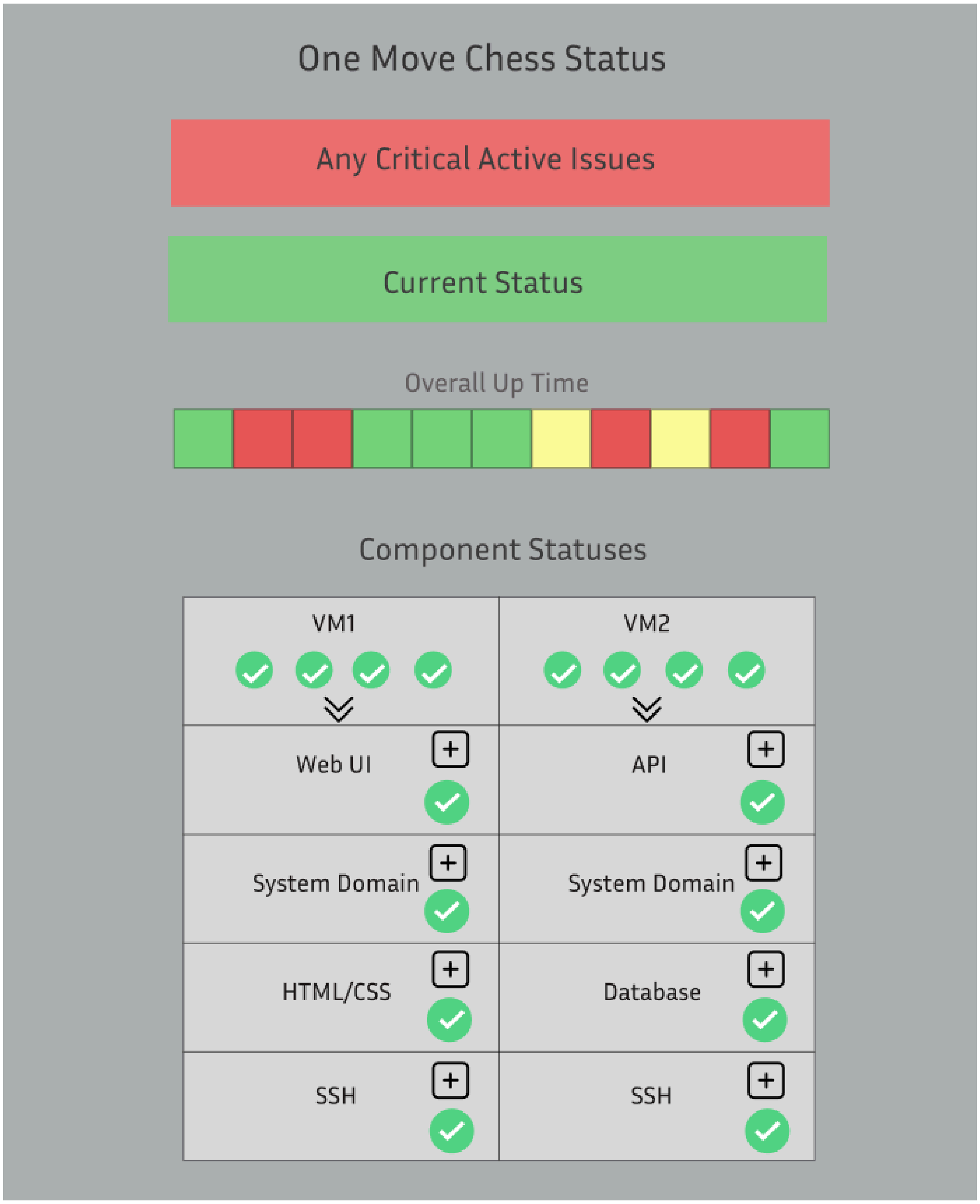
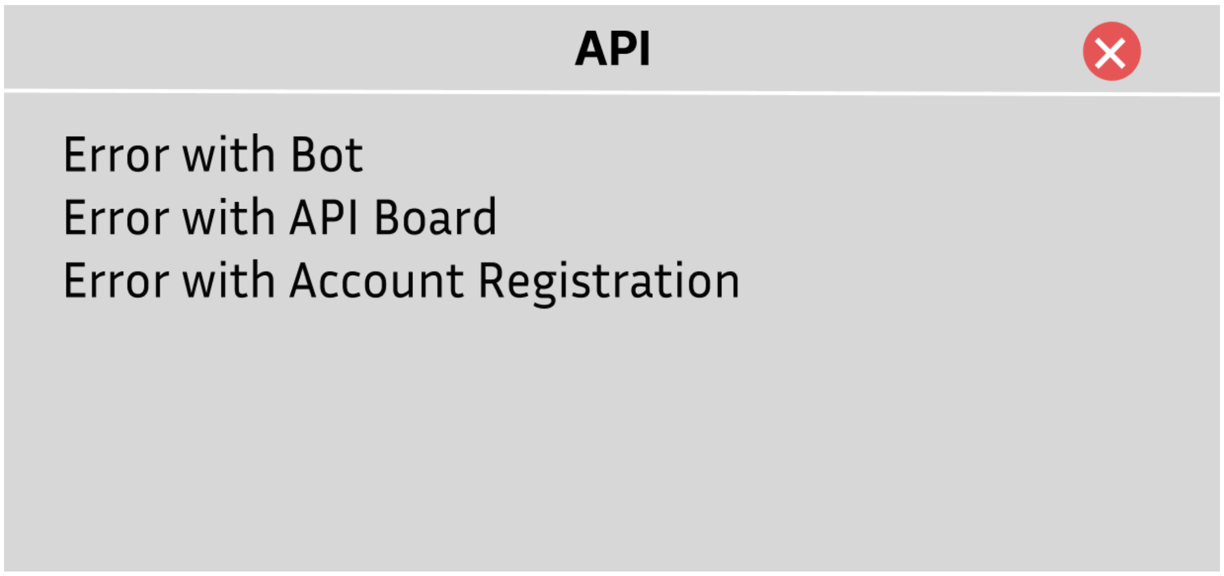

