Investigating Resiliency and Chaos Engineering Principles in Distributed Systems

Luke Waterhouse (190583697)

May 2022

Bsc. Computer Science
Supervisor: Raj Ranjan

Word Count: 14996

Contents

Abstract

This paper investigates the application of chaos engineering on a custom built distributed web application to outline tools and approaches. Background research on chaos engineering, monitoring and emergent failure is presented. The paper identifies main components of the system built before discussing Locust to simulate traffic and monitoring systems to observe system behavior such as Cadvisor, Prometheus and Grafana. Chaos engineering platform Gremlin is used to inject stress on the system regarding CPU and Memory exhaustion attacks. The results of attacks are evaluated, most do not exhibit emergent failure however there are some unexpected rises in CPU when Memory is overloaded possibly due to ‘Pagefile’. API interactions are also discussed and compared with timeouts having significant impact on response times. On memory exhaustion the systems database service crashes, causing considerable collateral failures. Further research such as distributing the system on multiple nodes is discussed in the conclusion.

Acknowledgements

I’d like to thank my supervisor Raj Ranjan for supporting me throughout the project. I would also like to thank Stanly Wilson Palathingal for helping me with issues relating to Locust and Cadvisor. I’d like to thank Ayman Noor for his guidance. I’d like to thank Vincent Gayle and Jason Yee from the Gremlin team for answering my setup questions on their slack page.

Declaration

I declare that this dissertation is my own work except where otherwise stated

Introduction

Scene setting

Chaos monkey

To provide context for the origin of ‘Chaos Engineering’ I will briefly outline how it came about. In 2008, *Netflix* made the decision to transfer all data to the cloud, unlike today *AWS* was relatively new and there were still occurrences where instances crashed without warning. Due to the nature of cloud hosting, components were now being horizontally scaled leading to a decrease in single points of failure [1]. From this vulnerability to instance failure, ‘Chaos Monkey’ was born. This application would go through instances *Netflix* was running and randomly shut them off during business hours without warning. This approach led to engineers being able to respond to and fix emergent failures that came about quickly, ultimately leading to increased resilience of the system as it brought potential failures to the forefront to be dealt with. This was taken a step further to test resilience whilst a whole AWS region went down, simulated in an activity called ‘Chaos Kong’.

Principles

As more organizations adopted this technique, the practice was formalized into ‘Chaos Engineering’ and its principles more officially outlined. The process aligns itself with typical experimental technique and is widely agreed as follows [2]:

- Build a hypothesis around steady state behavior
- Vary real world events
- Run experiments in production
- Automate experiments to run continuously
- Minimize the blast radius

Steady State

Steady state hypothesis can take various forms and shouldn’t be simply defined as ‘everything works correctly’. An example of a steady state hypothesis (if a distributed system is built so microservices work independently) could be ‘if a service's containers are taken down, other services continue to serve requests at steady rate’. Another example suggested by the Gremlin framework is ‘upon resource exhaustion of containers, rate of good responses goes down, errors increase at all layers, load balancer routes traffic away‘ [6].

Introduce Chaos

Once a steady state hypothesis has been identified, ‘chaos’ is introduced into the system. This could take a variety of forms, such as exhausting CPU resources of a container, simulating network failure/packet loss, saturating a database and shutting down services entirely. It is then up to the experimenter to analyze whether the initial steady state hypothesis has been proven or disproven. There is a further step involving ‘minimizing blast radius’ which involves healing the system and the impact of the emergent failure.

To more deeply understand approaches and tools that can be used to experiment on systems to test their resilience, I’ll use a custom built distributed web application to run experiments on. This will let me understand interactions of my components more deeply. I will use a variety of tools to examine the resilience of this system as well as run chaos experiments.

Objectives

- Become sufficiently aware of tooling and approaches for fault injection into multi service applications through reading papers and doing research (this will be ongoing during the project as I may discover/think of new ways of investigating the applications resilience)

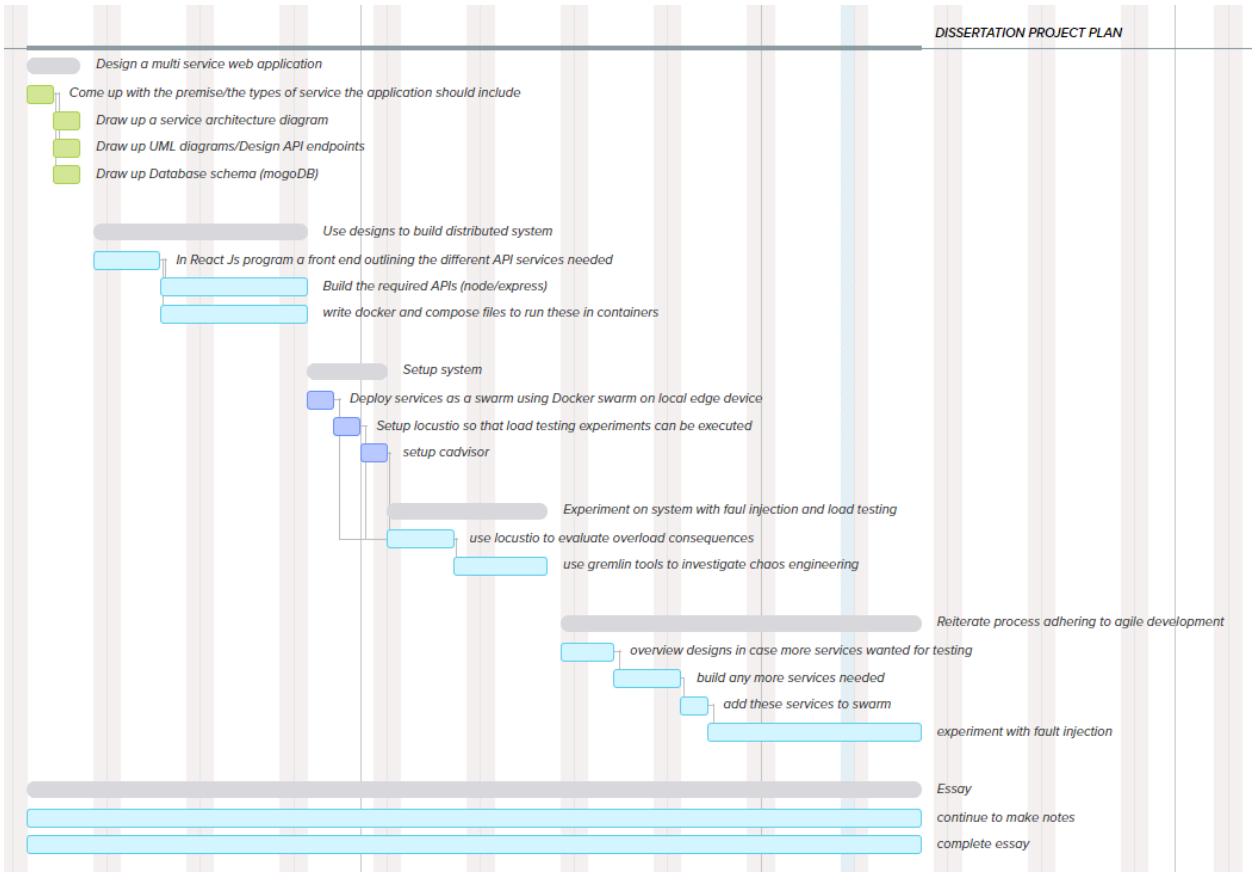
- Build a multi service web application using docker swarm to deploy, with appropriate components to lend itself to faults involving CPU, memory, network and component interaction.
    - Come up with a simple application idea (idea not so important as sufficient processes for fault testing included). I will use node/express for apis and react js for front end.
    - Draw up basic design diagrams for the different services/schemas
    - Program the different services, at least 2 different apis with CRUD functionality, also must have a login/reg for user specific data.
    - Deploy with Docker swarm on edge device (local machine) ready for fault testing.

- Through use of load testing tools (locust) investigate the consequences of overloading the system in various ways.
    - Use of Docker file constraints to change the container constraints such as cpu and memory will aid this.
    - Use of cadvisor to monitor container performance.
    - Use of Docker Visualiser to monitor container status.
    - Do a minimum of 2 overloading experiments with Locust.

- Investigate practise of chaos engineering and use relevant tools such as Gremlin to randomly inject fault into system to evaluate potential vulnerabilities from this information.

- Perform at least 2 chaos experiments with Gremlin.
- I may demonstrate other tools and techniques based on the time that I have.

Gantt Chart



I plan to use agile development to carry out work in my project, this involves cycles of creating the distributed web application to run tests on, running resilience tests and chaos engineering experiments. As I run tests I may discover other components that would be useful to build which will be possible using agile development if I get through at least two iterations.

What's changed

Regarding my initial proposal there have not been any explicit *changes*. The ones you see in the section above, I wrote in the original proposal. I have *deviated* slightly in some areas during

the project, but this was not a result of *changing* my objectives which I have discussed in more depth in the appropriate section of the conclusion.

I changed my Gannt chart however after getting feedback about adding dependencies, as shown in gray lines. This mainly includes things like starting Locust tests once the APIs are complete and using gremlin tools once initial Locust tests are complete.

Regarding deviations this mainly included the way I used Locust; focusing on component interaction rather than overloading with it, not using Docker visualiser much and instead relying on Grafana, carrying out more experiments than expected and focusing on CPU and Memory attacks rather than network. I will go into more detail in the conclusion.

Background Review

Emergent Failure

Defining Emergent Failure

Before diving into literature surrounding chaos engineering I will first look at some reasons it's needed; specifically emergent failures. A definition of emergent failure [4] is presented here:

'Emergent failures are types of failure characterized by their manifestation within constantly changing error propagation boundaries intersecting hardware and software components, their potential to be transient, and are only identifiable at system run-time.'

With rising user demand in large scale applications like at *Netflix* or *Facebook*, systems that enable them have become more complex and microservices that run them have increased heterogeneity. It has become impossible for one software architect to model these huge systems themselves and fully understand all moving parts. A consequence of the increase in complex systems is that the traditional role of software architect becomes less relevant over time [1]. This subsequently means software engineers have greater involvement and freedom in design of the systems themselves and individual services become more independently driven. This is one root of complex interactions between services that can give rise to unforeseeable failures.

Many emergent failures arise in cloud datacenters and systems, and although I'm assessing emergent failures in the context of distributed web applications on my local machine, it's still worth examining these issues.

Dynamicity

One source of failure comes from the dynamic nature of cloud systems. Workload fluctuates based on users needs, therefore conditions of failure vary considerably making it harder to predict the exact nature of failure. Dynamicity comes not only from CPU workload but also server power consumption, network traffic and even temperature. This dynamicity is exaggerated by the heterogeneity of system architectures, different types of processor, network configurations and more [4]. It is unlikely I will encounter emergent failure in my local system from the latter points as I'm running all services on the same machine and processor, however it is possible that as I vary workload intensity there will be interference between services or other issues I didn't predict.

The variety of error roots means that both software and hardware errors can interact leading to more complex emergent failures. As systems become larger the number of complex interactions between components increases, meaning the larger the system, the more potential emergent failures arise.

Container Interference

Since I am using docker swarm to organize containers and services on my local device, I should assess the possibility of interference between the different services running. Docker makes use of virtualisation to enable engineers to replicate environments and their programs running on the same underlying operating system (virtualized). This is important when replicating containers across multiple nodes for horizontal scaling in large systems that require lots of processing power. Nodes running multiple containers or services however all still use the same underlying resources and even by constraining the resource usage of a container it is still possible for containers to limit the output of others if one or more of them is under much load, leading to an increase in failed requests and longer execution times.

A study on interfering microservices [8] highlights the effects of running microservices on the same host machine, both in the same container as well as separate. It shows interesting results.

Regarding CPU performance it was found whilst running CPU intensive microservices there is less interference when running them in separate containers rather than the same with cgroup constraints enabled for both (found that disabling cgroups hence allowing containers to use extra resources not used by others increased performance slightly).

Something similar was found for network intensive microservices with intra container interference being much more impactful than inter container. This could be important when trying to limit emergent failures. As discussed before, competition for resources between containers is often a contributing factor to failure therefore organizing services in efficient ways on nodes is important. Regarding these findings I should ensure any CPU or network intensive microservices are running on separate containers on my machine.

It is possible to allow container orchestration tools to automatically create new containers or VMs to handle increased workload, it is possible, through creating more containers on the same host, other services running on that machine could be interfered with and their performance degraded [1].

Tailing Behavior

Tailing behavior involves a job executing slowly compared to other jobs. Resource managers allocate jobs to different machines or containers, however sometimes within a job certain computations or ‘tasks’ take considerably longer than others.

It has been identified that in large scale systems in production around 4-6% of task stragglers negatively affect over 50% of overall jobs [3]. Stragglers can occur for a variety of reasons and become more apparent due to reasons I have previously discussed such as hardware interference, component interactions and multitenant servers. Tailing can occur when processing requests are not handled well or if timeouts occur and error handling is inefficient. Some jobs rely on other jobs (task dependency) causing long execution times [5].

Some issues regarding emergent failures I have discussed are somewhat abstracted away from the engineer if they are using cloud services such as *AWS* or *Azure* as it is unlikely they will be dealing with hardware and some scaling is handled automatically.

Timeouts

It is possible that an engineer will configure their own load balancer when working on the backend of their distributed web application. This may include functionality to redirect requests to working services or servers if another goes down.

Over time if there are inefficiencies in code or unforeseen complex interactions between services, as the database becomes larger, request times become longer. If timeout length is relatively short, the application stops responding to requests entirely.

These factors may seem obvious however chosen timeouts or data retrieval times (and their interactions) may not have seemed erroneous during development [7]. Although simple, this brief example shows how emergent failure could occur in production.

Regarding distributed web applications made up of microservices similar to mine there are certain design practices for combatting failures involved in these systems I should bring to light. Services are often REST APIs making use of HTTP requests over the network to communicate with each other. These include timeouts for API calls to maintain responsiveness and release associated resources and bounded retries that retry calls, expecting that errors may be

temporary [13]. I shall keep these in consideration when assessing potential reasons for failures in my system.

It is somewhat hard to identify specifics of the emergent failures that will occur in my system as inherently they are emergent and I won't realize them until monitoring under load and even this won't truly simulate a production environment. Emergent failures in my system if there are any will likely be partly intertwined with the code I write.

Monitoring

To obtain meaningful results for my tests I must monitor the behavior of my system, this could include failure rate and response times of requests, the health of containers regarding their status, CPU usage and memory.

In large scale distributed systems run on the cloud, monitoring effectively can become difficult. A system may use different data centers and varying degrees of security across components coupled with complex interaction making it difficult to continuously monitor and find roots of failure with probes. Continuous collection and reporting of monitoring data can overload a cloud node, therefore intelligent communication and management between monitoring services must be implemented and unuseful data filtered out [9].

Locust for Monitoring

Locust is primarily used for load testing and will be useful for investigating resiliency of my system regarding its ability to handle large numbers of concurrent users. It does however also have a UI where one can see metrics such as requests per second, failed requests and response times. These can all be downloaded in graph format [9]. This won't be the primary way I monitor, mainly because the software doesn't show data such as CPU usage, however it is useful for spotting erroneous behavior during initial testing which I can further investigate with more powerful monitoring tools. I will be using its main functionality; sending HTTP requests to my endpoints en masse.

Cadvisor

Regarding Docker there are a few ways one can extract monitoring data. Performance statistics for docker containers are stored in the virtual file system. This data can be accessed using the 'docker stats' docker commands or by interacting with the 'Docker Remote API' [11]. This is what Cadvisor takes advantage of to track resource usage of containers. This alone cannot show historical data. Tools such as Prometheus can scrape and store metrics from Cadvisor, Grafana can query this data to show aesthetic graphs and data visualization to provide more complete insight.

Experimentation

I have already introduced and outlined the basics of running chaos experiments in the introduction and defined the experimental technique. I have done further reading on some tools and approaches in this field.
A study [12] provides a tested framework for implementing chaos engineering in an organization. It proposes implementation can be broken into 4 steps; Discovery, Implementation, Sophistication and Expansion.

Discovery involves defining the system and technologies used in it. This allows you to determine the chaos engineering tools to use (in my case my containers will run using Docker therefore Gremlin is an appropriate tool). One can then determine a set of experiments to run and prioritize them based on likelihood of this error occurring in the system.

During Implementation a first experiment can be selected. If there is no prioritization in place, it may be sensible to pick the most simple. Then a steady state hypothesis must be defined. On a blog source from Capital One exploring Devops integration of chaos engineering it's stated both technical and business metrics can be used to define steady state in order to most effectively consider the direct impact on users [14]. Technical metrics could include parameters like latency, error rate and CPU usage whilst business metrics may include logins per minute during peak or number of failed logins. A metric *Netflix* use is streams per second (SPS) which normally varies throughout the day however certain unexpected fluctuations can be cause for concern and indicate current health of the system [2]. This allows you to write the steady state hypothesis; as the experiment is run, the steady state hypothesis will not change. One can then run the experiment and monitor the system to determine if the hypothesis has been disproven.

Sophistication involves assessing the experiment and suggesting improvements like whether the experiment is run in production or simulated traffic, the automation and extensiveness of metrics used. All these factors can be assessed and improved in further iterations. The expansion part involves finding new experiments to run and new tools. I won't go into much detail about this.

Implementation

System Overview

The first task to undertake was building a basic system to experiment on. I will outline the main components of this system as it will give context when explaining tests I have run. The system comprises a front end built in React js and three backend node/express REST APIs which serve the front end through http requests. They also communicate with each other somewhat. These

components run on the machine as docker services orchestrated by docker swarm. I will show some UI to give a feel for the system functionality in figures 1-4.

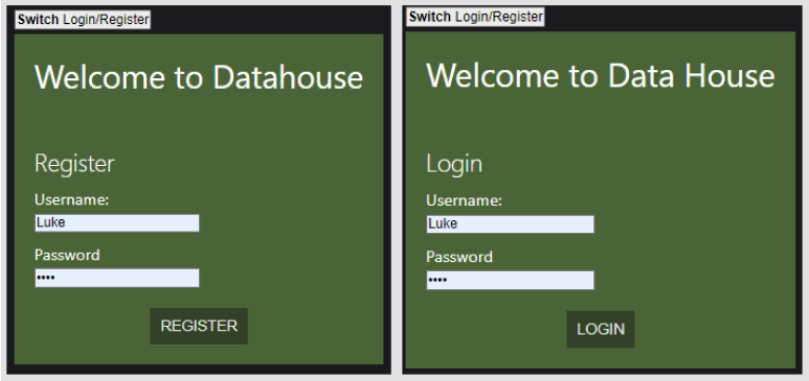


Figure 1: Login and Register, user can switch between them using the button in top left.

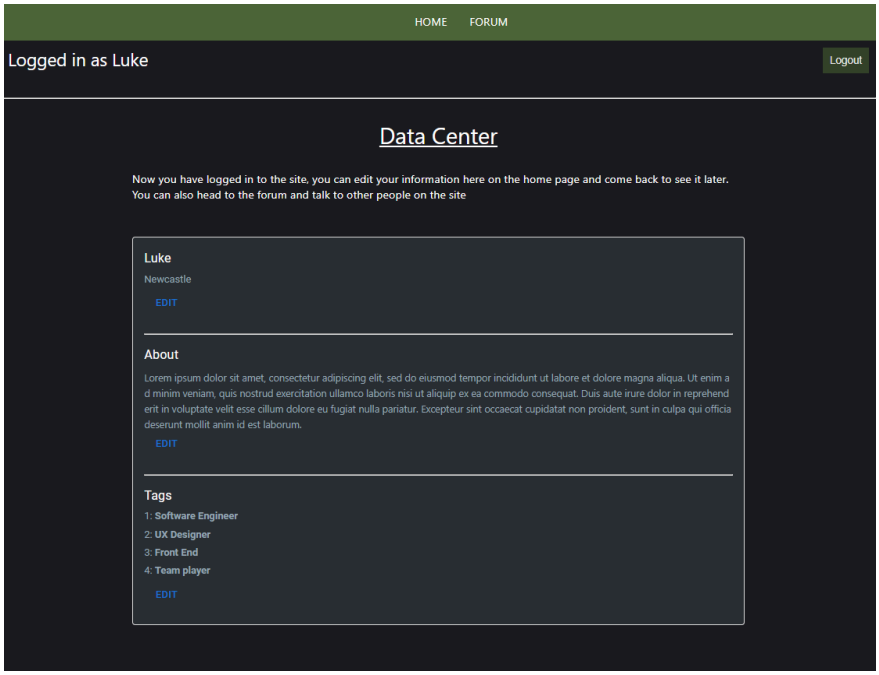


Figure 2: Homepage where users can logout in top right, edit their personal data and navigate the website using the navigation bar at the top

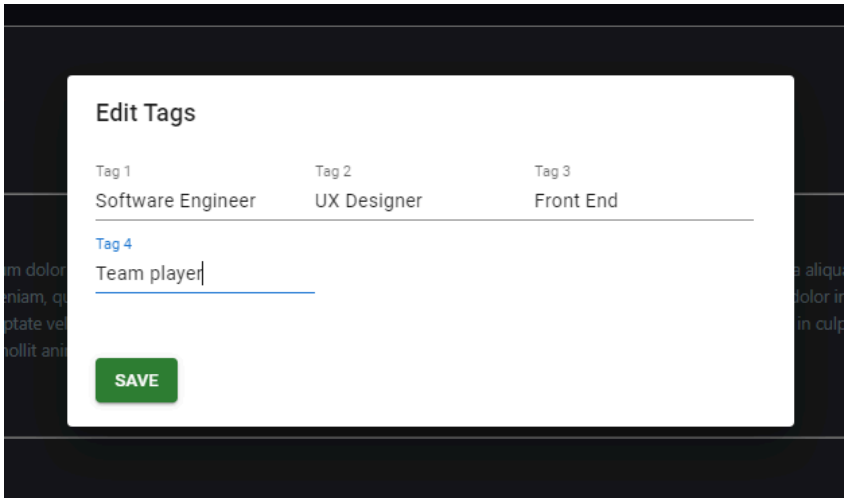


Figure 3: example of the edit pop up when selecting edit on any of the available profile categories. Saving will send a request to the corresponding service to update the user information.