One Move Chess Status
Any Critical Active Issues
Current Status
Overall Up Time
Component Statuses
VM1
VM2
Web UI
API
System Domain
System Domain
HTML/CSS
Database
SSH
SSH

Fig. 3: 0


API

Error with Bot

Error with API Board

Error with Account Registration

Fig. 4: Expanded Status View of Individual Component

### Database Sharding

a. What is Database Sharding?

Database Sharding is a technique that involves splitting a large database into smaller, more manageable pieces (which are called shards).
Each shard operates independently but together forms the complete dataset.

b. Why do we want to incorporate Database Sharding?

i. **Scalability:** One of the primary reasons for sharding is scalability. As data grows, a single database server might struggle to handle the volume and queries efficiently. By breaking the data into shards, you distribute the load across multiple servers, making it possible to handle much larger datasets.

ii. **Cost-Effectiveness:** Rather than scaling vertically (upgrading to a bigger and more expensive server), sharding allows you to scale horizontally by adding more servers as your dataset grows, which is typically more cost-effective.

iii. **Performance:** Sharding improves database performance because each shard can be queried independently. Instead of querying one massive table, you're only querying a smaller subset of the data, which reduces query time.

iv. **Fault Tolerance:** Sharding also offers a level of fault tolerance. If one shard goes down, the others can continue operating, ensuring that your entire system isn't compromised by a single failure.

c. How is Database Sharding Implemented?

i. **Range-Based Sharding:** This method involves dividing the data based on a range of values. For example, we could use  a column like user ID. Users with IDs 1-10,000 go to shard 1, and users 10,001-20,000 go to shard 2, etc. The benefit to Range-Based Sharding is that it is very simple to implement. However, a potential problem with Range-Based Sharding is that it can lead to hot spots, where certain shards become overloaded if the data isn't distributed evenly.

ii. **Hash-Based Sharding:** In this approach, a hash function is applied to a key (such as a user ID) to determine which shard the data should be stored in. The hash function distributes users evenly across all shards, helping to prevent any single shard from becoming a hot spot. For our purposes, this approach works well. However, issues can arise when scaling to a very large system, particularly when adding new shards, as data would need to be rehashed and redistributed.

iii. **Geographic Sharding:** This strategy involves dividing data based on geographic location. For instance, users from North America can be placed in one shard, while users from Europe go into another, and users from Asia in a third shard, etc. This method can optimize performance for globally distributed applications by reducing latency and improving data locality. For our purposes, we could test this by using a VPN to simulate different regions and recording timestamps before and after connection to identify if certain regions experience slower performance.