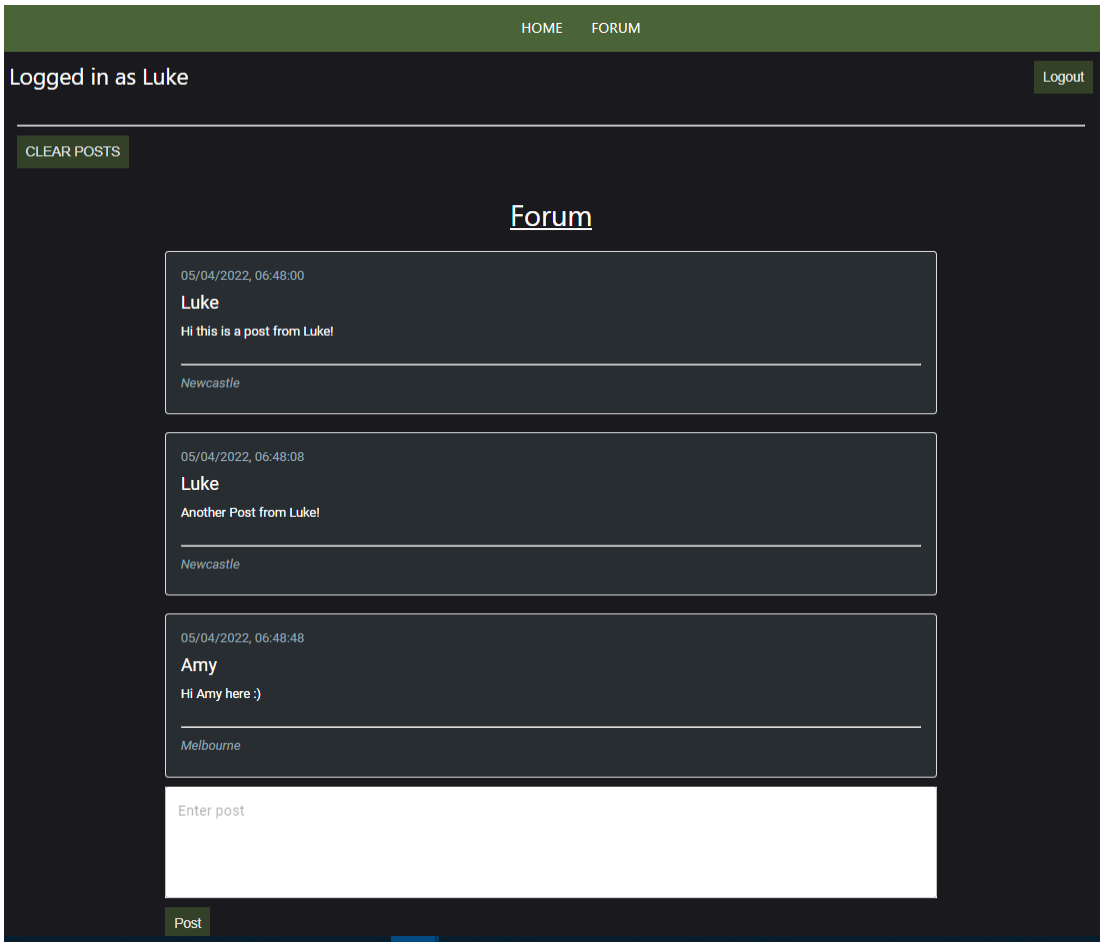


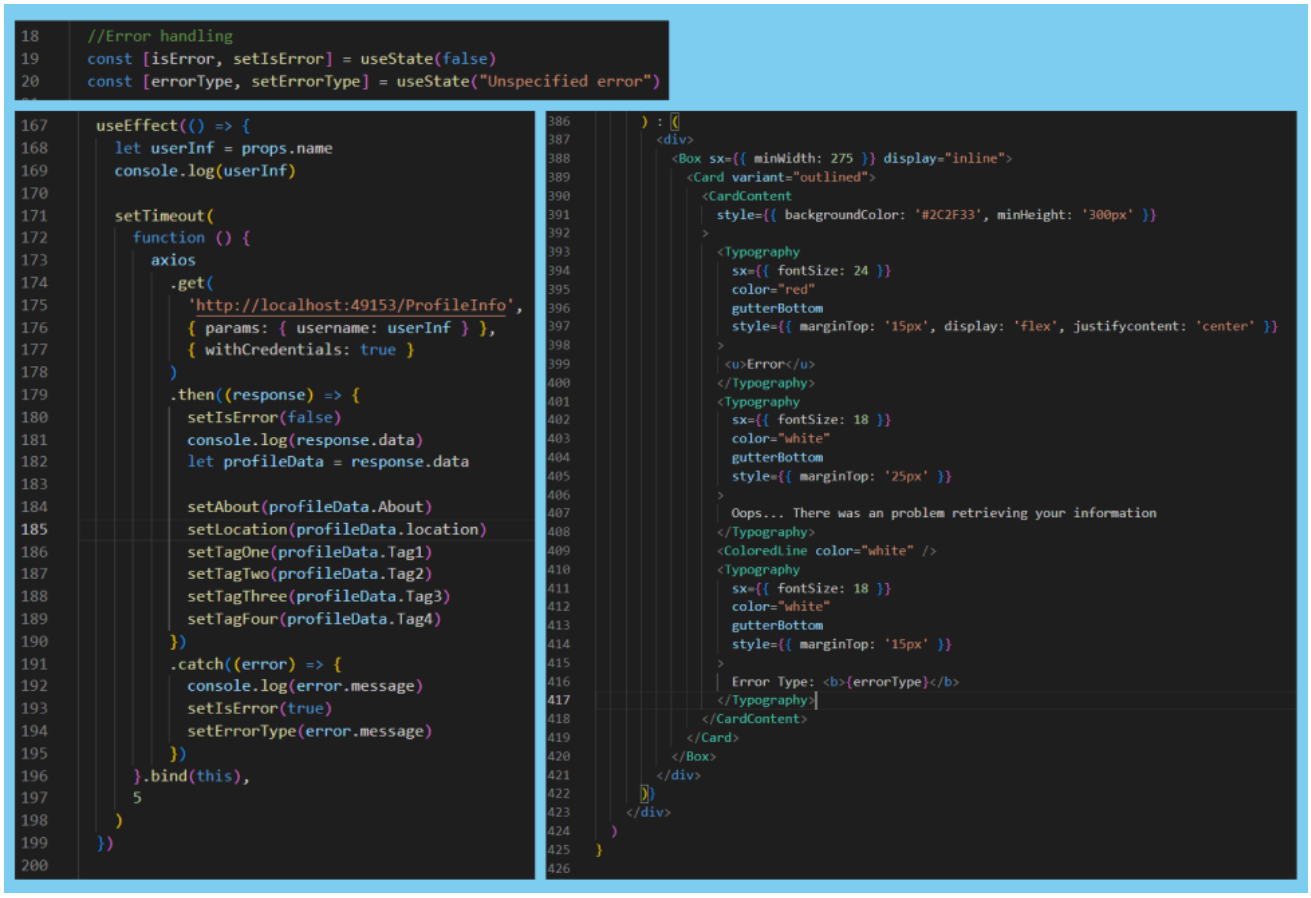
Figure 4: User interface for the forum page.

Error handling

Due to my project using multiple backend services I wanted to ensure if any went down the error would not crash the frontend and would display appropriate error components. To demonstrate this I manually shut down API3 (handling profile information), as seen in figure 6 an error component is conditionally rendered when an error is returned from the API3 request.

Code handling this is shown in figure 5, I make use of hooks to define two states regarding error handling (line 18). When an error is returned from the API request the error is caught by a catch statement (line 191) where error states are appropriately set.

When state is set, the page is re rendered and rather than the user profile, a conditional render (line 386) displays the error message shown, this includes the error type which was taken from the error response. There is more code that can be discussed however much of this is not as relevant as results.



```
18  //Error handling
19  const [isError, setIsError] = useState(false)
20  const [errorType, setErrorType] = useState("Unspecified error")
```

```
167 useEffect(() => {
168   let userInf = props.name
169   console.log(userInf)
170
171   setTimeout(
172     function () {
173       axios
174         .get(
175           'http://localhost:49153/ProfileInfo',
176           { params: { username: userInf } },
177           { withCredentials: true }
178         )
179         .then((response) => {
180           setIsError(false)
181           console.log(response.data)
182           let profileData = response.data
183
184           setAbout(profileData.About)
185           setLocation(profileData.location)
186           setTagOne(profileData.Tag1)
187           setTagTwo(profileData.Tag2)
188           setTagThree(profileData.Tag3)
189           setTagFour(profileData.Tag4)
190         })
191         .catch((error) => {
192           console.log(error.message)
193           setIsError(true)
194           setErrorType(error.message)
195         })
196     }.bind(this),
197     5
198   )
199 })
200
```

```
386     ) : (
387       <div>
388         <Box sx={{ minWidth: 275 }} display="inline">
389           <Card variant="outlined">
390             <CardContent
391               style={{ backgroundColor: '#2C2F33', minHeight: '300px' }}
392             >
393               <Typography
394                 sx={{ fontSize: 24 }}
395                 color="red"
396                 gutterBottom
397                 style={{ marginTop: '15px', display: 'flex', justifycontent: 'center' }}
398               >
399                 <u>Error</u>
400               </Typography>
401               <Typography
402                 sx={{ fontSize: 18 }}
403                 color="white"
404                 gutterBottom
405                 style={{ marginTop: '25px' }}
406               >
407                 Oops... There was an problem retrieving your information
408               </Typography>
409               <ColoredLine color="white" />
410               <Typography
411                 sx={{ fontSize: 18 }}
412                 color="white"
413                 gutterBottom
414                 style={{ marginTop: '15px' }}
415               >
416                 Error Type: <b>{errorType}</b>
417               </Typography>
418             </CardContent>
419           </Card>
420         </Box>
421       </div>
422     )}
423   </div>
424 )
425 }
426
```

Figure 5: Code relevant to the conditionally rendering the error component when API3 is not behaving correctly

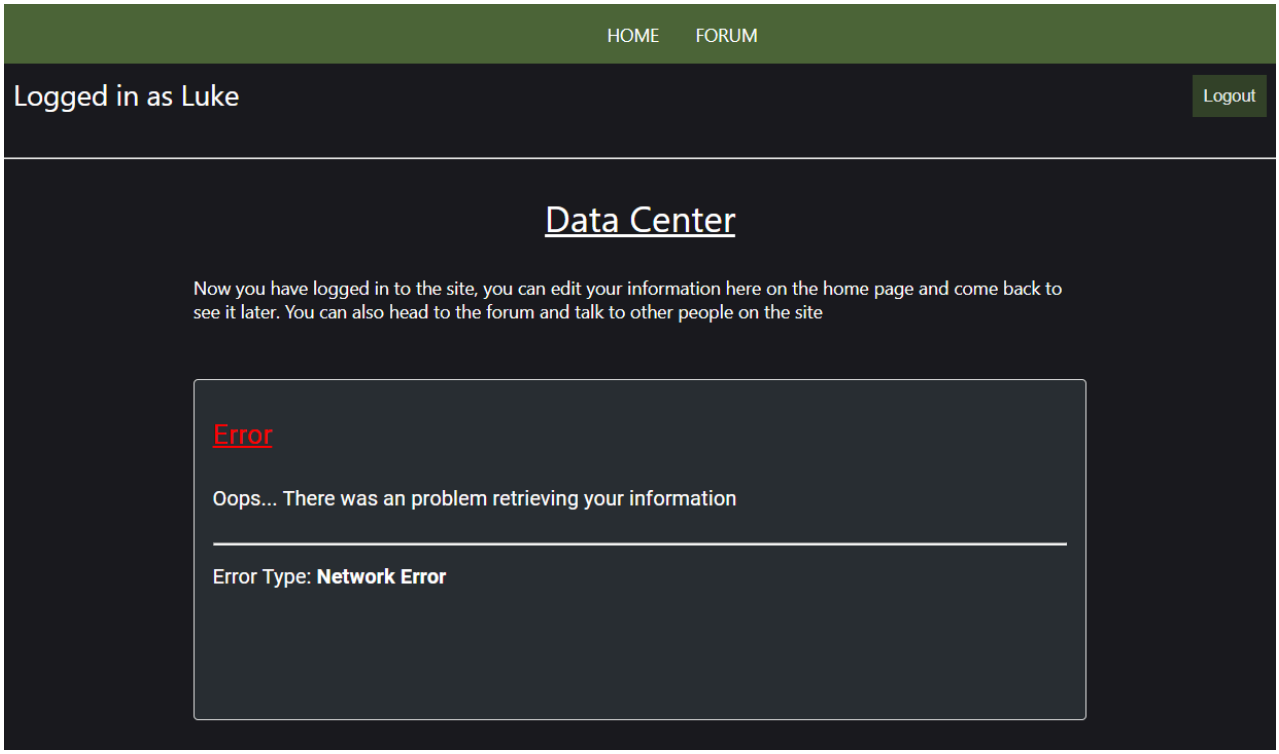


Figure 6: User interface for profile information showing conditional render of API3 request response error

API 1

API1 handles registration and logging in of users. One post endpoint in this API handles creating a new user shown in figure 7. The mongo database is searched to ensure no user with the username submitted in the request body already exists (at which point an error response is sent back). If there is no problem the password for the user is hashed and stored in the database along with username. API1 also contains endpoints handling logging in and out, this involves using web tokens and cookies to maintain information of whether the user is logged in at the front end.

```
//Registers a new user
app.post('/register', (req, res) => {
  console.log('Registering!')

  //destructures request body
  const { userName, password } = req.body

  User.findOne({ userName }).then((userInfo) => {
    console.log(userInfo)
    if (userInfo == null) {
      //if no user yet hashes password and stores in database
      const hashedPassword = bcrypt.hashSync(password, 10)
      const user = new User({ password: hashedPassword, userName, location: "", About: "", Tag1: "", Tag2: ""
      user.save().then((userInfo) => {
        console.log(userInfo)
        //sign webtoken with user info to send to client
        jwt.sign(
          { id: userInfo._id, userName: userInfo.userName },
          secret,
          (err, token) => {
            if (err) {
              console.log(err)
              res.sendStatus(500)
            } else {
              res
                .cookie('token', token)
                .json({ id: userInfo._id, userName: userInfo.userName })
            }
          }
        )
      })
    } else {
      console.log('user already exists!')
      //notify client user exists already
      res.send('userNameExists')
    }
  })
```

Figure 7: POST API1 endpoint for registering a new user

API 2

API2 handles requests regarding the forum component, such as posting new messages, returning all messages and deleting all messages. It also has an interaction with API3 where it retrieves profile information so when posting a new message, information about users location is sent along with it shown in figure 8 on line 36 where an axios request is sent. The forum is automatically updated at the front end with periodic checks on the database in case of new messages.

```
//Adds new post to database
app.post('/Posts', (req, res) => {
  console.log('making post reload?')
  const { userName, date, content } = req.body

  axios
    .get(
      'http://172.16.1.6:49153/ProfileInfo',
      { params: { username: userName } },
      { withCredentials: true }
    )
    .then((response) => {
      var location = response.data.location

      const post = new Post({
        userName: userName,
        date: date,
        content: content,
        location: location
      })
      post.save().then((postInfo) => {
        console.log(postInfo)
        res.send('made a post reload?')
      })
    })
})
```

Figure 8: Post endpoint for adding a new message to the mongo database

API 3

API3 is responsible for handling profile data (not username and password which is handled by API1). It has endpoints for creating a new profile shown in figure 9, retrieving a profile and updating fields in an existing profile.

```
app.post('/NewProfile', (req, res) => {
  const { userName, password } = req.body
  console.log('creating profile information')
  console.log(userName)
  const profileInfo = new ProfileInfo({
    userName: userName,
    location: 'placeholder',
    About: 'placeholder',
    Tag1: 'placeholder',
    Tag2: 'placeholder',
    Tag3: 'placeholder',
    Tag4: 'placeholder'
  })
  profileInfo.save().then((profileInformation) => {
    console.log(profileInformation)
    console.log('saved profile information')
  })

  res.send('Hi from server 3')
})
```

Figure 9: Post endpoint in api 3 for creating a new profile on registration

Structure

These APIs make use of mongoDB as a database which is non relational. There are a few schemas for the system. I show these in figure 10.

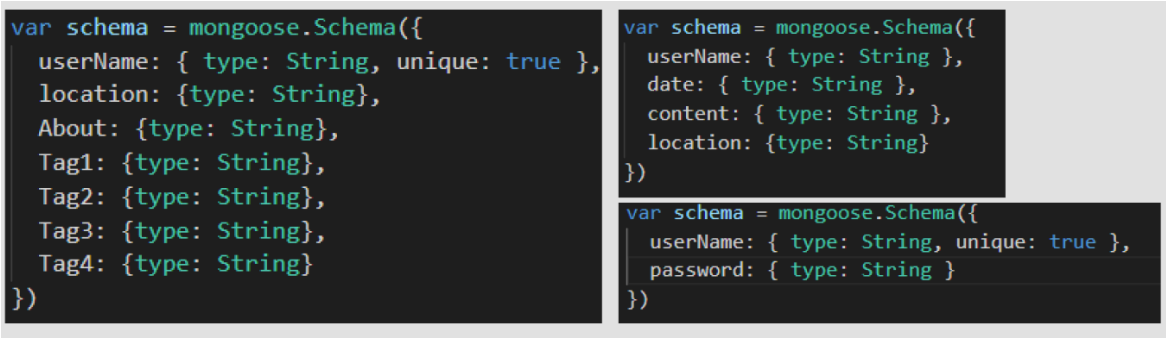


Figure 10: schema for a users profile information (left), schema for a message on the forum (top right), schema for a users username and password (bottom right)

The APIs run as services on docker. During development the system is run using 'docker-compose up' and nodemon is used to enable hot reloading of code changes, allowing more efficient development. The different services communicated over a bridge network. I have provided figure 11 to show the architecture of my system.

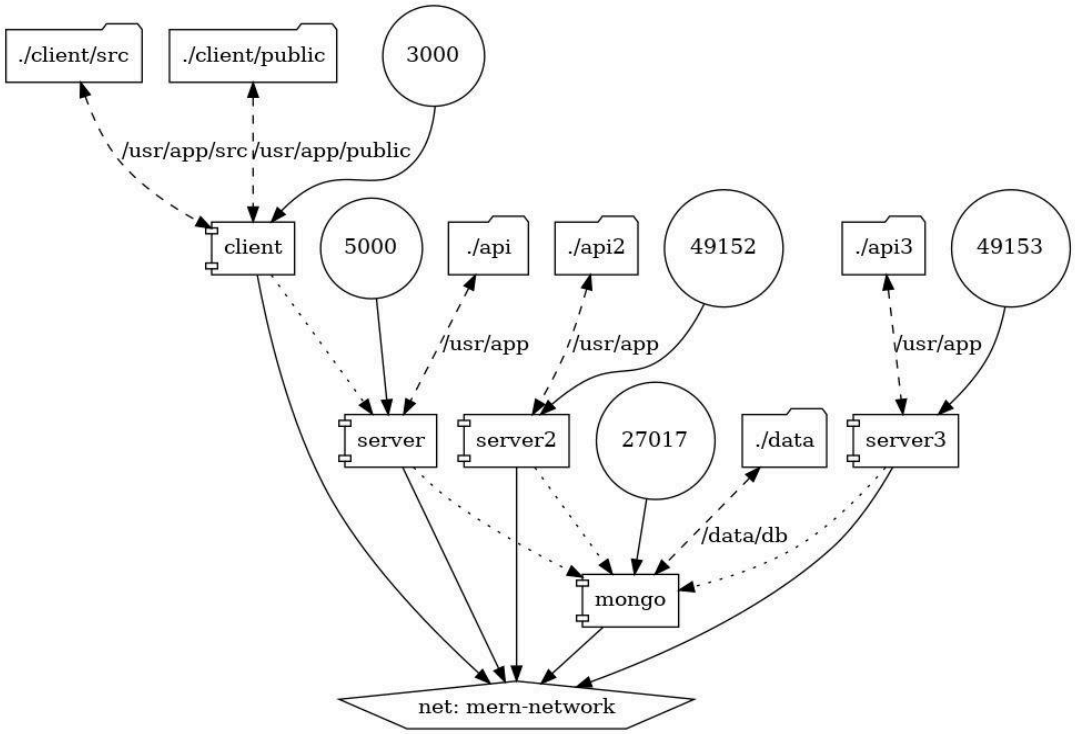


Figure 11: architecture diagram of my system running with docker

In figure 11 one can see the 5 different services (server (API) 1,2,3, client and database) are all running on ports on the local host and communicating over a central bridge network named ‘mern network’ named after the web stack I used. The servers (APIs) all have dependencies on the mongo service demonstrated by dotted arrows so errors don’t occur when servers startup and try to connect. The data in the mongo container is mapped into the ‘./data’ folder on the local drive so data remains consistent on restarting.

Deploying in swarm mode is slightly different and there are more services running regarding load testing and monitoring for running experiments. As seen in figure 12 I ran a visualiser container as well as 3 locust containers running on different ports each using the same root file where the locust.py files are stored (defining the load tests and API calls). The two variations of deployment are kept in separate ‘yml’ files so that I can differentiate between development and deployment for testing.

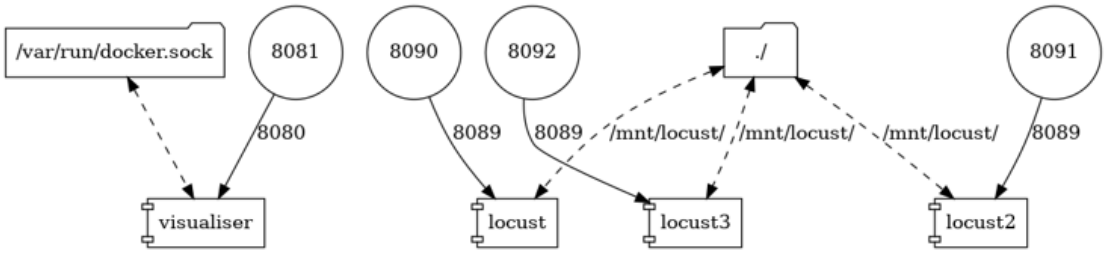


Figure 12: diagram of the visualiser and locust containers

Locust

Once my system was built I started injecting load. I hypothesized this would give me experience with my tools and give insight into more tests I could do later on. The first component I tested was the forum, specifically API2.

Locust Test 1

The first test I ran involved load testing the GET and POST endpoints for API2 which handle forum posts.

I ran one Locust container for GET requests and one for POST to monitor response times for each. When running both load injections concurrently as shown in figure 13 response times go up rapidly until the application is not usable, this made it apparent that the current design of my system is not adapted to high user load. This is probably because the client requests all posts in the database to map through and renders them on one page, so as the test runs the response size starts becoming very large. Oncerequest times become very high, error rate increases substantially shown in figure 14. It may have been sensible to implement timeouts in the code as there becomes a large backlog of work once response times get high as shown in the figure 13.

On reflection I should have usd pagination to limit the size of the response and the amount of data to be extracted from the database, so less posts are requested at a time.

As seen in figure 15, when I run the same experiment whilst running DELETE requests simultaneously (clearing the database of forum posts), response times remain steady for GET requests (also the error rate remains 0 and the POST requests in the same container are also less affected).

Figure 13: Get requests for forum posts response times (top) running at the same time as Post requests for forum posts response times (bottom) (test code shown on right).

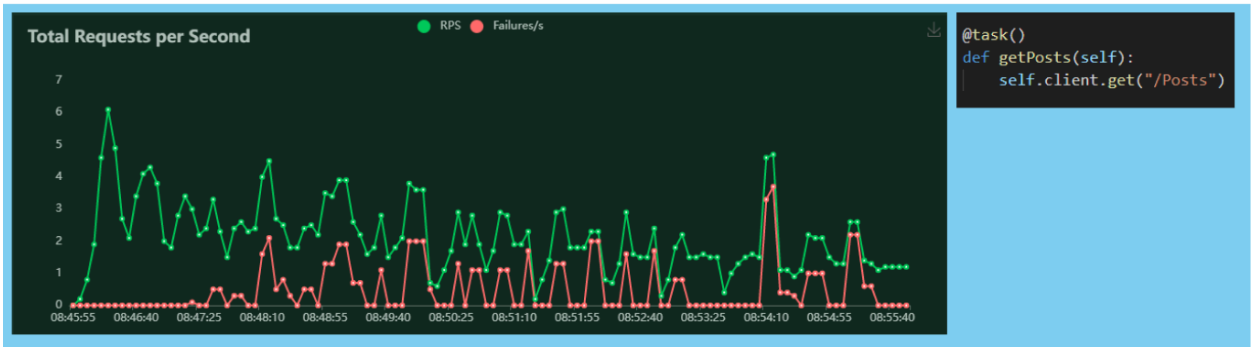


Figure 14: Error rate (red) and requests per second (green) regarding the GET requests load test from figure 11 (test code shown on right).

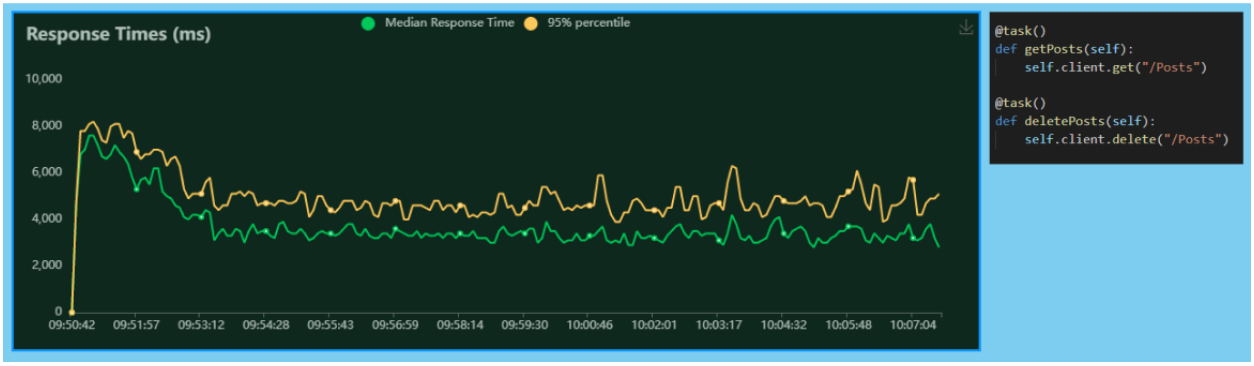


Figure 15: *Get requests running with Delete requests whilst Post requests are running response times the same as in figure 11 (test code shown on right)*

Locust Test 2

For this test I investigated the effects of running API2 concurrently with API3 with their endpoints being loaded. I started loading the POST API2 endpoint (for posting messages to forum) and monitored response times, and then started loading an API3 endpoint after about 2 minutes. As seen the ramping up of API3 requests immediately starts ramping up the response times of the API2 endpoint shown in figure 16.

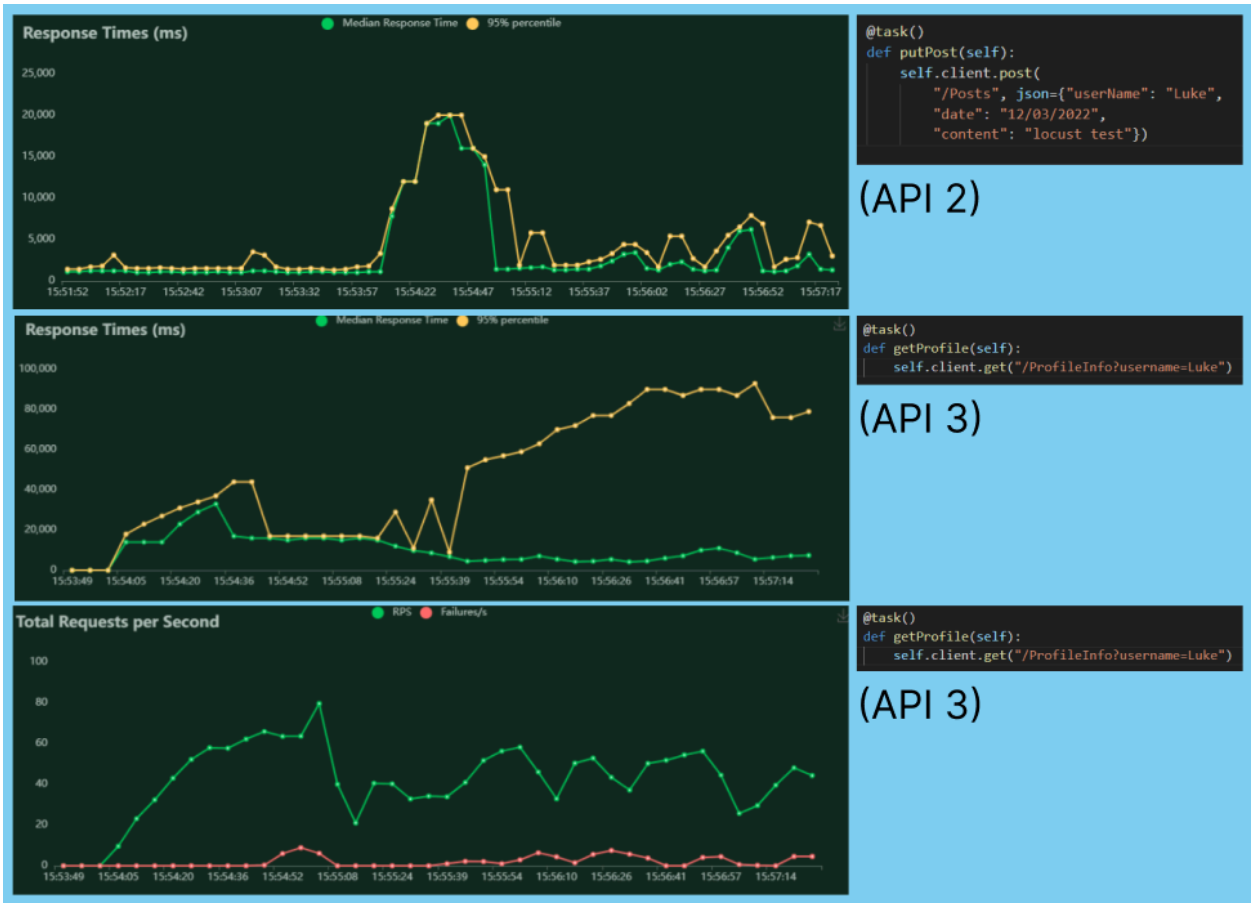
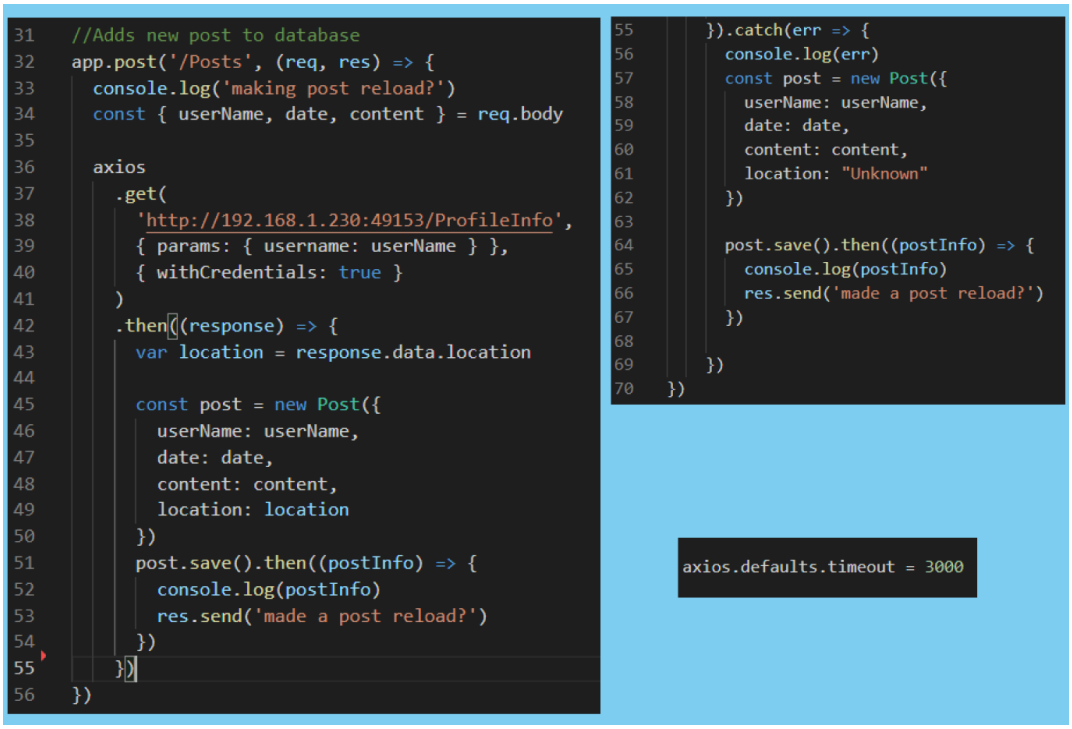


Figure 16: POST requests for API2 response times (top) with response times and requests per second for API3 (mid and bottom)

The main reason I explored the interaction between these two APIs is due to a reliance the POST request for adding new forum posts in API2 has on an API3 endpoint. As seen in the API2 code on line 38 (figure 17) when making a POST request to API2, it sends a request to API3 to get the profile information so that the users location can be added to the post (figure 4), meaning stress on API3 will actually increase the response time in API2. Running this test confirms there is actually quite a large impact on API2 when increasing load on API3. To remedy this I could program a timeout so if API3 doesn't respond after a time limit API2 proceeds to a default option.



```
31  //Adds new post to database
32  app.post('/Posts', (req, res) => {
33    console.log('making post reload?')
34    const { userName, date, content } = req.body
35
36    axios
37      .get(
38        'http://192.168.1.230:49153/ProfileInfo',
39        { params: { username: userName } },
40        { withCredentials: true }
41      )
42      .then((response) => {
43        var location = response.data.location
44
45        const post = new Post({
46          userName: userName,
47          date: date,
48          content: content,
49          location: location
50        })
51        post.save().then((postInfo) => {
52          console.log(postInfo)
53          res.send('made a post reload?')
54        })
55      })
56  })
```

```
55      }).catch(err => {
56        console.log(err)
57        const post = new Post({
58          userName: userName,
59          date: date,
60          content: content,
61          location: "Unknown"
62        })
63
64        post.save().then((postInfo) => {
65          console.log(postInfo)
66          res.send('made a post reload?')
67        })
68
69      })
70  })
```

```
axios.defaults.timeout = 3000
```

Figure 17: code in API2 endpoint for adding post to database (left) and catch statement addition with timeout specifier added (right)

Timeout Implementation

Although these are not regarded as chaos engineering experiments, running load tests using Locust can be a quick way of uncovering vulnerabilities and interactions. Despite interactions demonstrated being quite trivial, it showcases them without diving into chaos experiments which take longer and require more thought and preparation. They could provide insight into further testing to investigate.

Before continuing I implemented a fix regarding API3 interfering with API2, as a main benefit of running individual microservices is that they don't interfere with each other too much so I wanted to emulate this in my system. This allows me to run chaos experiments with a steady state hypothesis such as 'upon failure of any one service running the system, other services continue to run at a steady state without being affected'. As seen on line 55 in figure 17 (right) I implement a catch statement so that upon a timeout error instead of the location of the post being retrieved from API3 it is simply stored as 'Unknown'. I have set the default axios timeout to 3000ms meaning that if the request to API3 takes a long time due to network traffic, or high CPU load (which I will likely simulate in my experiments), API2 will default to this catch statement and continue functioning relatively normally.

Monitoring

Cadvisor

One of the first steps when conducting chaos experiments is defining a steady state hypothesis. Whilst in production this is often done by monitoring something like ‘SPS’ (streams per second) at *Netflix* or response rates [2], as well as CPU usage, network usage, error rates, container health and other metrics. To define a steady state I set up a monitoring system to extract some metrics from containers running my system. I spent time setting up a Cadvisor container using prometheus to scrape metrics into a redis database whilst using the prometheus user interface exposed on a local port number. I accomplished this by adding the required images to my swarm.yaml file, adding the appropriate container dependabilities and volumes so that they ran automatically when I launched the rest of my containers with the same command.

Figure 18: Cadvisor metrics with CPU, Network and Memory for API2 container with load injection running from locust (left). Locust file demonstrating requests that were running simultaneously to API2.

To examine the usefulness of Cadvisor I ran a Locust test, sending requests to API2 endpoints. You can see some metrics shown in Cadvisor for the API2 container in figure 18. I already knew Cadvisor only gives current metrics not historical, somewhat limiting its usefulness, however I

found Cadvisor useful for gauging what kind of memory and CPU constraints to put on my containers. As seen in figure 18, memory usage as a percentage of total allocated to the container can be seen. It is also useful for confirming my monitoring systems are picking up the changes in load on containers, for example when I started the Locust test I could see the network throughput increasing regarding bytes transmitted and received as well as memory usage and CPU. There are also metrics for network errors which can be useful to catch anything initially wrong with my setup.

I discovered I could more accurately simulate users by introducing wait times between user requests to my API endpoints using wait times (right of figure 18 at the bottom), as opposed to each user sending new requests as soon as the previous finishes. This randomly selects a value between 2 and 5 seconds and waits that time before that user sends another request. I found this made my APIs much more stable and ran into far less crashing, it is also unrealistic that each user would be sending multiple requests per second for my particular application, it is more sensible to increase the amount of users and keep this random wait time if I want to increase the load on my system using Locust.

Prometheus

It became apparent Cadvisor wouldn't be enough to gather all data I needed so guided by my background research I set up Prometheus to scrape data from cadvisor into a redis database and exposed it on a port. I created a prometheus.yml to configure settings and set the scrape interval to 5 seconds. I set the target to cadvisor running on port 8080.

Using prometheus proved more complicated than I thought, there are a multitude of metrics one can query which you must execute as expressions in the prometheus interface. It was not immediately apparent which of these would be useful for my investigation. As the main areas I will concentrate on are CPU, Memory and Network I deduced a few queries that might be useful. Prometheus includes metric types which should be understood before collecting data. These include Counter, Gauge, Histogram and Summary.

Counters are cumulative metrics, a counter value can only increase or be reset to zero, examples include number of requests served, tasks completed or error counters. A Gauge on the other hand represents a single numerical value that can arbitrarily go up and down and can be used for measured values, like current cpu or memory usage [15]. I will not be using histograms or summaries so I won't go into detail about them.

Prometheus provides a variety of functions to be used on metrics. One of the more useful ones I found was 'rate()'. This function calculates the per second average rate of increase of the time series in the range vector and should only be used on counter metrics. I could for example use the rate function on the 'container_network_receive_errors_total' metric to give me an idea of the rate of network errors occurring for a container, encouraging me to investigate further.

Regarding cpu monitoring a few metrics stood out. ‘container_cpu_usage_seconds_total’ outputs cumulative cpu time consumed per core in seconds and would be useful to monitor the general cpu usage of a specific container. This does however give the outputs of all cores therefore I should aggregate this. This metric is a counter type which means it would not necessarily give me a good indication of a container's cpu use at any one time. I can use some query language to make this more useful. As seen in figure 19 (top) this metric queried alone shows the total cpu usage which is increasing over time.

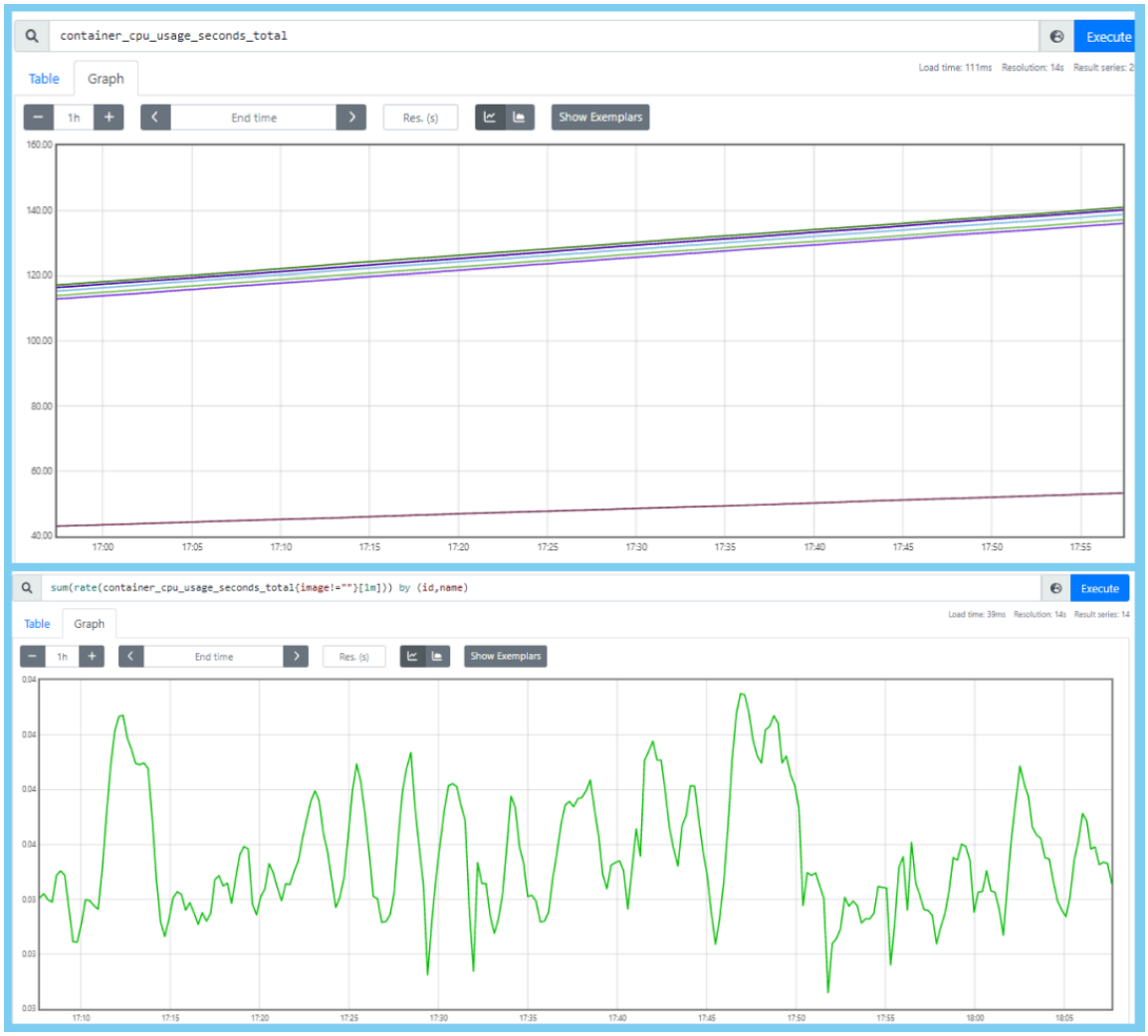


Figure 19: Prometheus container cpu usage in seconds total for API2 container showing all cores working, cadvisor cpu usage shown in crimson at the bottom (top) sum of all cores shown as rate for API2 container (bottom).

By aggregating total cpu usage in seconds of all cores for a particular container (done by summing with id and name) after using the rate function shown in figure 19 (bottom) I output a graph resembling the containers CPU usage at any one time (the prometheus user interface allows one to select certain containers selected by their name and id to be shown on the graph).

I have ensured only series with image names (my containers) are fetched to avoid confusion shown by the expression syntax in figure 20.

```
sum(rate(container_cpu_usage_seconds_total{image!=""}[1m])) by (id,name)
```

Figure 20: Prometheus expression for displaying cpu usage of a container

By enabling the display of all my containers on this graph I give a nice overview of the cpu usage of my containers to spot when one is misbehaving due to something like CPU exhaustion shown in figure 21.

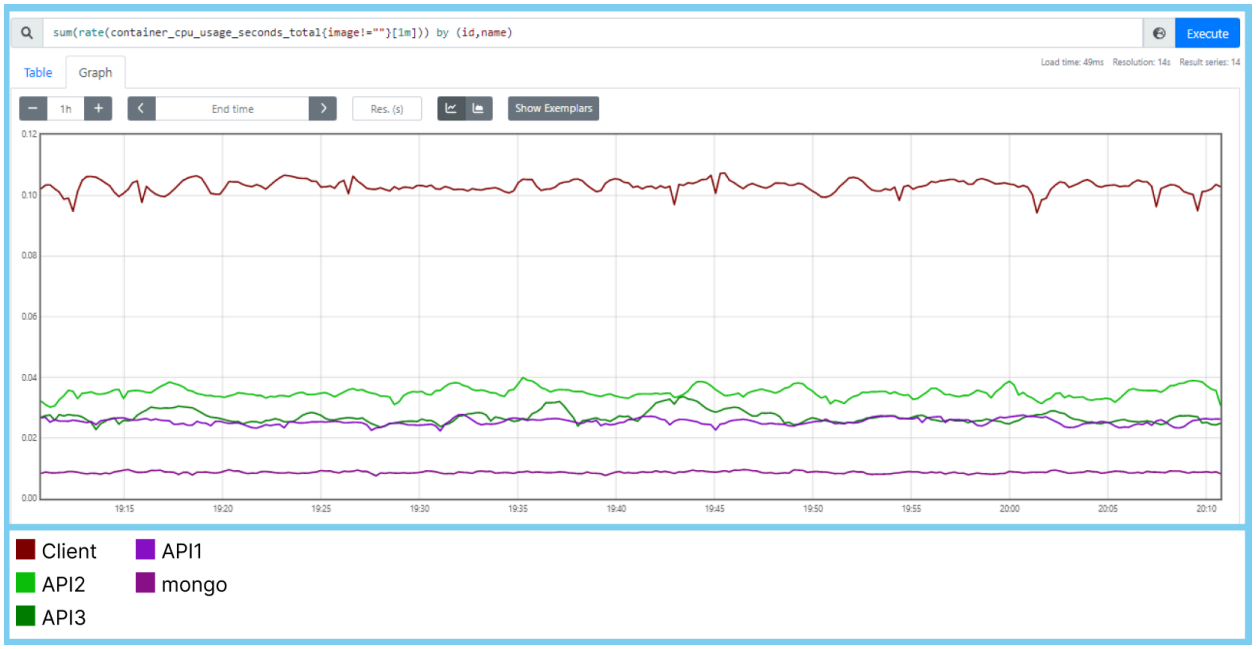


Figure 21: prometheus graph displaying cpu usage for relevant system services

I set up monitoring for network activity of my containers. For this I did something similar. I found a useful counter metric called ‘container_network_receive_bytes_total’ and manipulated it to show rate, shown in figure 22. I also sum by name because the metric differentiates between different network interfaces which is not something I need. I do a very similar thing to show network transmitted bytes but I won’t show this here. These monitoring methods would be useful when injecting network traffic, packet loss and black holes as I can ensure that failure injection is working as intended. The graph in figure 22 shows metrics during the same locust load tests as figure 16. This is not important for now but explains why API2 is shown to be receiving more network traffic than others. It shows API3 which is used by API2 in certain requests being relatively active which is expected. No requests are being sent to the client or to API1 which is why they are not receiving any traffic. The mongo container is however receiving traffic as requests are being sent through mongoose by API2 and API3 to store and read their respective data.

Figure 22: prometheus graph displaying network input for relevant system services

I would like to monitor memory usage. I found a decent expression for showing this that will be demonstrated later as the current graph does not show much as memory usage seems to be quite low.