The database is a critical component to our software, and we need a systematic way to scale it as the project grows. Additionally, when we intentionally take down sections of our software,  we want to  ensure that a single failure does not compromise the entire database, maintaining the system's overall stability and availability. Therefore, Database Sharding is a cost-effective solution to these problems.

## Throttling/Rate Limiting

a. What's the issue?

   i. When scaling via purchasing IaaS(as we are in our comps), there is a need to balance *performance* and *resources* used(i.e. money spent).

      1. Under-provisioning causes poor performance and thus a poor experience for the user
      2. Over-provisioning causes resource wastage

   ii. It is very hard to predict what the maximum rate of traffic a site will ever receive. And trying to figure that out is a witch hunt. The reality is that despite the best balancing of under and over-provisioning, an important aspect of scaling is **dealing with relatively short, high bursts of traffic and requests that are above the current capabilities of our infrastructure.**

b. What is *throttling*?

   i. **Throttling** is when requests are intentionally delayed so that they are more spread out over time. Ideally, this keeps requests below a level(indicated by the gray dotted line labeled "Unacceptable performance threshold) where users do not want to use the site or worse, the server crashes.

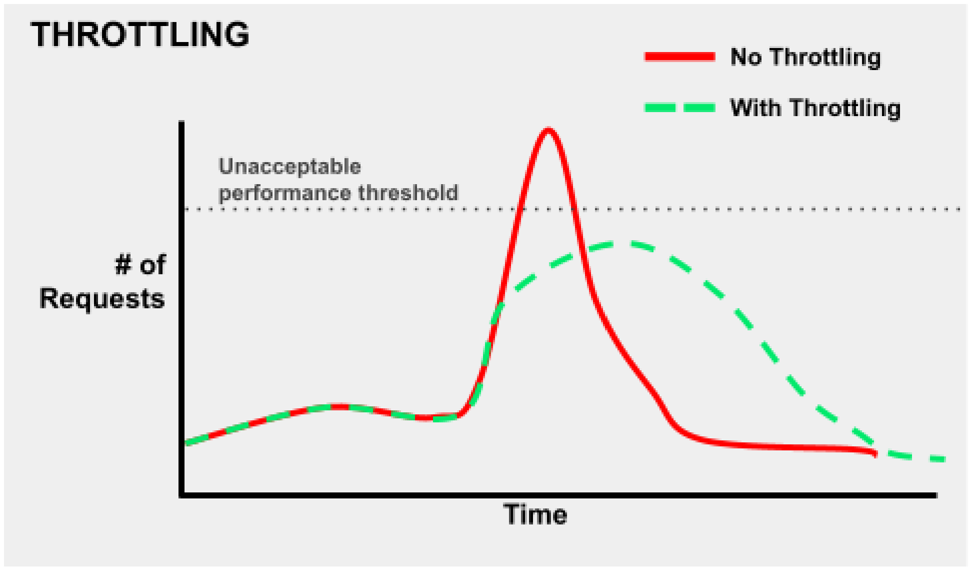


Fig. 5: Throttling

c. What is *rate limiting*?

   i. **Rate limiting** is putting a hard cap on the number of requests a user can make within a certain period of time.