Grafana

As discussed in the introduction I wanted to look into Grafana. Grafana essentially scrapes information straight from Prometheus using an exporter, allowing more complex and customizable data visualization. To set this up I pulled the Grafana image and added it to my docker compose file, I also set up a few Grafana configuration files so Grafana targeted prometheus as its data source, then exposed the service on port 3000 to access the user interface.

Grafana lets users customize dashboards which can be saved, these can display various metrics you're interested in all at once so you can identify issues and anomalous data quickly. The expressions and queries Grafana uses are essentially the same as prometheus so I reuse most of the expressions I created previously. I found Grafana has a large library of premade dashboards and panels to use, some of them designed for docker. I found a dashboard that seemed useful for monitoring my system, then with customization saved panels that display the most important metrics I have explored with prometheus, I have shown this in figure 23.

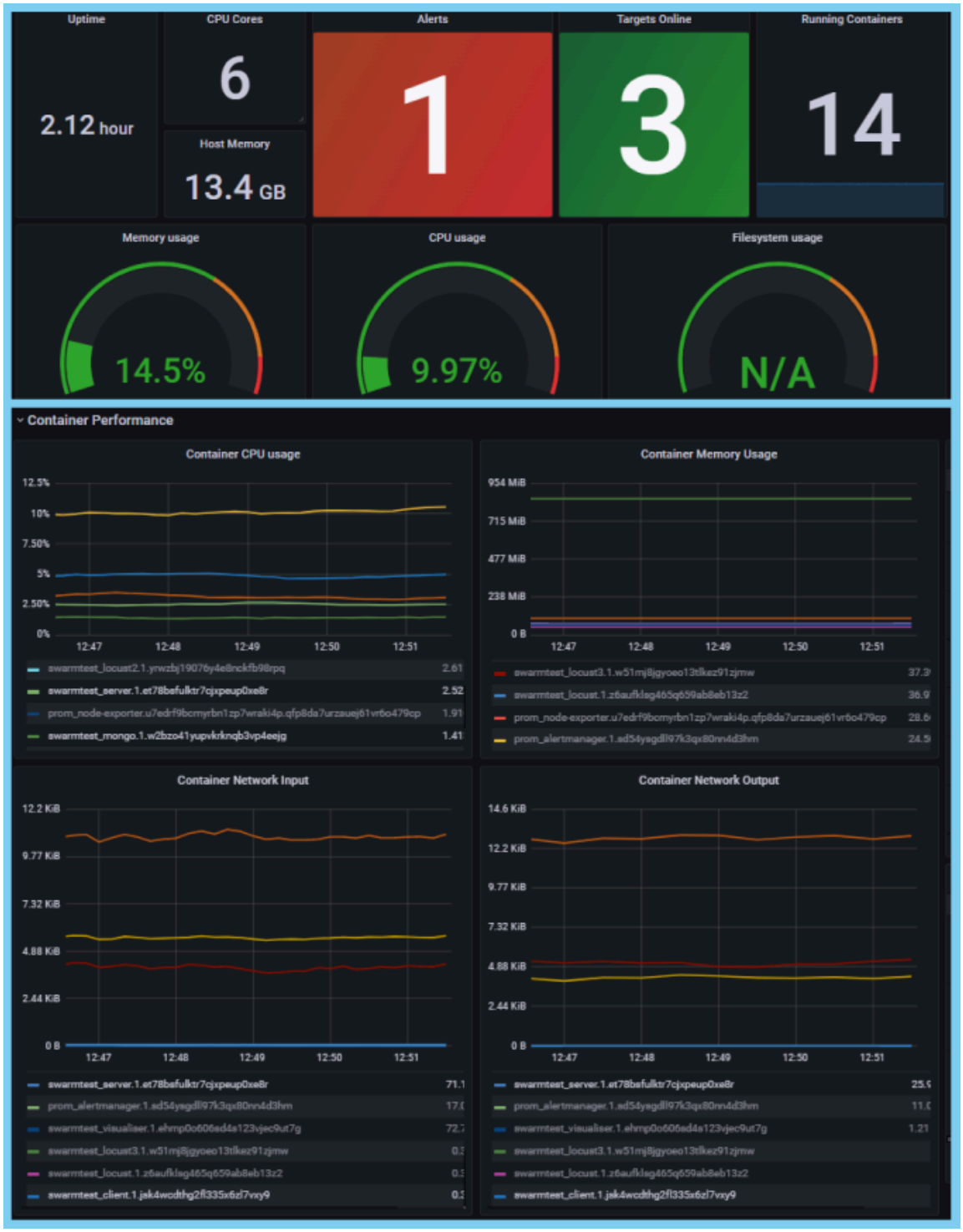


Figure 23: Custom grafana dashboard displaying important metrics for monitoring my systems behavior.

As I looked into chaos experiments to run I realized although it was useful having metrics on cpu and memory it would also be important to measure more directly impact on users as this is more important from a business perspective. Even the network transmit and receive metrics don't give as much detail about the API behavior as I'd like. It is possible to use Locust graphs from the UI to monitor response times and failure rates, however I found it quite messy having all these windows open simultaneously. There is also less customisation in the Locust UI so I decided to look into importing metrics from Locust into Prometheus and Grafana.

I found a project called Locust metrics exporter [17] which exports Locust container metrics into prometheus to be further manipulated. I pulled the image for the Locust metrics exporter and set appropriate environment variables to target my locust container running on port 8089. I added another scrape target in the prometheus.yml file to make it available as a target for cadvisor shown in figure 24.

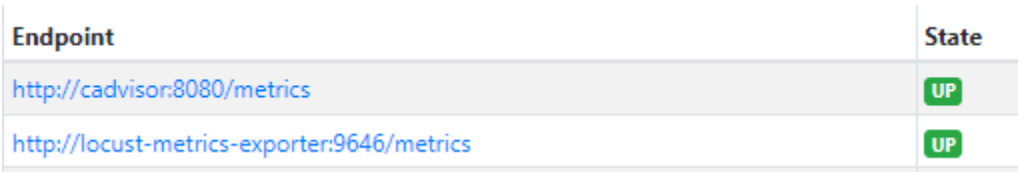


Figure 24: Prometheus targets displayed in UI

This allowed me to make queries in Prometheus and Grafana to form graphs showing Locust information about response times, request rates and error rates of endpoints in my different APIs as well as show these on my Grafana dashboard by adding new panels.

I found that rather than running multiple Locust containers at once for different APIs, I could simply have one locust container running and define different kinds of users in my Locust file. I streamlined my Locust environment to one file and container which made setup quicker everytime I ran tests. I show the code in my Locust file in figure 25. I also define the endpoint for each class so this does not have to be set up manually in the Locust interface. As seen in figure 25 when I run this in Locust requests are being run concurrently to their respective endpoints.

(API 1)

```
class api1(HttpUser):

    host = 'http://192.168.1.230:5000'
    @task()
    def registerUser(self):
        self.client.post("/register", json={"Luke", "test"})


    @task()
    def getPosts(self):
        self.client.get("/")

    wait_time = between(2,5)
```

(API 2)

```
class api2(HttpUser):

    host = 'http://192.168.1.230:49152'
    @task()
    def putPost(self):
        self.client.post(
            "/Posts", json={"userName": "Luke",
            "date": "12/03/2022",
            "content": "locust test"})

    @task()
    def getPosts(self):
        self.client.get("/Posts")

    @task()
    def deletePosts(self):
        self.client.delete("/Posts")

    wait_time = between(2,5)
```

(API 3)

```
class api3(HttpUser):

    host = 'http://192.168.1.230:49153'
    @task()
    def getProfile(self):
        self.client.get("/ProfileInfo?username=Luke")


    wait_time = between(2,5)
```

Type	Name	# Requests	# Fails
GET	/	4117	35
DELETE	/Posts	2755	31
GET	/Posts	2755	18
POST	/Posts	2716	17
GET	/ProfileInfo?username=Luke	7030	8
	Aggregated	19373	109

Figure 25: Locust file classes for each API in the same file, Locust UI whilst Locust test is running (bottom right)

I set up some useful queries in Grafana to monitor my APIs. I decided on three core metrics to consider; request rates, response times and error rates.

The Locust API has a metric for number of requests for each endpoint so I turned this into an appropriate rate and summed them by name and method (POST, DELETE, GET). I show this metric in Grafana in figure 26 whilst load is injected by Locust configured as in figure 25 for a 30 minute timeframe. This will likely remain constant as when I introduce latency, cpu load as well as shutting down services the Locust requests will still be made as the Locust container isn't affected. However, it is a nice metric to have as it confirms on the dashboard that requests are being made and Locust is working correctly.

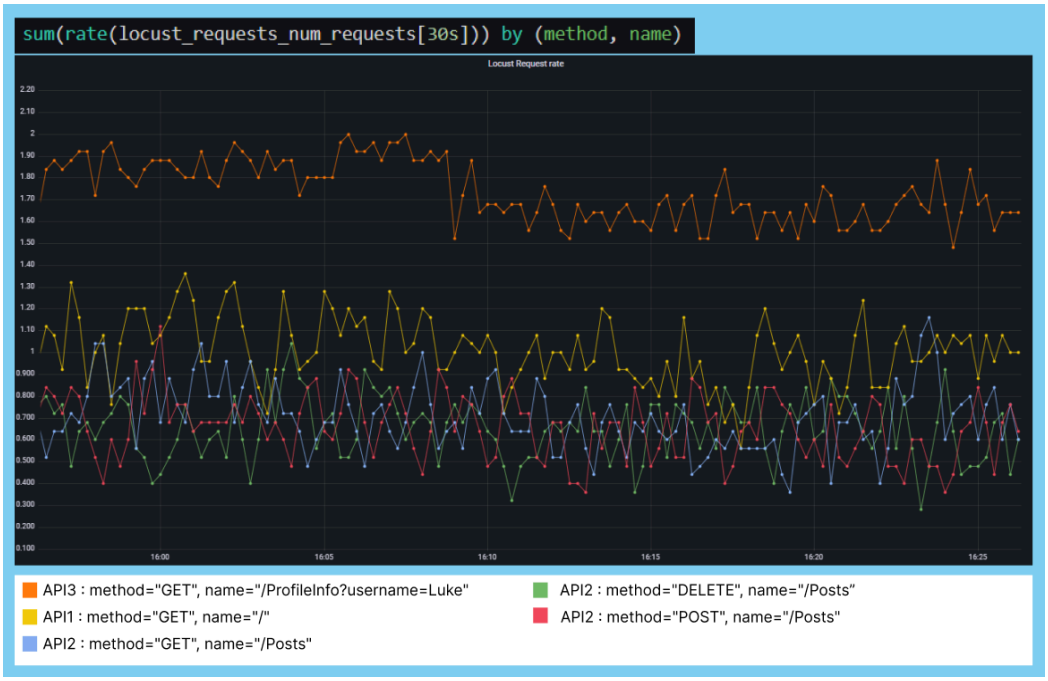


Figure 26: Locust request rate queried and displayed in Grafana

The average response times metric will be useful for monitoring how users would be affected, as I anticipate when I run attacks increasing latency or cpu usage response times will increase, meaning users have to wait longer to access features of the system. I have come up with an effective query to get this information as a rate shown in figure 27.

Prometheus and Locust don't always provide axis labels and units which is why some graphs provided are not labeled on the axis, especially when query functions are used it can become difficult to accurately define units. In this case this is not so important as I'm mostly comparing rates on the same graph and significant increases and decreases so the units remaining somewhat arbitrary is not a significant problem. I have done my best to show units in the figure labels.

To gauge what response times are appropriate for user experience I did some research. I found there is generally accepted advice regarding response times which has been around for about 30 years stemming from the 'Nielsen Norman Group' [16] determined by human perceptual abilities. 0.1 seconds is roughly the limit for a user feeling the system is reacting instantaneously, 1.0 second is about the limit for the user's flow of thought to stay interrupted even though the user will notice the delay, and is about the limit where no special feedback should be necessary. This 1.0 second limit is ideally what I would like the average API response time to stay underneath, especially since there will be a small delay whilst the UI updates. As shown in figure 27 all response times are well under 1 second with most endpoints serving requests sub 20ms. The response times for API1 seem to be slightly more volatile for unknown

reasons however they are still acceptable according to the response time research presented from the source.

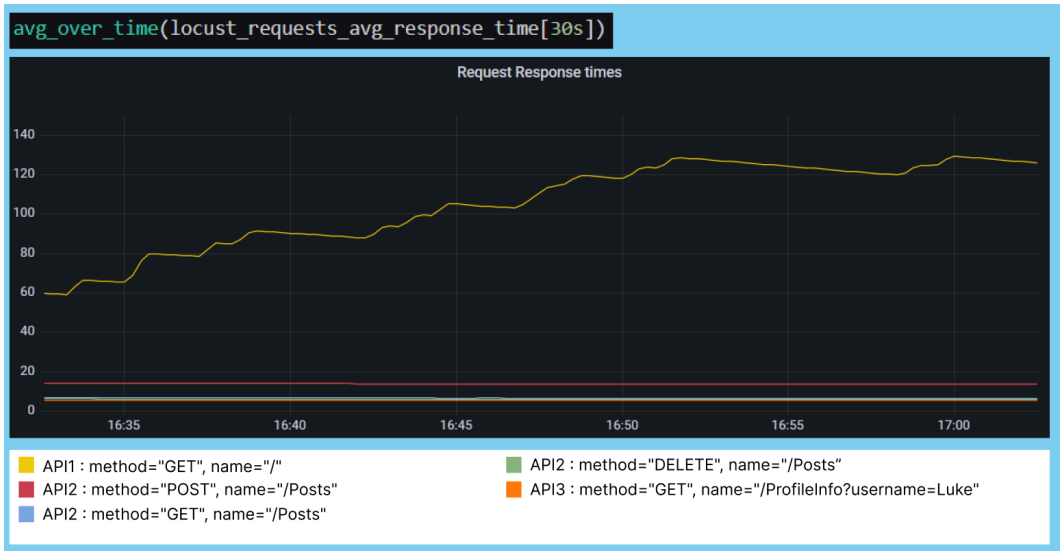


Figure 27: Locust request response times (ms) for each endpoint

I provide an error rate metric as a quick indicator that an endpoint has broken or is failing to serve large amounts of requests. Data in figure 28 shows very low error rates as there is no fault injected whilst producing these graphs. I anticipate once stress is injected onto the system error rates will increase quickly. It seems these sparse errors often come under the category of 'RemoteDisconnected('Remote end closed connection without response')'. According to python documentation this occurs when the attempt to read response results in no data read from the connection, indicating the remote end has closed the connection. I couldn't find any official documentation on why these errors occur but after reading discourse on 'Stack Overflow' it seems they are quite common through general packet loss and fluctuations in internet connectivity.

So whilst running chaos experiments I can quickly notice important metric changes; I have organized my panels on a dashboard which I show in figure 29.

Figure 28: Locust request current fail rate

Figure 29: Overview of panels on grafana dashboard including those imported from cadvisor and Locus

Chaos Engineering Tools

Gremlin

To carry out attacks I must establish a method and platform. I considered two options; Gremlin and Pumba. Pumba seems like a good option however I found more online support for Gremlin, as well as more configurable options and an optional user interface. I chose Gremlin to carry out chaos experiments.

Gremlin is a SaaS chaos engineering platform that can be installed and run in various ways. Gremlin allows injection of failure into services, hosts or specific containers using a library of attacks. One can run attacks in sequences (called scenarios) which the user can customize themselves or choose from a recommended selection.

The attack library consists of multiple modes, designed to replicate common failure conditions. This includes CPU; generating load across CPU cores to ensure a system can withstand stressful conditions caused by high demand or heavy traffic. Memory; consuming a specific amount of RAM in the container. I/O; which creates read/write pressure such as hard disks. Disk; consuming a specific amount of space on a storage device. Shutdown (with the option of automatically rebooting). Termination of processes. A variety of network attacks are also available such as Blackhole which drops all network traffic based on port, network interface etc, latency which simply injects delays into outbound network traffic which validates that a system can still function under slow network conditions, packet loss which drops or corrupts a percentage of outbound network traffic.

To set up this system on my machine, I ran the Gremlin agent within a docker container in order to run experiments on other containers. There is also functionality to run attacks on the host machine itself however I don't think this is necessary for my project. I won't go into how I set up the Gremlin agent in detail, however I ran into quite a few issues. It consisted of setting up environment variables and issuing a series of docker commands.

Once the container was set up I could either run attacks using the CLI in the docker container or from the user interface shown in figure 30. I decided to mostly use the latter. As seen in figure 30 once the Gremlin client is set up I can run attacks, the gremlin agent identifies other containers running on the host system allowing me to select them for attacks.

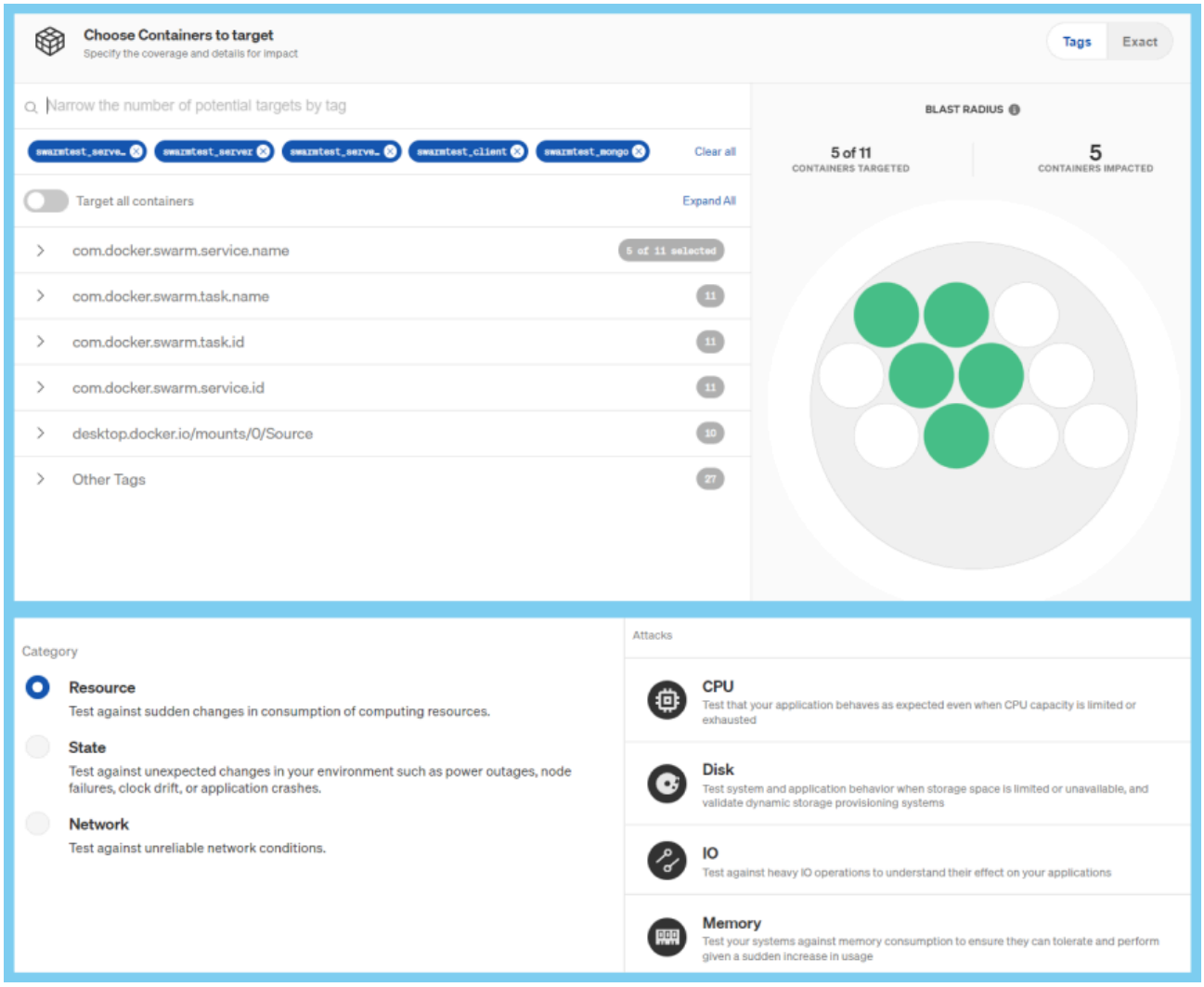


Figure 30: Gremlin web application user interface, container targeting and attack method UI displayed

Preliminary experiments

Before I went about doing the bulk of experiments I experimented with Gremlin and my monitoring system a little. The first attacks I looked into regarded CPU exhaustion. I attacked my containers occupying 95% of CPU capacity on all cores for 120 seconds each. I found throughout all initial runs the response times of API1 steadily climbed without stopping with no stress injection to the point where it dwarfed all other response times shown in figure 31. This was due to an issue with requests to the register endpoint so I altered Locust to not include it. I used this time to gauge what request levels would be appropriate for running tests. I initially spawned 50 users to send requests however I found after running my initial set and then testing with 300 my results looked more meaningful so I stuck with this. I realized that there were more metrics that could be useful. I could query information about CPU throttle to measure how much each container's CPU usage is being limited by c constraints.

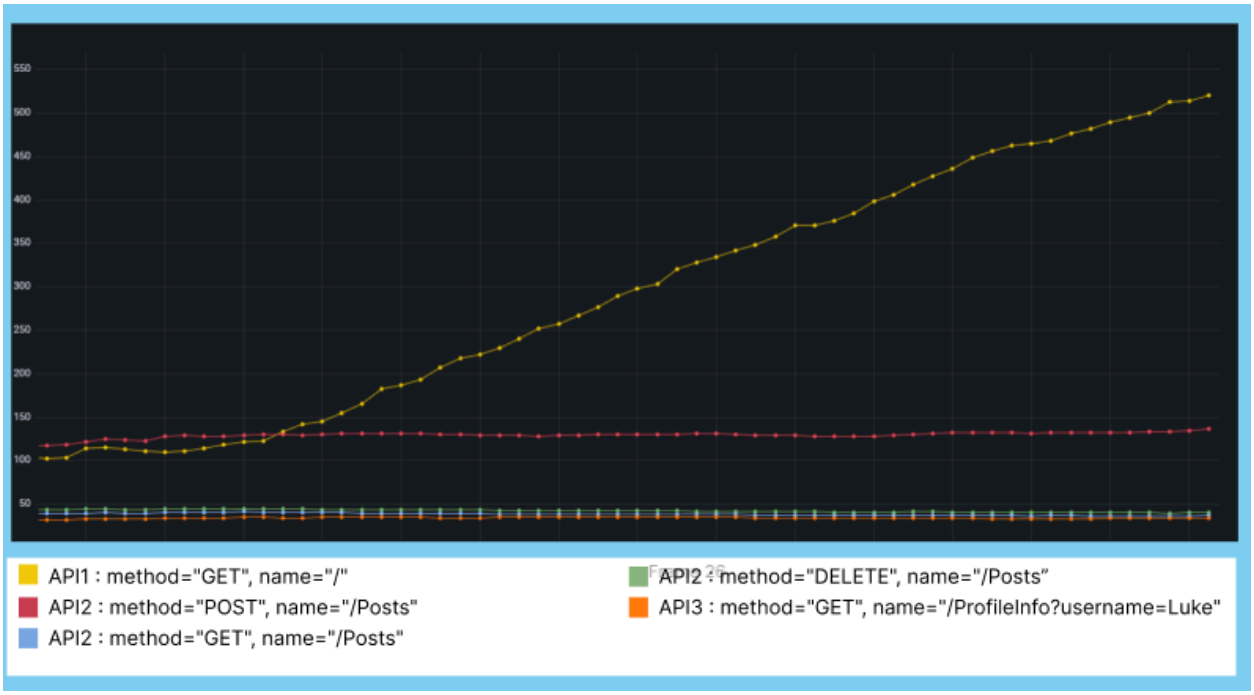


Figure 31: Endpoint average response times (ms), API1 GET endpoint misbehaving

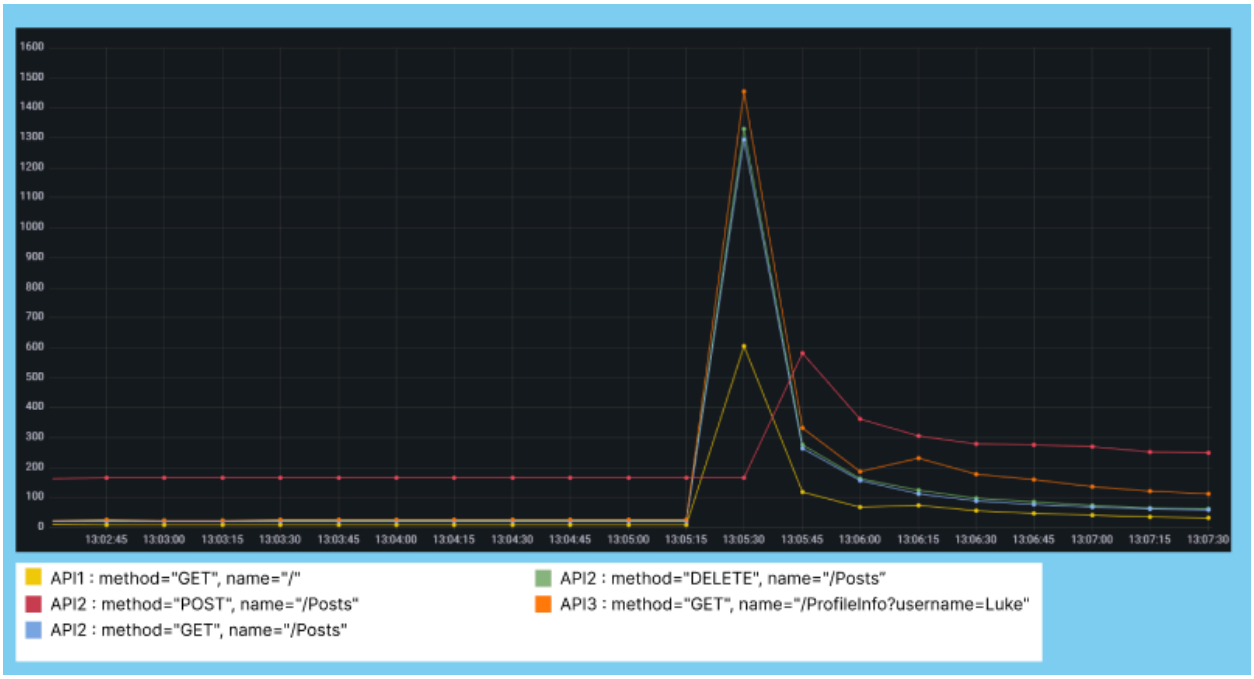


Figure 32: Endpoint average response times (ms), impact of starting new Locust test and spawning new users at 13:05:15

I realized my technique for collecting data about response times for endpoints was flawed. I queried from the Locust exporter a metric called ‘locust_requests_avg_response_time’ which is

not an indication of current average response time over a short time period but rather the average response time for the whole Locust test. As seen in figure 32 when starting a new Locust test the users first must be spawned and the response times for each endpoint is initially very high as it is an expensive task. This distorts the average for the rest of the experiment so I eventually found that by simply resetting the statistics for Locust test with the UI a short time after the test started I could cut down response times to a more realistic indication of what they actually are at any time. This made the effects of injecting stress much clearer in certain experiments.

In figures 33 and 34 you can see before and after implementing this. In two runs of the same experiment (injecting CPU overload into API2) the initial response times are considerably shorter after, showing a bigger increase in response times when fault is injected. It is quite obvious that response times are increasing beforehand however if fluctuations were more subtle useful information could be lost.

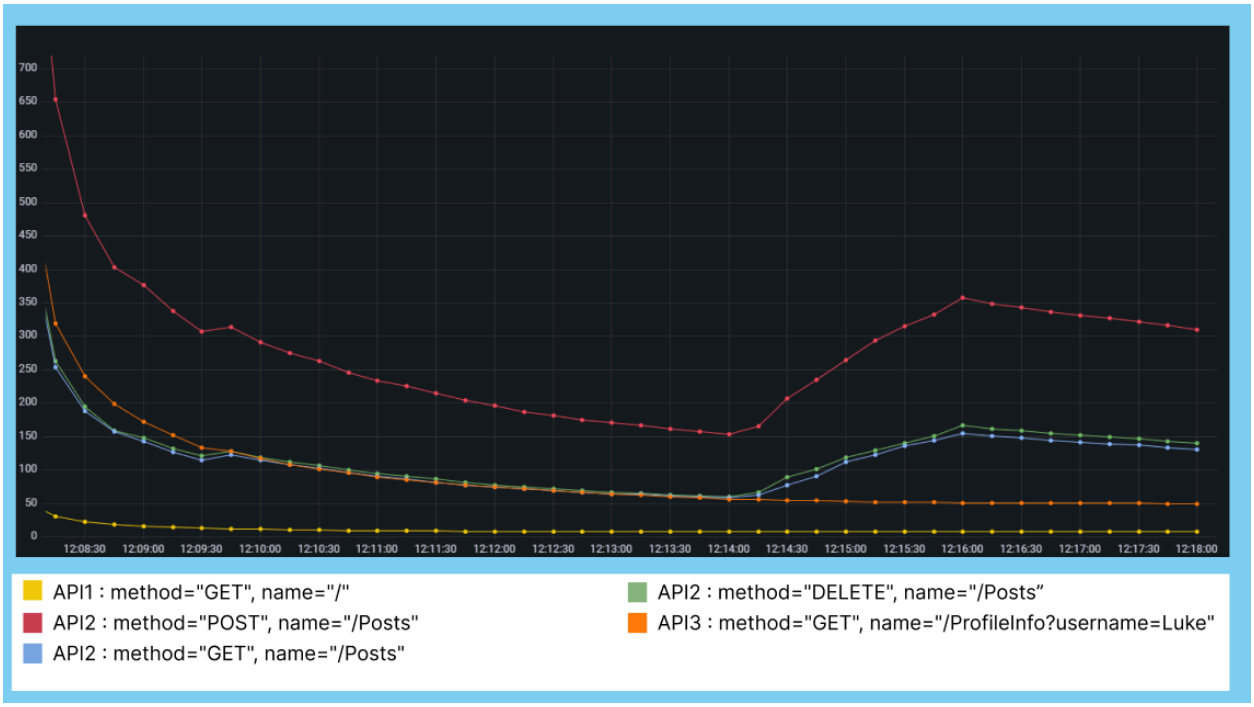


Figure 33: Response times of endpoints (ms) with API2 endpoints being loaded. Fault is injected around 12:14:00. (No Locust reset)

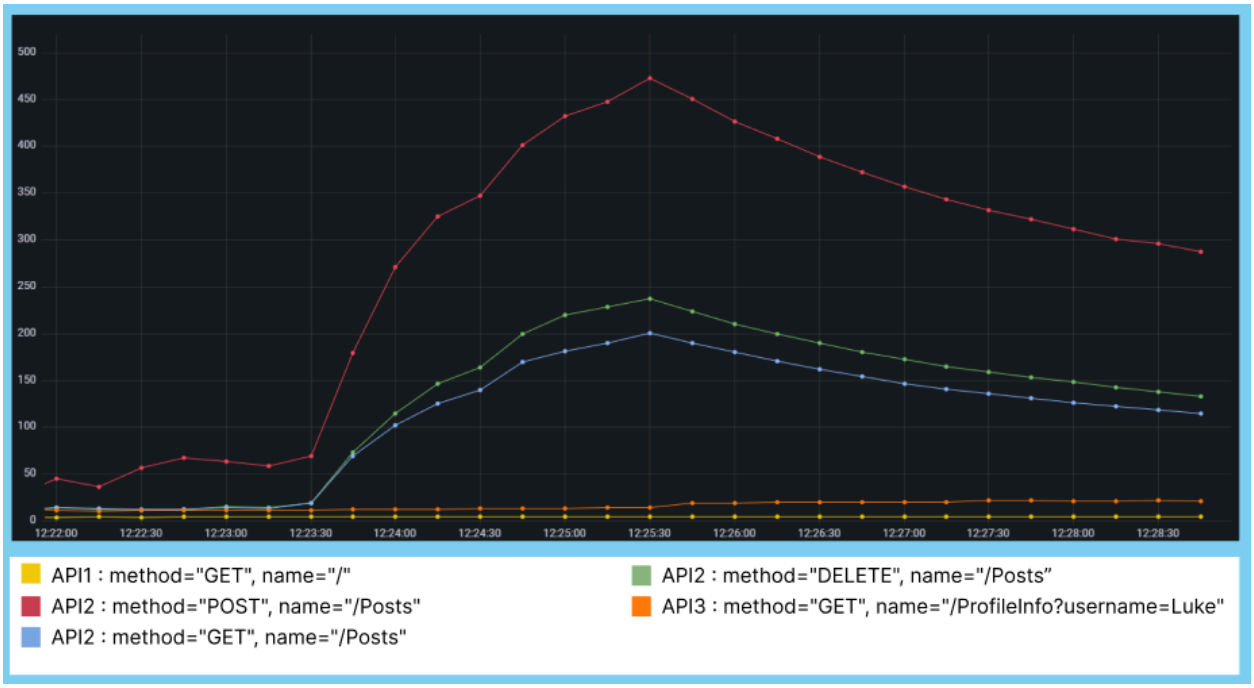


Figure 34: Response times of endpoints (ms) with API2 endpoints being loaded. Fault is injected around 12:23:30. (Locust reset)

I found whilst trying to get a better indication of current response times there was a Locust metric queried as 'locust_requests_current_response_time_percentile_95'. This metric displays at any one time the response rates of all endpoints in the test averaged but only for the 95th percentile. Rather than giving an average for the whole Locust test like 'locust_requests_avg_response_time' this gives a more current indication meaning changes in response times can be more stark. However since there is no option for getting this metric for each endpoint separately, if one endpoint had much higher response times than another during normal load without failure injection, using this metric alone might cause smaller changes in lower response rate endpoints such as with API1 to be overshadowed and missed. It is necessary to include the average response times for the separate endpoints as well.

Results and Evaluation

In this section I go through experiments I ran for different attack categories, regarding CPU and Memory attacks.

Regarding the steady state hypothesis, the approach I took with most experiments is to allow services to run whilst being injected with load for enough time to garner a steady rate of response times. This can be compared to response times once stress has been injected by Gremlin. For all experiments 300 users were spawned by Locust using configurations discussed earlier. Shown in figure 35 are requests per second for one experiment, however this is similar for all of them as the number of users spawned remains constant across experiments.

As seen in figure 35 the number of requests per second to each endpoint is not consistent. This is because load across each API is the same however API2 has three separate endpoints being tested therefore as seen the requests per second for each endpoint in API2 is roughly three times lower than that of API1 or API3 which both only have one endpoint being loaded.

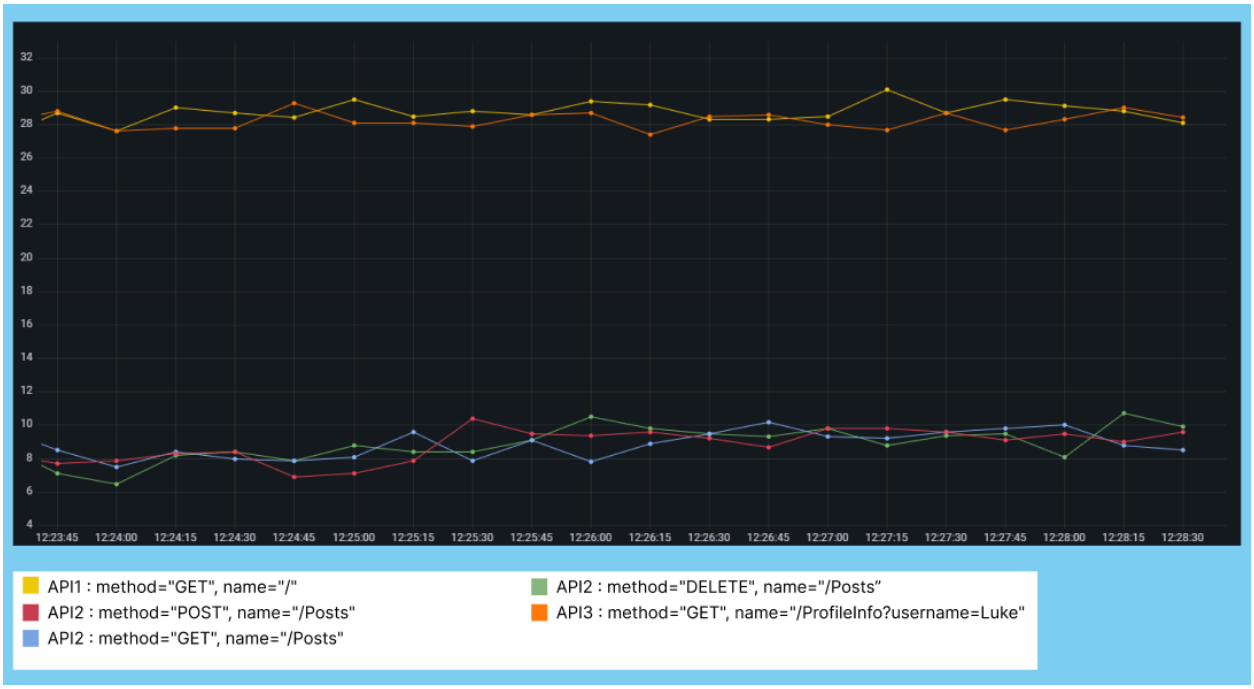


Figure 35: Requests per second shown for each endpoint with 300 users spawned

CPU Attacks

Experiment 1.1

Hypothesis:

- Upon overloading CPU usage of API1, response and error rates go up however other services are not adversely affected in terms of response time. After the attack, response times and error rates start returning to normal.

Attack Description:

- I exhausted the CPU capacity consuming 100% of usage on all cores regarding the API1 container. This attack was carried out for 120 seconds between **12:55pm** and **12:57pm.**

Results and Evaluation

CPU and Throttle

To ensure the attack was carried out correctly, I observed CPU usage and throttle for the container. As seen in figure 36 the CPU usage for API1 was throttled starting just after 12:55 when the attack began suggesting CPU usage began to exceed its constraint of 10%. This throttling quickly started returning to normal levels once the attack finished just after 12:57. Throttle levels are considerably higher for API2 and API3 suggesting CPU usage for normal functioning of these APIs under load is higher than API1. This makes sense as the API1 endpoint is very simply retrieving a short json response whilst the others are reading and writing to disk and performing more computationally expensive tasks. Regarding figure 37 showing CPU usage, this essentially mirrors the graph for throttle as I would expect. When the CPU attack is carried out it can be seen that CPU usage peaks at 10% and then remains consistent for the duration of the attack. This will be because I have issued a constraint on the API containers at this level shown in figure 38, therefore this seems to have worked as intended.

Figure 36: CPU Throttle for each docker service during Experiment 1.2 across time

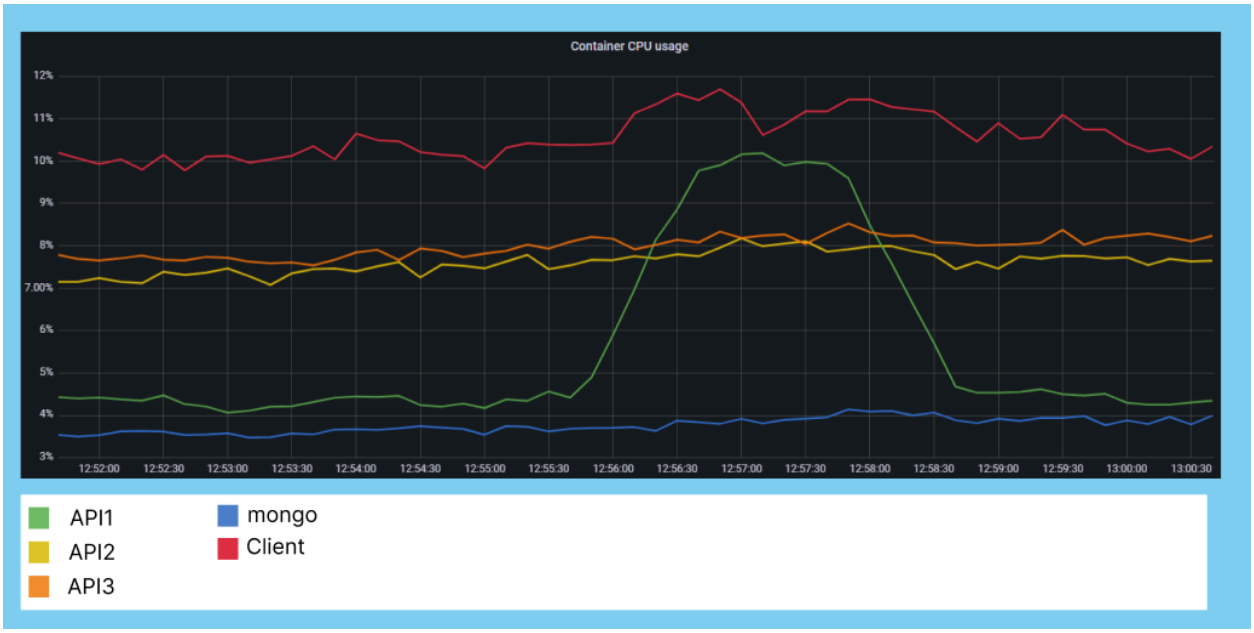


Figure 37: CPU Usage (% of host) for each docker service during Experiment 1.1 across time

```yaml
deploy:
  replicas: 1
  resources:
    limits:
      cpus: '0.1'
      memory: 100M
```

Figure 38: Constraints on APIs defined in swarm.yml file

Memory

As seen in figure 39 for the duration of the attack memory usage seems to go up slightly in API1. I could not find a good explanation for this however it is entirely possible that there is some memory allocated to the gremlin agent itself to increase the CPU usage of the container.

Figure 39: Memory usage for each container (MiB) during Experiment 1.1 across time

Network input

For the container metrics regarding network input and output there were no significant changes.

Response times and error rate

Regarding response times I had two metrics to consider, the average response times for the Locust test shown for each endpoint as well as the current response times (95th percentile). As seen in figure 40, response times for API1 are the lowest compared to the rest, due to simplicity. There is an increase however in its response time, increasing from around 5ms to its peak at 10ms before slowly going back down. I could not find any significant increases in response times for the rest of the containers however whilst API1 was overloaded. It was essentially impossible to gather any meaningful data from the aggregated endpoint current response times metric (figure 41), probably because the shift in response times compared with the other endpoints was minute.

It seems the original hypothesis remains true for this experiment.

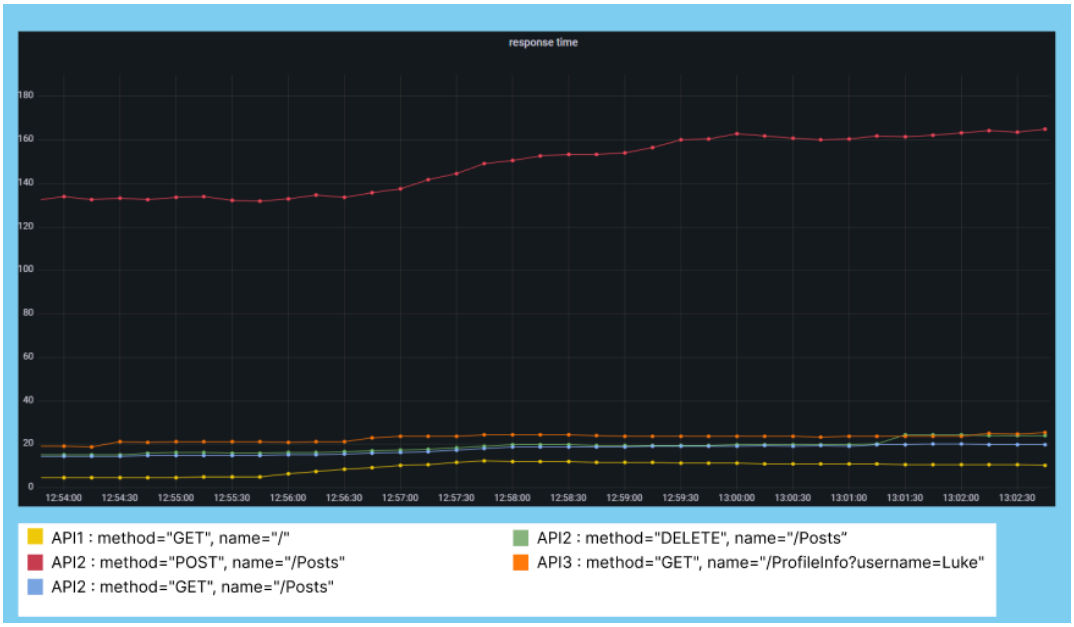


Figure 40: Average response times (ms) for endpoints during Experiment 1.1 across time

Figure 41: Current Response time for endpoints aggregated (ms) during Experiment 1.1 across time

Regarding error rate there was no significant increase in the duration of the attack shown by figure 42.