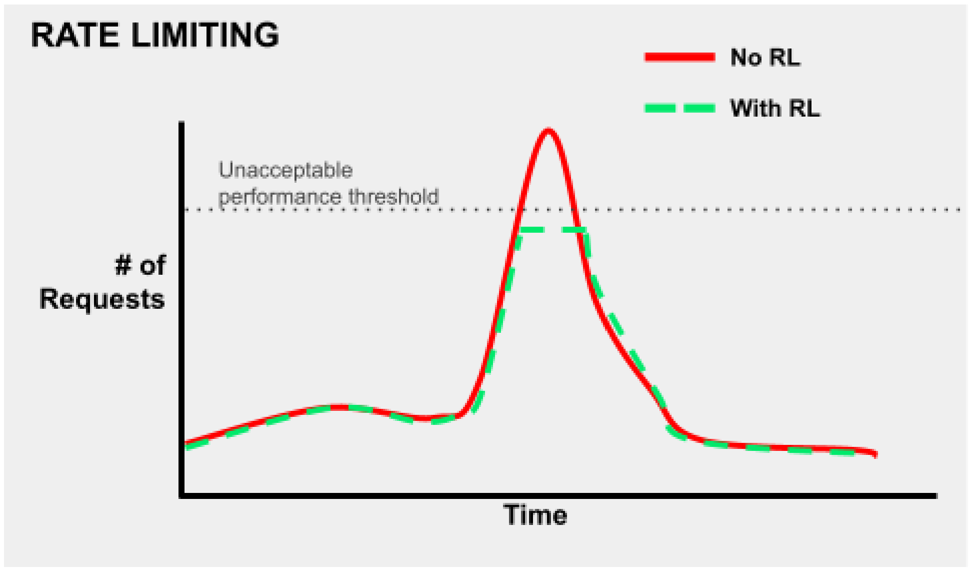


Fig. 6: Rate Limiting

d. Throttling vs Rate Limiting

   i. Throttling does not intentionally drop requests from users. In that sense, it is less severe of an action than rate limiting, which after a certain point blocks any requests.

   ii. Rate Limiting protects against bots, DOS, and DDOS attacks

   iii. Implementing throttling alone does not ensure that traffic will never crash our infrastructure. The nature of rate limiting and its hard cut off gives a more confident solution to preventing extremely large, unmanageable amounts of traffic.

e. How would our project implement this?

   i. Broadly, a combination of throttling and rate limiting is the goal. Throttling would be the initial response to any spike in traffic that stresses our servers, but request rates that are above a specified range(which will be experimentally determined) will be rate limited.

1. We would track the number of requests based on IP and rate limit if the number of requests goes past a human-reachable limit.
2. There are a number ways to strategize the specific types of requests that get throttled and rate limited that would need to be automated. Our main strategy will initially be based on simple, manual stress testing of our servers.
3. Although not directly the mechanism of throttling and rate limiting, some sort of internal autonomous diagnosing/monitoring tool will be set up. (This tool will also eventually be used give information to the status page)

### Load Testing

a. What is load testing?

1. It examines how a system performs during normal and high loads and determines if a system, piece of software, or computing device can handle high loads given a high demand of users

b. Why is it useful?

1. **Discovering bottlenecks before deployment:** Load testing identifies performance constraints under heavy usage, allowing issues to be addressed before the system goes live
2. **Enhances the scalability of the system:** It helps determine how well the system can handle increased user loads, ensuring it can scale effectively as demand grows
3. **Reduces the risk of downtime in a system:** By simulating high traffic, load testing reveals potential failures, minimizing unexpected outages in production
4. **Improved user experience:** It ensures that the system remains responsive and performs well under stress, providing a smooth experience for users.
5. **Reduced failure cost:** Catching performance issues early through load testing prevents costly fixes and losses associated with system crashes or slowdowns in production.

c. How could we implement it into our comps?

1. **Performing Load Testing on Our Own:**
    - Manual Load Testing: For a smaller scale, we can manually stress the system by simulating high traffic through concurrent browser sessions, APIs, or scripts. This approach requires less tooling but is limited in scalability.
2. **Using Azure's Load Testing Tools:**
    1. Azure Load Testing Service: Azure provides a cloud-native load testing service that can simulate high traffic and assess the system's performance under stress. This service integrates easily with Azure-hosted projects, giving real-time insights into performance metrics like response time, throughput, and error rates.
        - **Benefits**: Simplified setup, detailed analytics, and scalability. It can simulate thousands of virtual users, making it ideal for cloud applications.
        - **Implementation**: Configure load testing scenarios for various endpoints of your system. This can be done through the Azure portal or using automation scripts via Azure CLI or ARM templates, allowing the team to schedule regular tests.
    2. Azure Monitor and Application Insights: To monitor the health and performance of our system under load, we can integrate Azure Monitor and Application Insights. This combination helps gather detailed telemetry on app performance and infrastructure health, allowing proactive adjustments.
        - **Benefits**: Provides end-to-end visibility of system performance and automatic detection of performance bottlenecks.

## Fault Injection

a. What is fault injection?

    i. In software testing, fault injection is a technique for improving the coverage of a test by introducing faults to test code paths; in particular error handling code paths that might otherwise rarely be followed.
    ii. Essentially, it is the practice of intentionally introducing (injecting) bugs, glitches, or other failures (faults) into a piece of software or web application.
    iii. Fault injection can be manual or automated, and can focus on software faults (unusual situations within the context of the code itself) or hardware faults (unusual situations within the context of the hardware that the software is running on).
    iv. Hardware faults are harder to implement without direct hardware access, but their results can be simulated fairly effectively via specially-designed software faults designed to simulate loss of/damage to hardware without actually damaging the hardware in question.

b. Why perform fault injection?
    **i. Improved reliability/testing capabilities**
    - Fault injection allows testing of issues with code by introducing situations that would be difficult to produce organically under normal operation of the software.