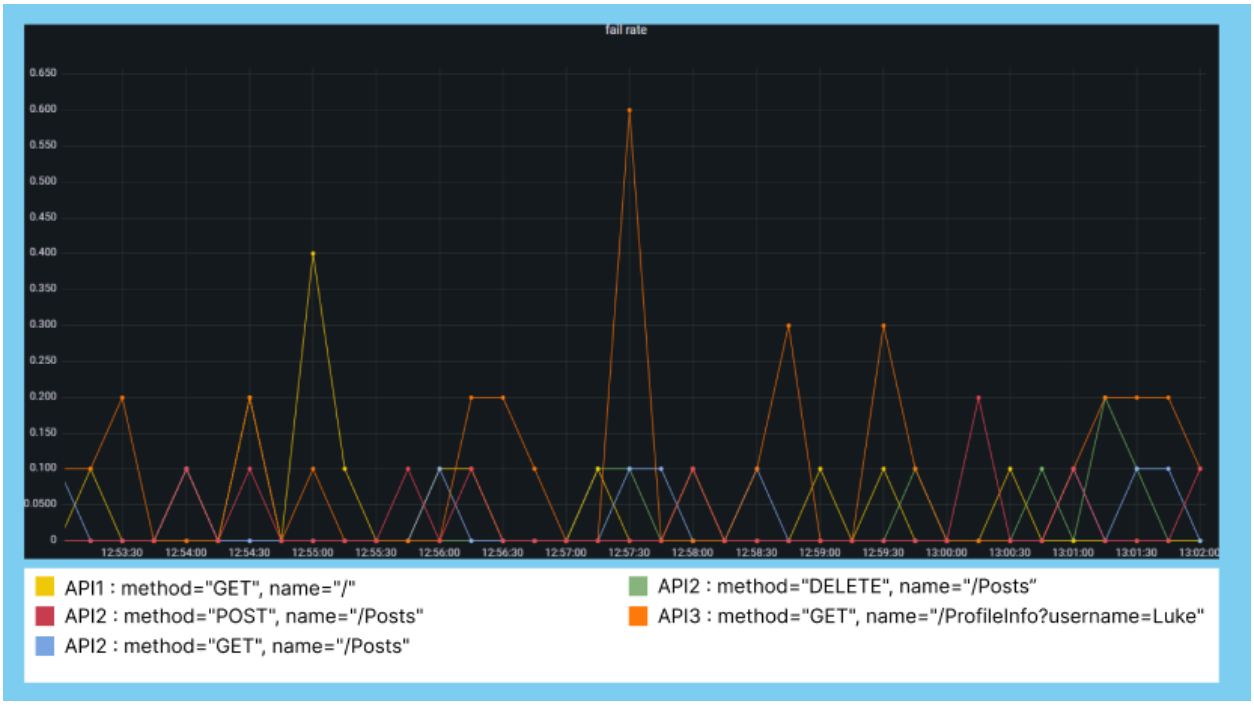


Figure 42: Error rate for endpoints (per second) during Experiment 1.1 across time

Experiment 1.2

Hypothesis:

- Upon overloading CPU usage of API2, response rates and error rates go up however other services are not adversely affected in terms of response time. After the attack, response times and error rates start returning to normal.

Attack Description:

- I exhausted the CPU capacity consuming 100% of usage on all cores regarding the API2 container. This attack was carried out for 120 seconds between **12:23 pm** and **12:25pm.**

Results and Evaluation

CPU and Throttle

I will not discuss the CPU usage and throttle again as it is essentially the same situation as experiment 1.1. It is also similar for memory usage.

Network Input

It seems like network input slightly decreased in the API2 container for the duration of the attack for some unknown reason shown by figure 43.

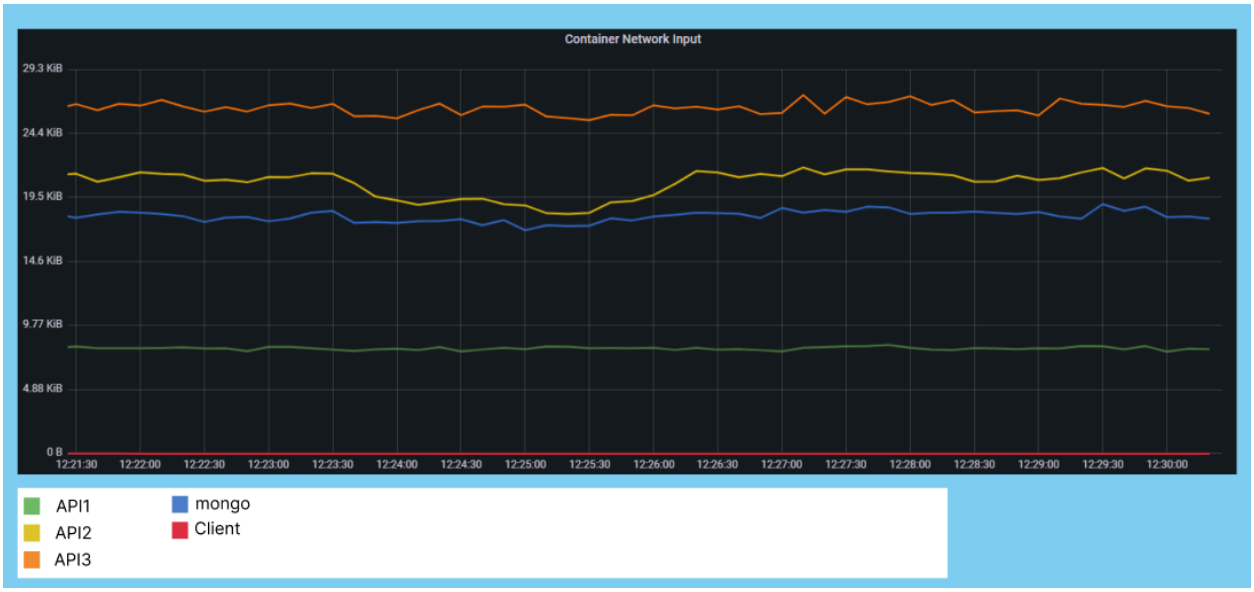


Figure 43: Network Input for containers (KiB) during Experiment 2.1 across time

Response times and error rate

Compared to API1 there seems to be a much greater impact on response rate when CPU is exhausted in API2. As seen in figure 44 all three endpoints for API2 significantly increased regarding response times when the attack started, for example at steady state the POST endpoint for adding posts to the forum was dealing with requests in around 75ms however at its peak during the attack the response time reached around 475ms which is over 6 times greater. It seems like the POST endpoint in API2 has a greater response time both before and after the attack, suggesting that writing to the mongo database through mongoose is more computationally expensive than reading or deleting. There seemed to be no impact on the endpoints for API1 or API3. It is possible that on continuation of the attack the response times for the API2 endpoints would have continued to rise as they were still on an upward trajectory, as seen in the figure once the attack stops at 12:25 the average response times for the API2 endpoints immediately start to decrease.

Unlike experiment 1.1 results for current response rate (95th percentile) were more significant. It's clear that around 12:23 in figure 45, when failure injection begins, overall response times for the endpoints increase significantly, before returning to normal after the attack finishes.

Regarding error rate in figure 46 it can be seen error was far more affected by the attack on API2 than API1. There is a notable increase in error rate for all three endpoints regarding API2 for the duration of the attack. The difference is likely due to the fact that requests sent to the API1 endpoint require very little CPU so occupying the containers CPU usage had little effect, whilst API2 endpoints involved interaction with a database meaning CPU exhaustion causes these tasks to be executed more slowly and incur more error.

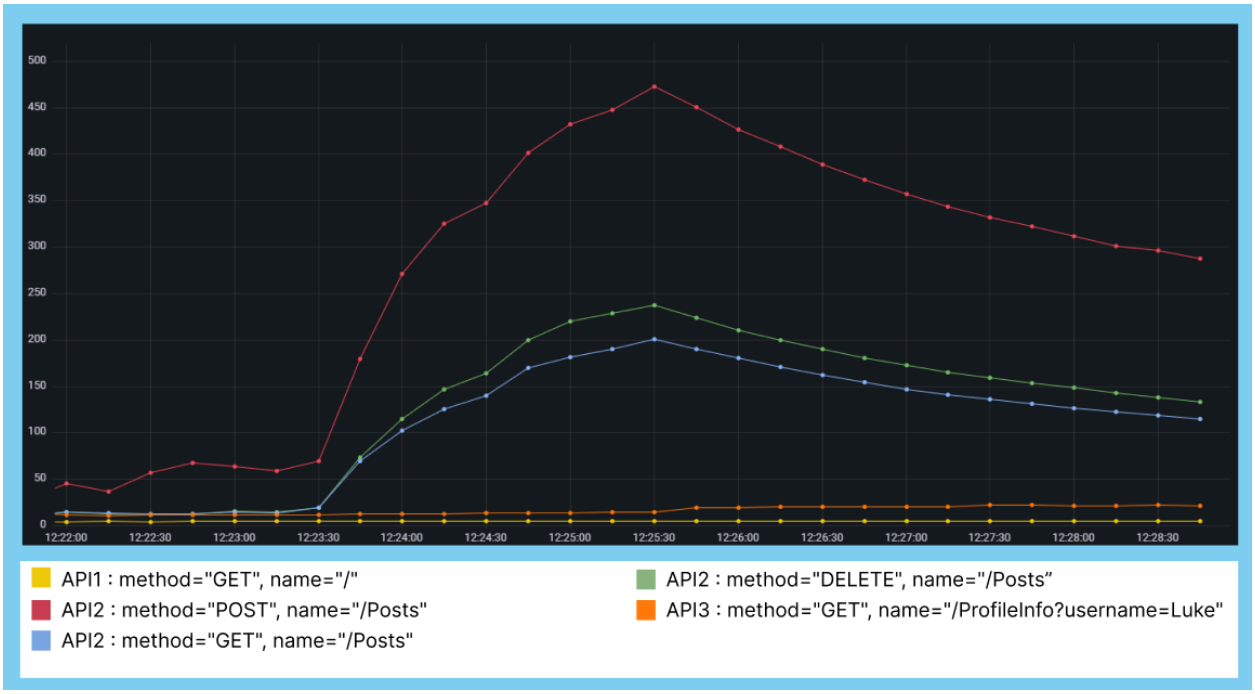


Figure 44: Average response times (ms) for endpoints during Experiment 1.2 across time

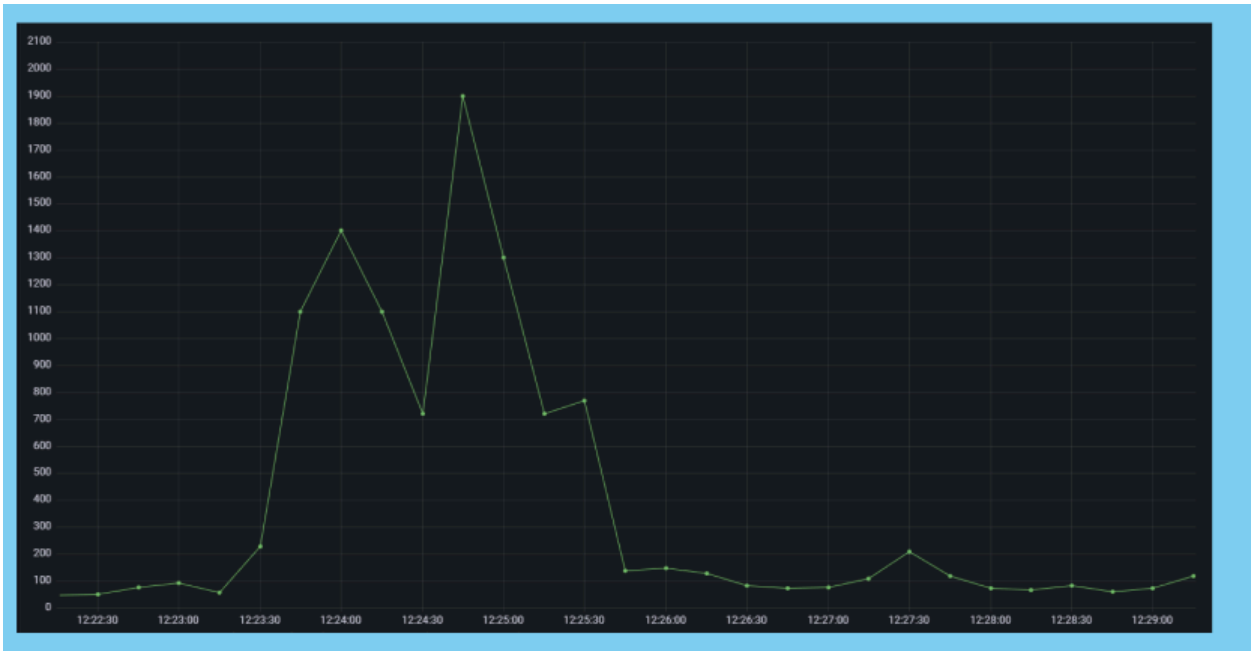


Figure 45: Current Response time for endpoints aggregated (ms) during Experiment 1.2 across time

Figure 46: Error rate for endpoints (per second) during Experiment 1.2 across time

In conclusion for experiment 2 the response times were much more heavily affected in comparison to experiment 1.1 however the original hypothesis remains valid, the response times and failure rate did increase however after the attack concluded they went back to previous levels.

Experiment 1.3

Hypothesis:

- Upon overloading CPU usage of API3, response and error rates go up however other services are not adversely affected in terms of response time. After the attack, response times and error rates start returning to normal.

Attack Description:

- I exhausted CPU capacity consuming 100% of usage on all cores regarding the API3 container. This attack was carried out for 120 seconds between **12:32 pm** and **12:35pm.**

Results and Evaluation

I won't discuss CPU usage and throttle again as it is essentially the same situation as experiment 1.1 It is also similar for memory usage. There was no significant change in the network input or output metric.

Response times and error rate

Shown in figure 47, when the CPU attack on API3 occured at 12:33, response times for the POST request in API2 as well as the GET request in API3 both increased. This somewhat mirrored initial findings when carrying out Locust tests. As shown in code earlier (figure 17), a request is sent to API3 to retrieve profile information. Once the request is served and the server has responded with a success or failure response, the request promise is fulfilled. It seems like by overloading API3, this promise takes longer to fulfill which in turn makes the request time for the API2 POST request longer. The endpoints for the other two API2 requests are not affected by API3 being overloaded as they make no requests to API3. The current response time is also affected considerably by exhausting CPU in API3 as seen in figure 49.

Shown in figure 48, failure rate for API3 is significantly affected however the API2 endpoints are not affected despite response times for the POST request increasing significantly. This supports the hypothesis that other services are not affected, however the statement about response times not being significantly affected does not hold true as discussed. It seems that response times and failure rates do go back to normal after the attack concludes.

In conclusion the original hypothesis is disproven as injecting failure into API3 increases response times in API2.

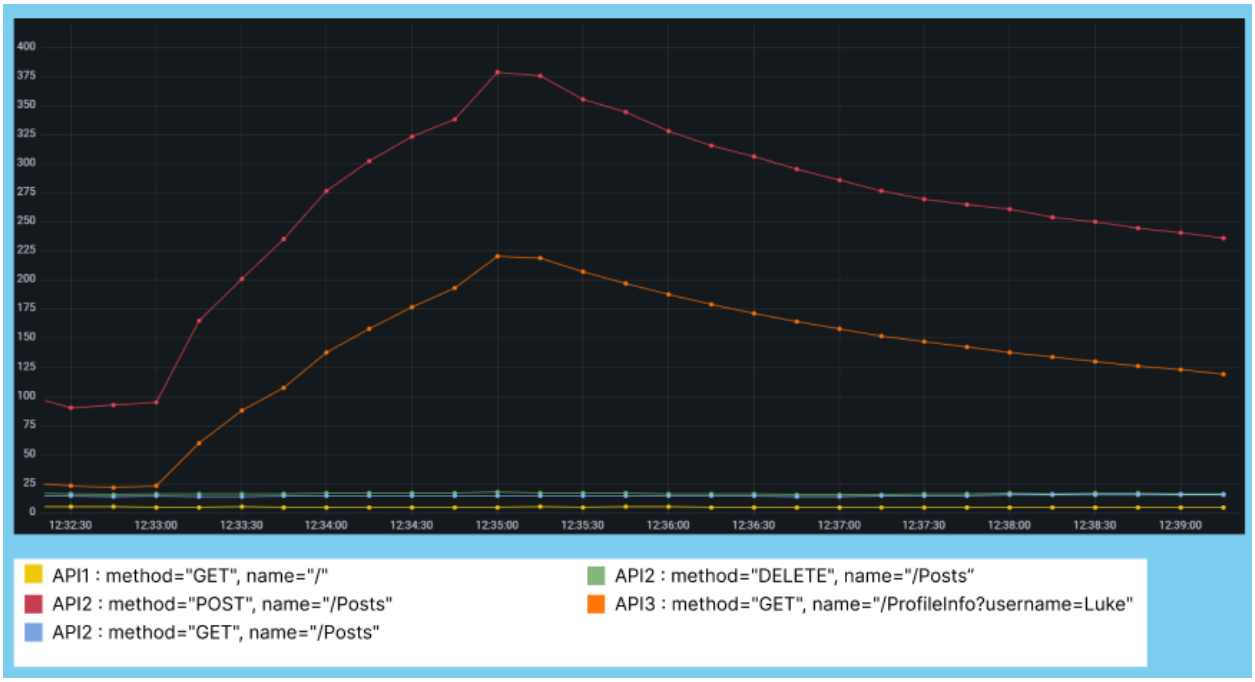


Figure 47: Average response times (ms) for endpoints during Experiment 1.3 across time

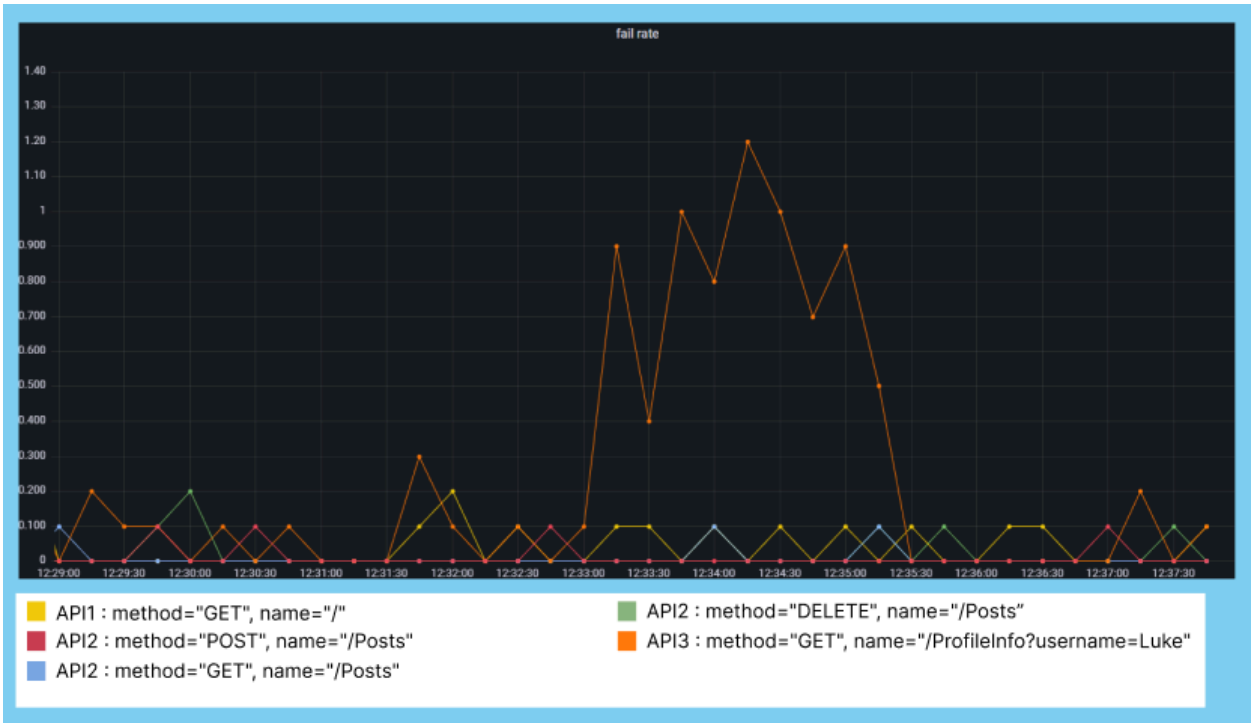


Figure 48: Error rate for endpoints (per second) during Experiment 1.3 across time

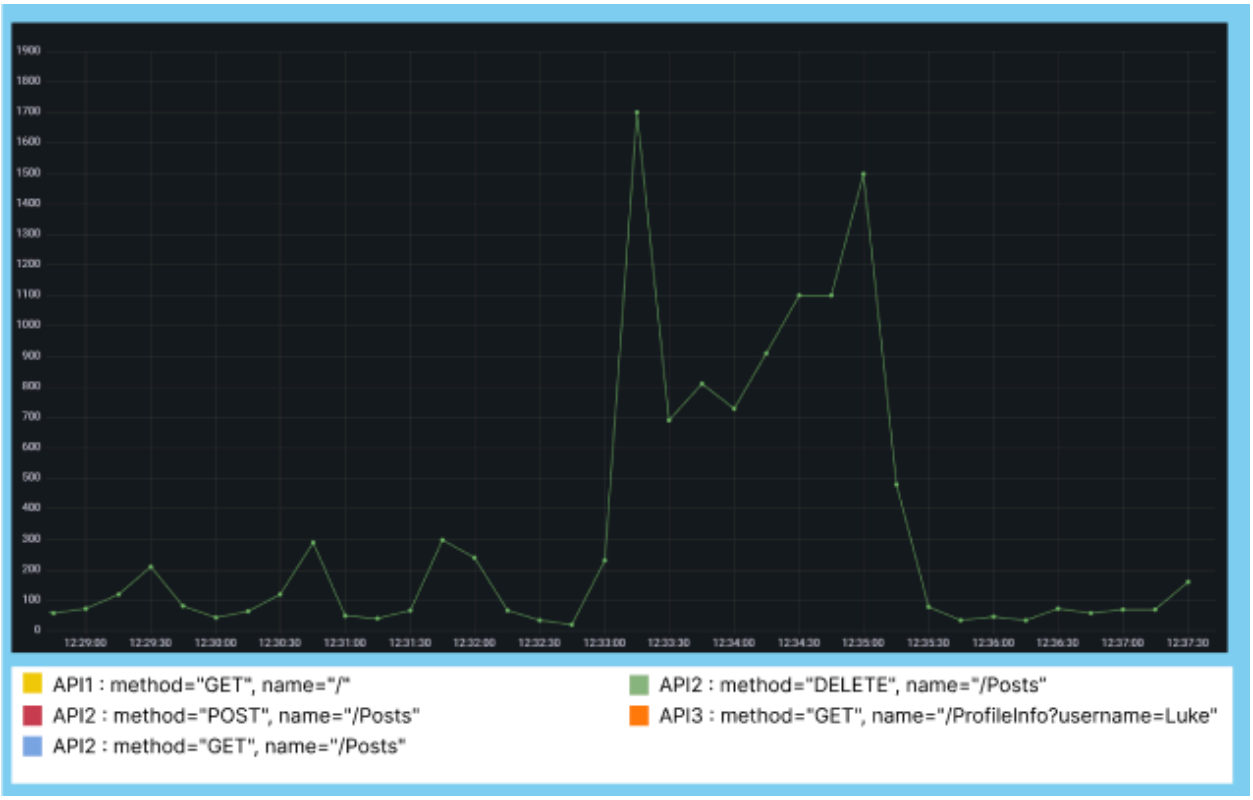


Figure 49: Current Response time for endpoints aggregated (ms) during Experiment 1.3 across time

Experiment 1.4

Hypothesis:

- Upon overloading CPU usage of the mongo container, response and error rates go up regarding all endpoints involving database interaction. After the attack, response times and error rates start returning to normal.

Attack Description:

- I exhausted the CPU capacity consuming 100% of usage on all cores regarding the mongo database container. This attack was carried out for 120 seconds between **1:10pm** and **1:12pm.**

Results and Evaluation

I will not discuss CPU usage and throttle again as it is essentially the same situation as in experiment 1.1. It is also the same for memory usage. There was also no significant change in the network input or output metric.

Response times and error rate

Shown in figure 50 whilst the attack runs all response times for endpoints in API2 and API3 increase AP1s don't. The effect on response time is greatest regarding the POST request in API2. It seems like the effect on response times for each endpoint is not as great when overloading the mongodb service in comparison to overloading the endpoints respective service. This is probably because the only code affected by the attack in each endpoint will be code sending requests through the mongoose API to update and read from the database. This added wait time is most likely the reason for higher response times in the API2 and API3 endpoints.