    **ii. Testing without user reporting**
    - Standard bug reporting requires user action and response for each bug found - fault injection gives the programmer the ability to create, test, and resolve bugs.

c. Determining faults to inject:
    i. Software and hardware faults commonly-found in other pieces of software or hardware can be injected.
    ii. An innovative approach called iBiR: Bug-report-driven Fault Injection proposes parsing bug reports to determine faults to inject. This does require bug reports.

d. Implementation into our Comps:

i. The “Chaos Monkey” part of the project will inject software faults into our web application OneMoveChess and the software running on the VMs, and simulate hardware faults on VM1 and VM2, or other hardware that we run the code on

## 7. Conclusion

At the end of this project, our goal is to deliver a demo or video that clearly demonstrates how our changes have improved the scalability and resilience of our software. We'll show a before and after comparison using bots to simulate traffic, first with throttling off to observe the strain on the system, and then with throttling on to highlight how the system maintains performance under high demand. We'll also demonstrate how our system handles high query loads effectively through database sharding, ensuring fast response times even with increased traffic. Lastly, we'll introduce fault injection—such as turning off a VM in Azure or modifying the database—to illustrate how our software fails gracefully. Through all of these examples, we will demonstrate how our status page correctly identifies issues and provides visibility into the system's behavior.  This demo will show how our additions allow the software to perform efficiently and remain resilient under real-world conditions.

## 8. References

### a. Status page

Instatus, “Our 10 Step Guide on How to Create an Internal Status Page.” https://instatus.com/blog/internal-status-page-guide

Hostko Blog, “Ultimate Guide to Status Pages.” https://www.hostko.com/blog/network/the-ultimate-guide-to-status-pages-benefits-tools-and-best-practices/#:~:text=Best%20Practices%20for%20Status%20Pages&text=Be%20Transparent%3A%20Your%20status%20page,in%20your%20status%20page%20updates.

Gremlin, “Announcing Status Checks to ensure safe chaos engineering scenarios.” https://www.gremlin.com/blog/announcing-status-checks-to-ensure-safe-chaos-engineering-scenarios

### b. Database Sharding:

Amazon Web Services, Inc. “What Is Sharding? - Database Sharding Explained - AWS.” Accessed October 4, 2024. https://aws.amazon.com/what-is/database-sharding/

RobBagby. “Sharding Pattern - Azure Architecture Center.” Accessed October 4, 2024. https://learn.microsoft.com/en-us/azure/architecture/patterns/sharding

“Understanding Database Sharding | DigitalOcean.” Accessed October 4, 2024. https://www.digitalocean.com/community/tutorials/understanding-database-sharding

## c. Throttling/Rate Limiting

[1]

Elijah Asaolu. 2024. Rate limiting vs. throttling and other API traffic management. *LogRocket Blog*. Retrieved October 18, 2024 from https://blog.logrocket.com/advanced-guide-rate-limiting-api-traffic-management/

[2]

dlepow. 2023. Advanced request throttling with Azure API Management. Retrieved October 18, 2024 from https://learn.microsoft.com/en-us/azure/api-management/api-management-sample-flexible-throttling

[3]

mumian. 2024. Request limits and throttling - Azure Resource Manager. Retrieved October 18, 2024 from https://learn.microsoft.com/en-us/azure/azure-resource-manager/management/request-limits-and-throttling

## d. Fault Injection

Fault injection - Wikipedia, https://en.wikipedia.org/wiki/Fault_injection

Ahmed Khanfir, Anil Koyuncu, Mike Papadakis, Maxime Cordy, Tegawende F. Bissyandé, Jacques Klein, and Yves Le Traon. 2023. iBiR: Bug-report-driven Fault Injection. ACM Trans. Softw. Eng. Methodol. 32, 2 (March 2023), 33:1-33:31. https://doi.org/10.1145/3542946

## e. Load Testing

“Performance Testing vs. Load Testing vs. Stress Testing” https://www.blazemeter.com/blog/performance-testing-vs-load-testing-vs-stress-testing#:~:text=Load%20testing%20is%20a%20type,systems%20handle%20expected%20load%20volumes.

"What is Azure Load Testing?"

https://learn.microsoft.com/en-us/azure/load-testing/overview-what-is-azure-load-testing