It is shown in figure 51 that overloading the mongo service does not significantly increase error rate in any services. I speculate this is because any code not involving mongo interaction will not be affected by the attack, an increase in response time waiting for the database update request to return will not affect the error rate of the API2 or API3 endpoint itself.

Regarding the current response times although not as stark as other attacks, a small but consistent increase in response times can be seen in figure 52 which was to be expected.

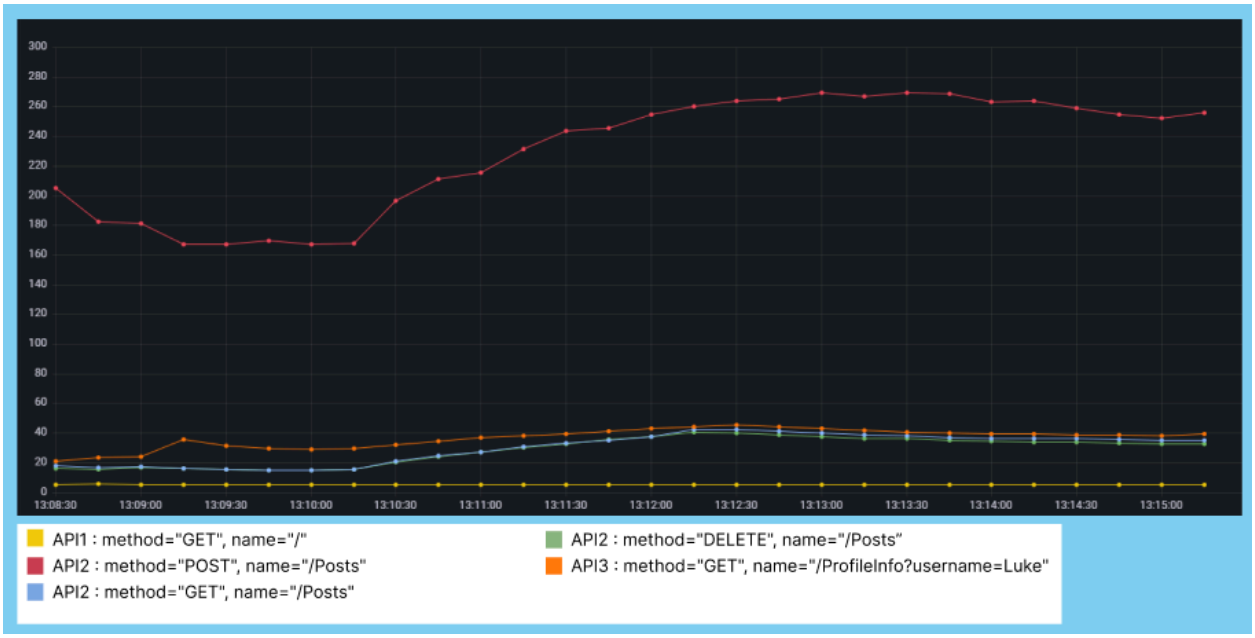


Figure 50: Average response times (ms) for endpoints during Experiment 1.4 across time

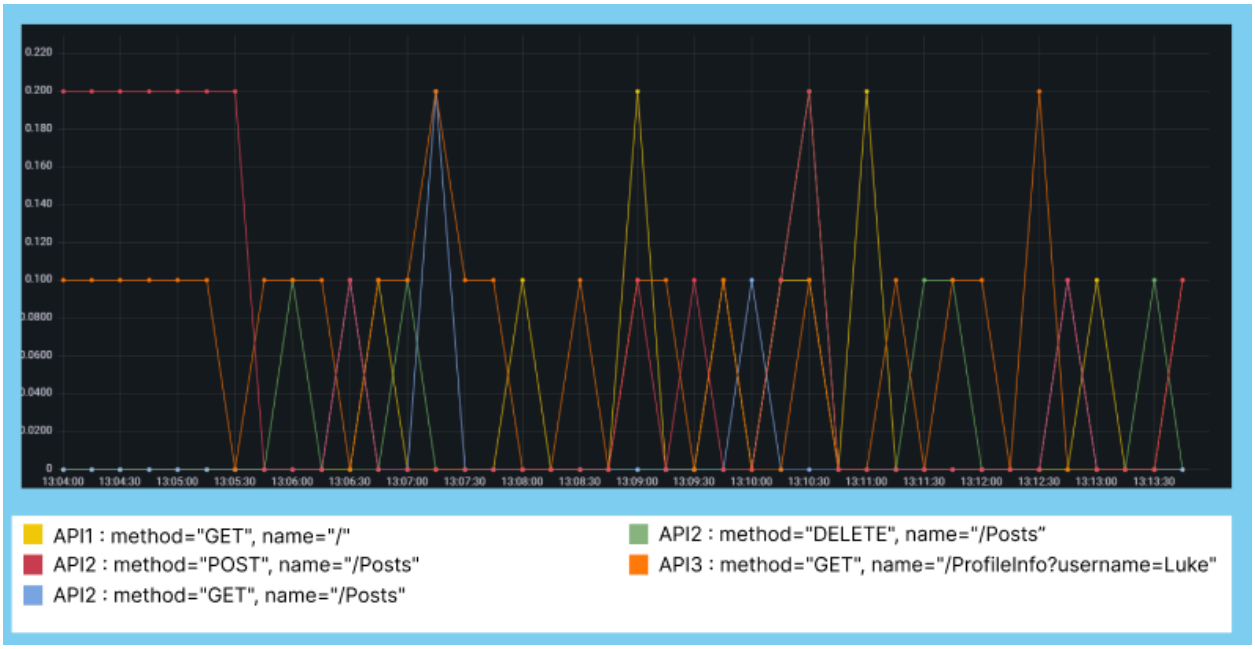


Figure 51: Error rate for endpoints (per second) during Experiment 1.4 across time

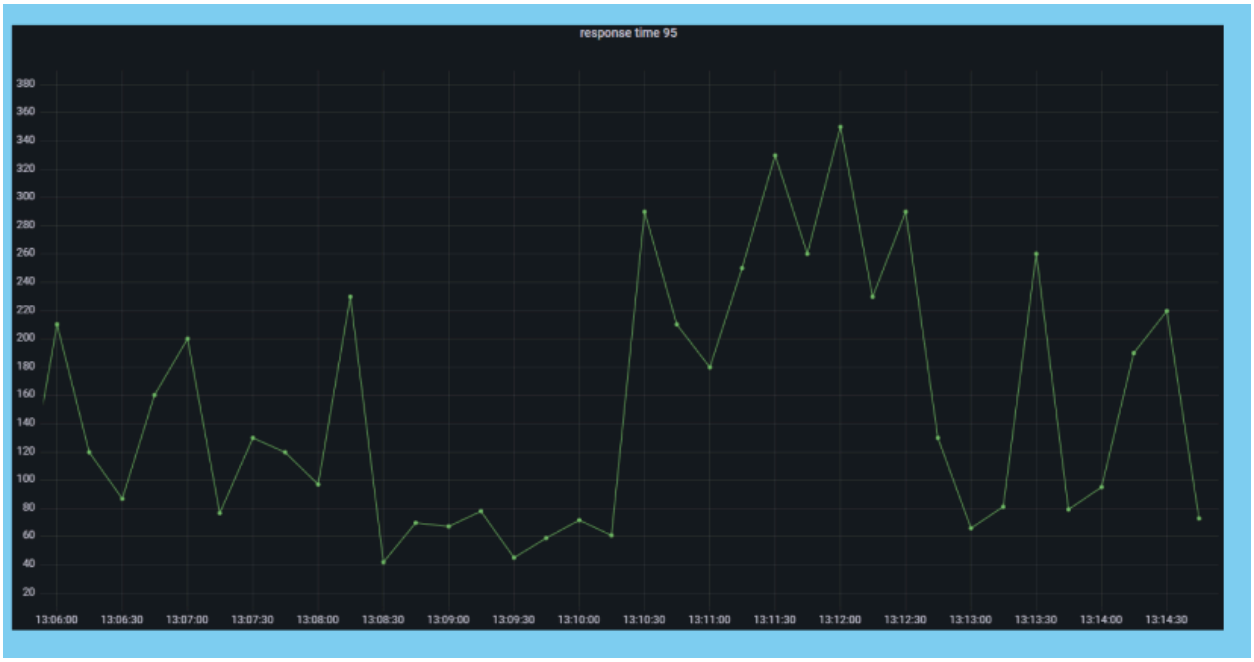


Figure 52: Current Response time for endpoints aggregated (ms) during Experiment 1.4 across time

Experiment 1.5

Hypothesis:

- Upon overloading CPU usage of all services, response and error rates go up regarding all endpoints. After the attack, response times and error rates start returning to normal.

Attack Description:

- I exhausted the CPU capacity consuming 100% of usage on all cores for all services. This attack was carried out for 120 seconds between **1:37pm** and **1:39pm.**

Results and Evaluation

CPU and Throttle

Shown in figure 53, CPU throttle increases for all containers to limit CPU to 10% of host for each. Memory usage increased slightly for all containers as expected and there were no significant changes in network input/output.

Figure 53: CPU Throttle for each docker service during Experiment 1.5 across time

Response times and error rate

As seen in figure 54 response times for all endpoints increased when the attack was run on all services which was to be expected. Consistently with previous experiments the POST endpoint in API2 was the most adversely affected regarding response times. I'd also like to note that the increase in response times in each of the endpoints was similar when all services were attacked in comparison to individual services being attacked. For example the increase in average response time for the GET API3 endpoint to peak response time when attacked alone was about 200ms, when attacked alongside all other attacks the increase was actually slightly less at only 150ms. There was a similar situation for the rest of the endpoints. This if anything suggests that putting one container under stress in this system does not negatively affect the response times of the other containers.

Figure 55 shows a large increase in current response times for the duration of the attack. Figure 56 shows a significant increase in error rate for all endpoints for the duration of the attack. It

seems for whatever reason API3 is the most vulnerable to increases in error rate when the corresponding container is put under stress.

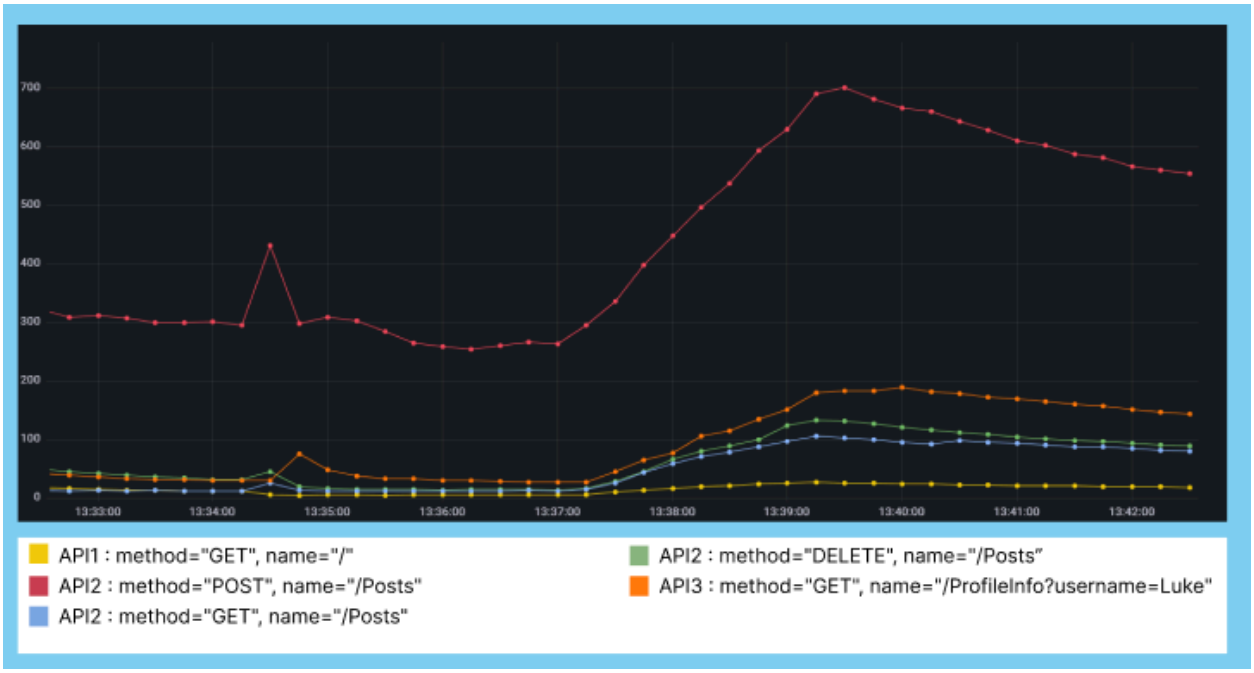


Figure 54: Average response times (ms) for endpoints during Experiment 1.5 across time

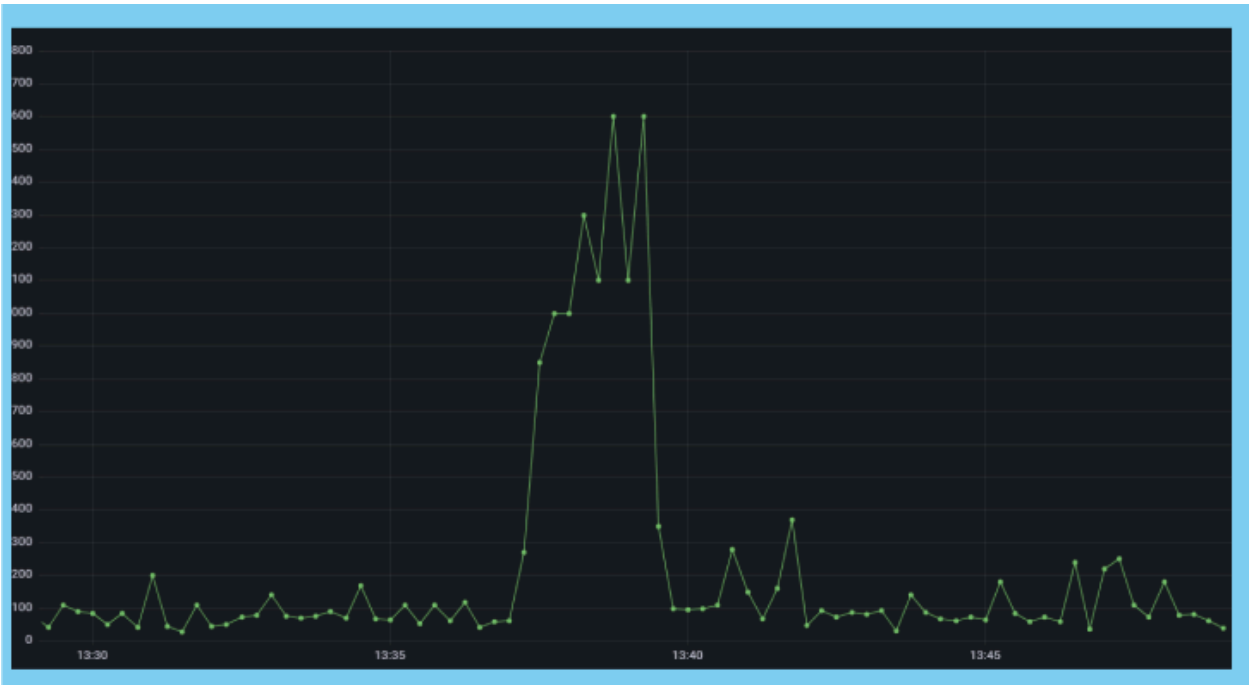


Figure 55: Current Response time for endpoints aggregated (ms) during Experiment 1.5 across time

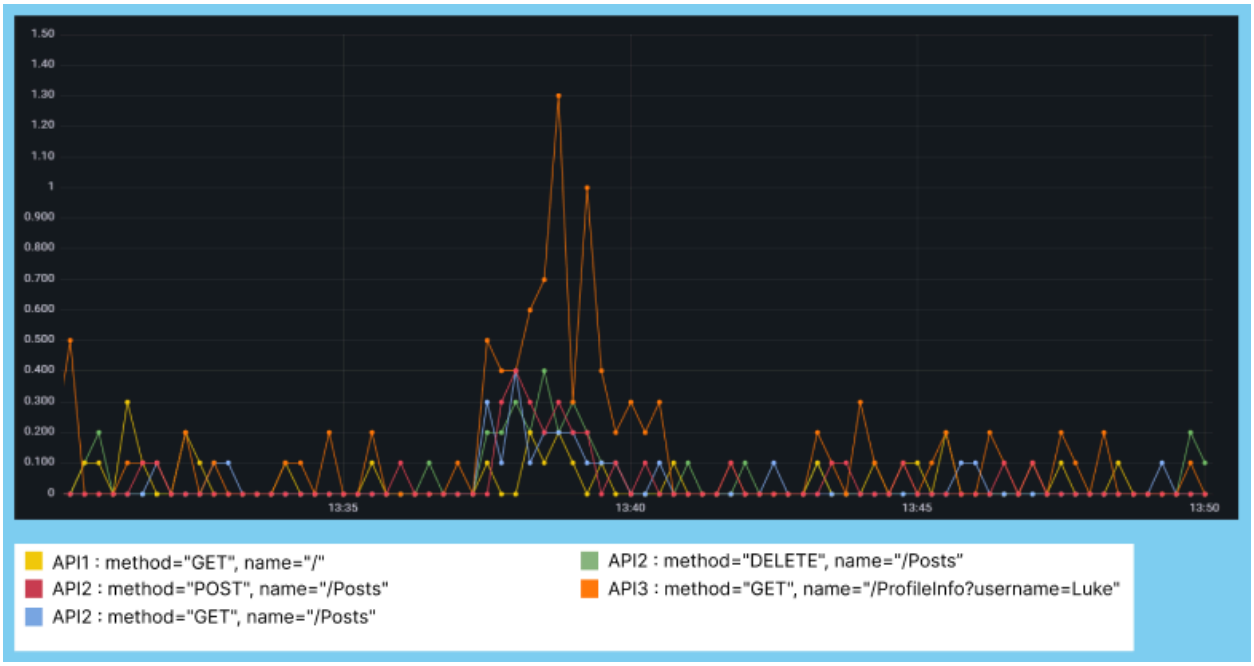


Figure 56: Error rate for endpoints (per second) during Experiment 1.5 across time

Further CPU Experiments

Experiment 1.6

To investigate the effect of removing CPU usage constraints of my containers in experiments I ran tests as before but without resource constraints.

I ran a test where I utilized 100% of CPU in API2 from **9:55pm** to **9:57pm** with no resource constraints on any services.

Results and Evaluation

CPU

As seen in figure 57 when there are no constraints on CPU usage, CPU usage reaches very high levels as more of the host machine CPU cycles are allocated to the containers work. This is far more than previous where CPU only reached 10% of host CPU capacity.

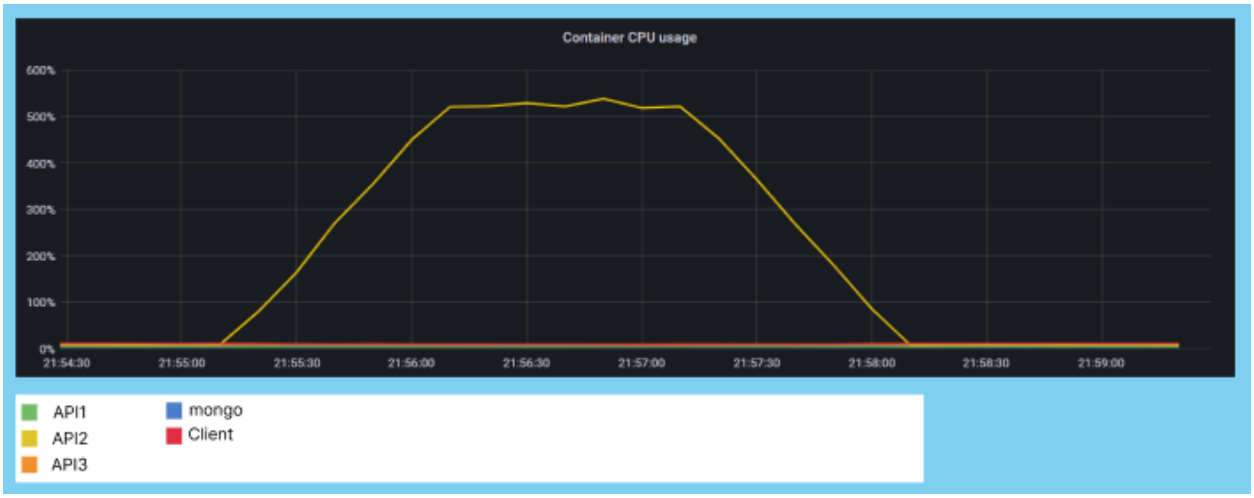


Figure 57: CPU Usage (% of host) for each docker service during Experiment 1.6 across time

Response times and Error rate

Seen in figure 58 response times go up slightly peaking at 30ms. It is clear that response rates at both steady state and during attacks are far lower than when there were constraints on services, at around 15ms to 20ms at steady state. Whereas without constraints response times at steady state fall more in the 100 to 200 range. It is unclear whether this is due to removal of the 10% CPU constraint or the 100MB memory constraint.

Response times for all other endpoints increase slightly albeit only ranging around 2ms to 8ms. This suggests the increased usage of CPU by the host machine means less CPU cycles are allocated to various services running, increasing execution times which did not occur when constraints were used. Shown in figure 60 showing current response times, there is a clear increase however response times before and after failure injection are far lower, suggesting that putting constraints on services has a considerable impact on response times stressed or not.

Shown in figure 59, failure rate does not increase from steady state, indicating the lack of resource constraints somehow decreases number of errors.

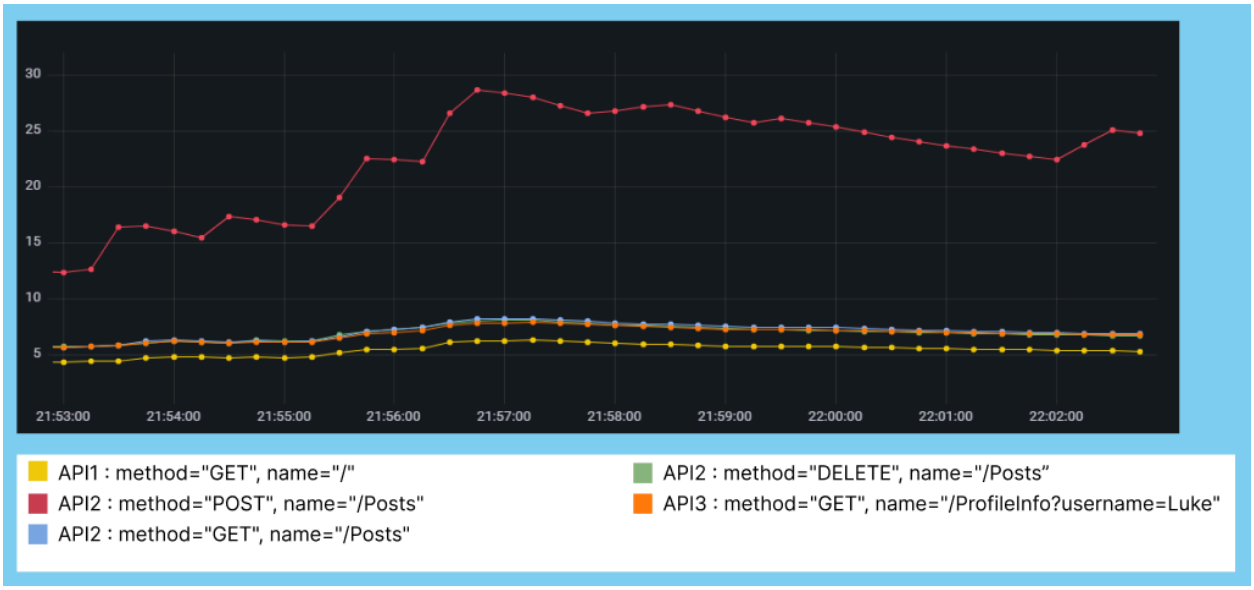


Figure 58: Average response times (ms) for endpoints during Experiment 1.6 across time

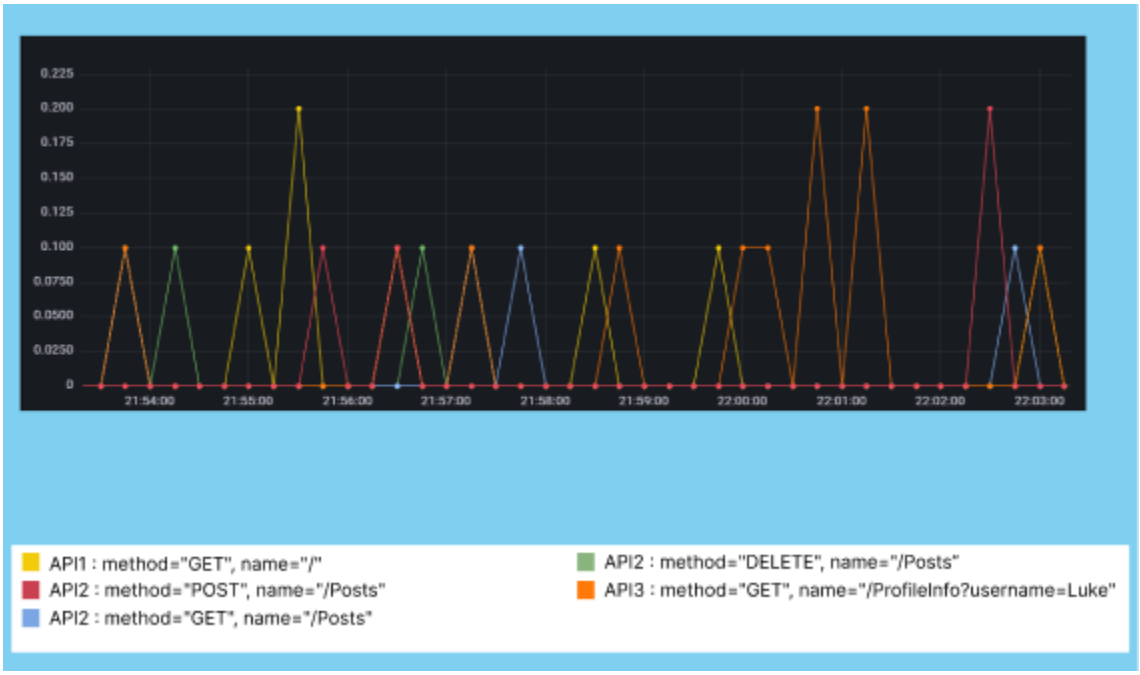


Figure 59: Error rate for endpoints (per second) during Experiment 1.6 across time

Figure 60: Current Response time for endpoints aggregated (ms) during Experiment 1.6 across time

Experiment 1.7

I'd like to briefly mention I carried out a test without constraints attacking all services. This was carried out from **10:10pm** to **10:13pm**

Results and Evaluation

Shown in figure 62 during the attack, CPU usage did not seem to escalate at the same time for all services like it did for other experiments. This was probably due to a lack of CPU availability for the Gremlin agent therefore it could not carry out attacks as efficiently leading to them staggering, meaning results are slightly less meaningful.

It can be seen in figure 61 that similarly to the attack on just API2, all endpoints had increased response times, however peak response rates rose slightly higher. For example in the attack on just API2, response rates peaked at around 28ms however in the attack on all services the response rates for POST in API2 peaked at 47ms. It was apparent that lack of resource constraints resulted in lower response times in general, supporting my background research.

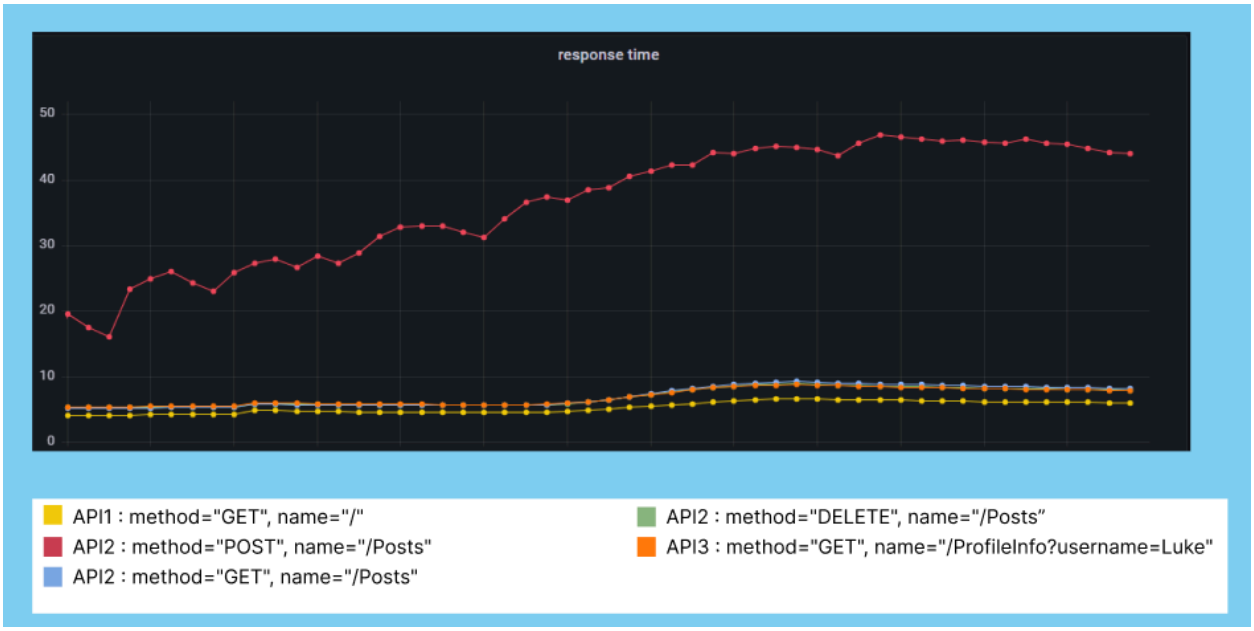


Figure 61: Average response times (ms) for endpoints during Experiment 1.7 across time

Figure 62: CPU Usage (% of host) for each docker service during Experiment 1.7 across time

Experiment 1.8

As discussed in experiment 1.3, when overloading the CPU of API3, due to reliance on API3 for retrieving data in the API2 POST endpoint, response times are affected. I ran a similar experiment however, decreased the timeout threshold for the request to API3 from 3000ms to 200ms. I hypothesized this would decrease response time increase of the API2 POST endpoint when overloading API3. I chose 200ms as during attacks the average response time for the API2 POST endpoint always exceeds this however during steady state it is always below.

This experiment was ran from **10:35pm** to **10:38pm**

Results and Evaluation

As seen in figure 63, average response time for the API3 endpoint increases significantly. The API2 endpoint also increases its response times however not to the degree it had during experiment 1.3. During experiment 1.3 response times for the API2 POST endpoint always exceeded API3s however here the average response time for API3 clearly exceeds the API2 one during the attack.

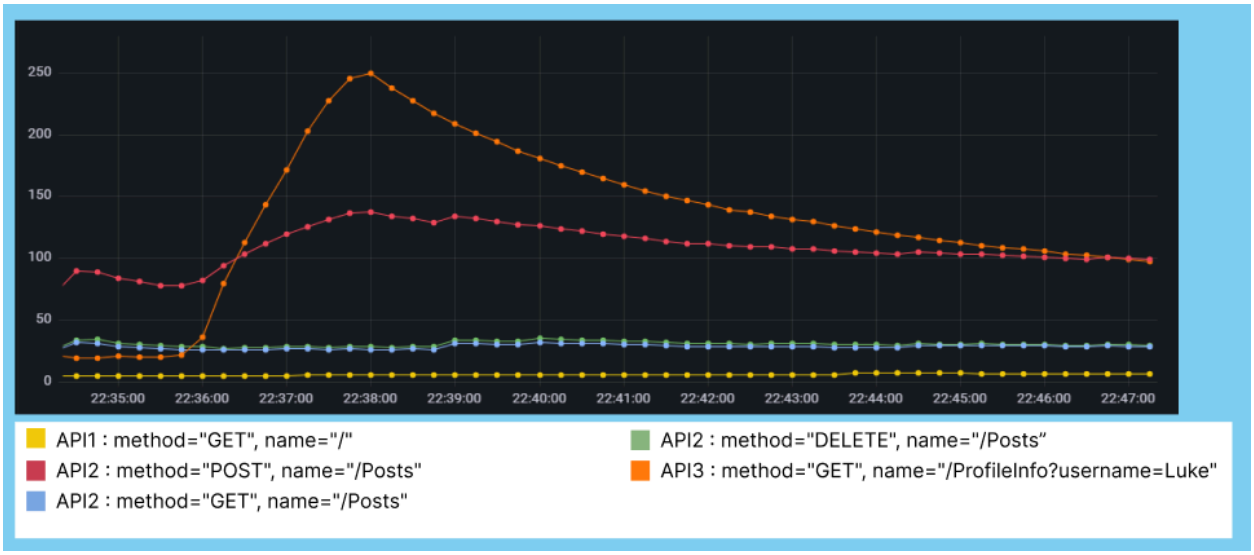


Figure 63: Average response times (ms) for endpoints during Experiment 1.8 across time

Memory attacks

Once I finished CPU attacks I proceeded to memory attacks. These test my system against memory consumption to ensure they can perform given a sudden increase in usage. I did run a memory attack with no memory constraints however this kept crashing my computer and I could not gain any results.

Experiment 2.1

Hypothesis:

- Upon overloading memory usage of API1, response and error rates go up in API1. After the attack, response times and error rates start returning to normal.

Attack Definition:

- I exhausted the memory capacity to 100% of usage in API1. This attack was carried out for 120 seconds between **3:56pm** and **3:59pm.**

Results and Evaluation

CPU

As shown in figure 64 for the duration of the memory attack on API1, CPU usage for API1 actually increased substantially, this was also the case for CPU throttle (figure 65). These results were not expected and could be considered emergent.

I did research to explain this trend although none was conclusive [18]. If a computer is under much stress and its RAM is at full capacity, a system involving ‘Pagefile’ can be used. Pagefile is a hidden system file usually stored on a C drive on Windows. The Pagefile allows the computer to perform more smoothly by reducing workload of the physical memory or RAM. When there is more load than RAM can accommodate, the programs already present on the RAM are automatically transferred to the Pagefile which works as a secondary RAM in a process known as Paging.

This juggling of data to and from the disk and RAM requires the CPU which could potentially explain the increase in CPU usage. As Memory usage in the API container reaches full capacity of 100MB it is possible it makes use of Pagefile.

Figure 64: CPU Usage (% of host) for each docker service during Experiment 2.1 across time

Figure 65: CPU Throttle for each docker service during Experiment 2.1 across time

Memory usage

As expected, for the duration of the attack memory usage for API1 increased shown in figure 66. Memory usage did not increase for other containers. Once the attack was over, memory usage seemed to drop to a lower level than it was before the attack, this was the same for all

experiments regarding memory overload. This result was not expected and could be considered emergent as I don't know the cause.

Figure 66: Memory usage for each container (MiB) during Experiment 2.1 across time

Response times and error rates

As observed in figure 67, during the attack, the API1 endpoint average response time increases from around 5ms to about 27ms before falling back down once the attack concludes. Response times of other services are not affected. These results were expected by the hypothesis.

The increase in response time during the memory attack for API1 is slightly more than the CPU attack which only increased to a peak of 17ms however there is not enough data to make a conclusive decision on which has more effect.

Shown in figure 68, failure rate in API1 increases significantly for the duration of the attack. It can be noted that the CPU attack did not increase failure rate at all in API1, suggesting that surges in memory usage cause higher failure rate in executing code (no interaction with database) than CPU surges.

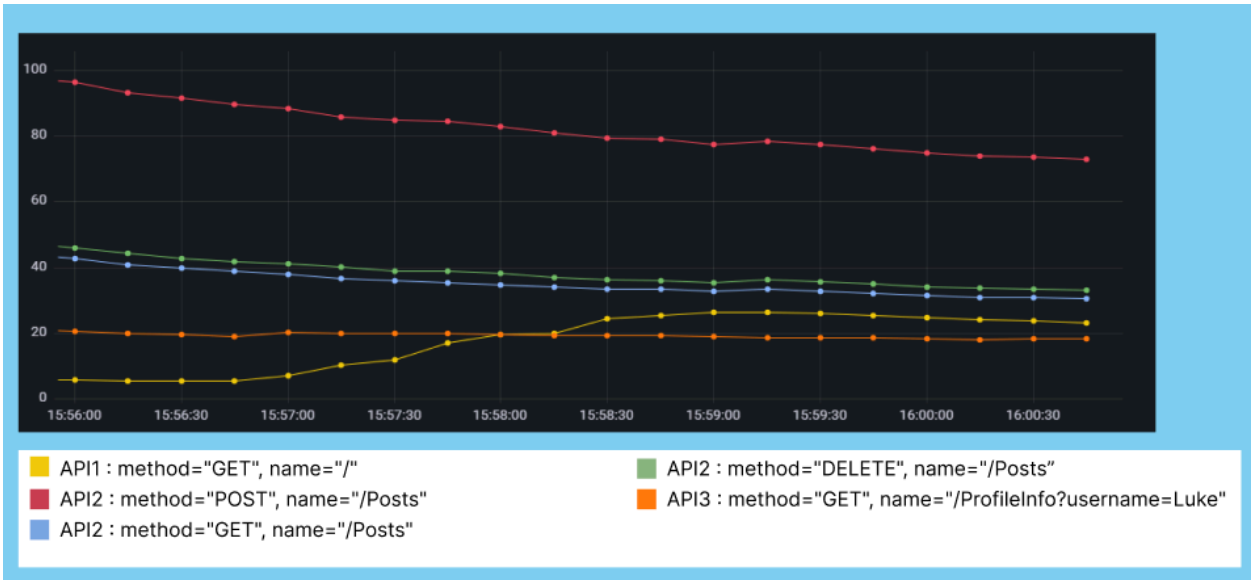


Figure 67: Average response times (ms) for endpoints during Experiment 2.1 across time

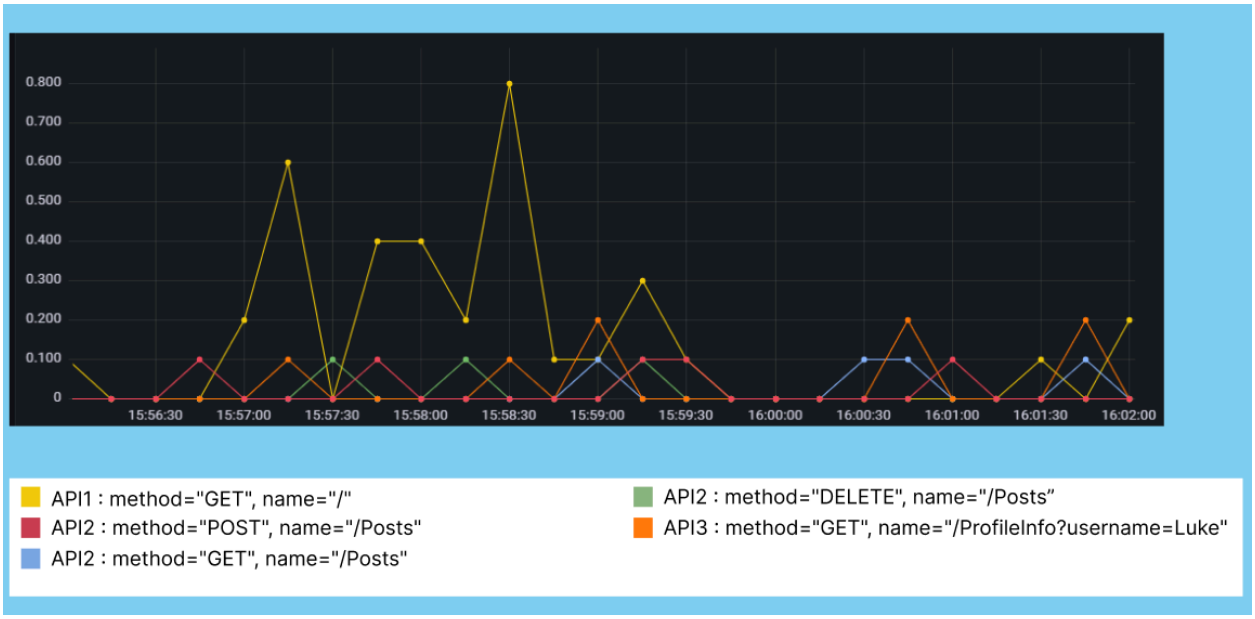


Figure 68: Error rate for endpoints (per second) during Experiment 2.1 across time

Experiment 2.2

Hypothesis:

- Upon overloading memory usage of API2, response and error rates go up in API2 endpoints. After the attack, response times and error rates start returning to normal.

Attack Definition:

- I exhausted the memory capacity consuming 100% in the API2 container. This attack was carried out for 120 seconds between **4:05pm** and **4:08pm.**

Results and Evaluation

CPU and memory

As in experiment 2.1, findings were essentially the same. CPU and memory usage increased in a similar way. Memory usage increased for the duration of the attack and then fell to lower levels than before the attack began.

Response times and error rates

As seen in figure 69 similarly to the CPU attack response times for all API2 endpoints increased for the duration as expected and then began to decline to steady state levels after the attack. It can be seen from figure 70 that the error rate increased in the API2 endpoints during the attack and then decreased to steady state levels afterwards, as expected.

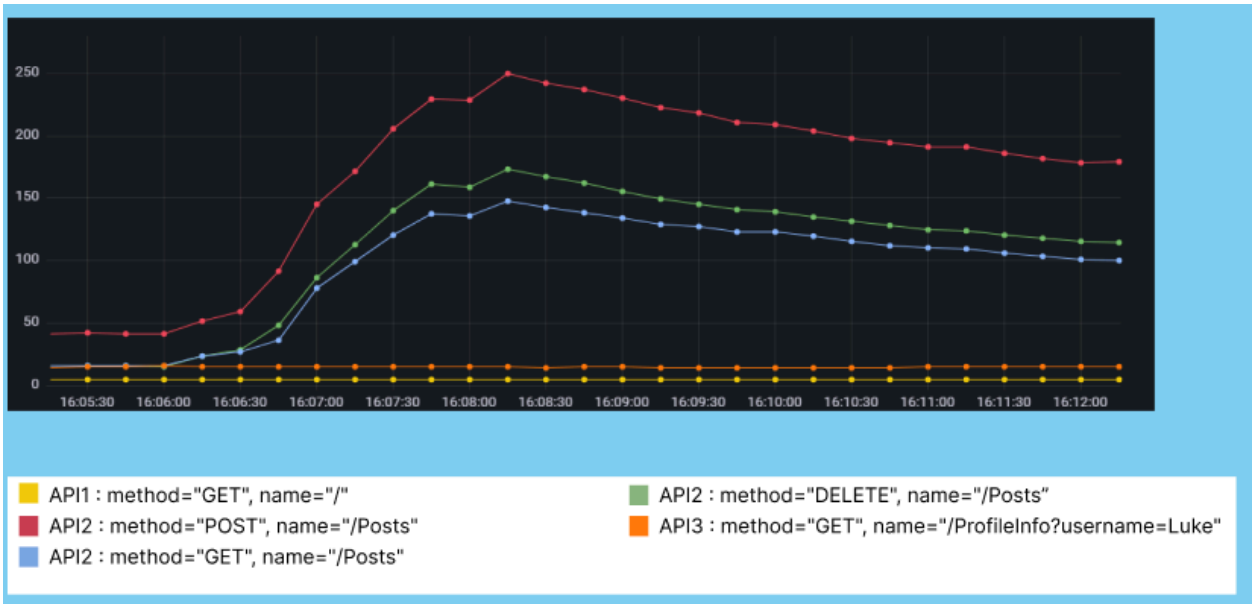


Figure 69: Average response times (ms) for endpoints during Experiment 2.2 across time

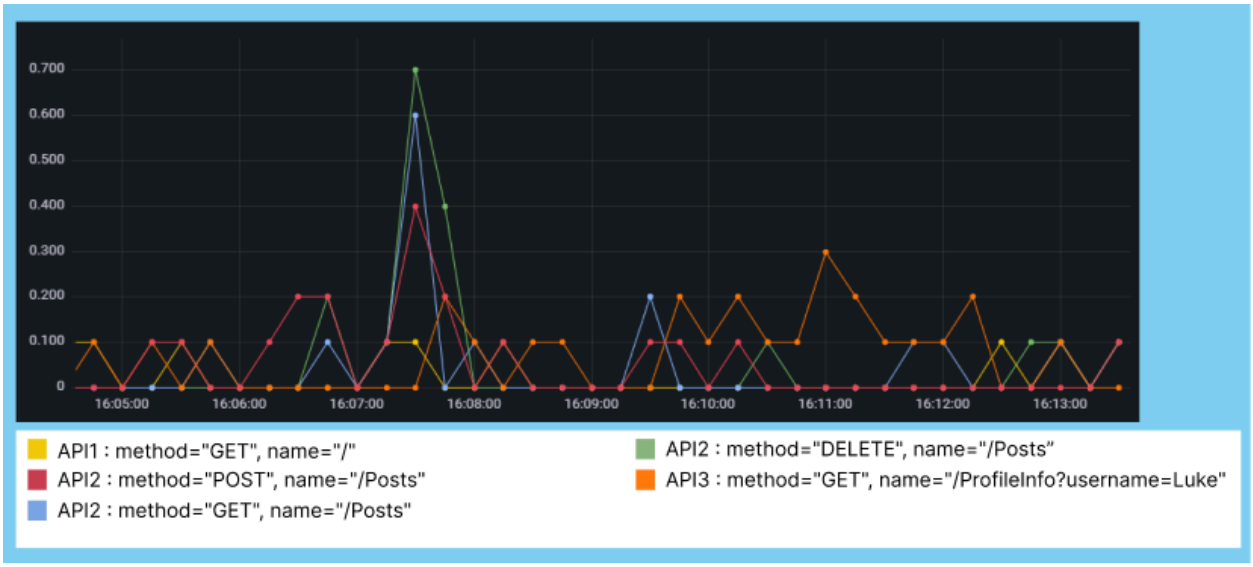


Figure 70: Error rate for endpoints (per second) during Experiment 2.2 across time

Experiment 2.3

Hypothesis:

- Upon overloading memory usage of API3, response and error rates go up in API3 endpoints. After the attack, response times and error rates start returning to normal.

Attack Description:

- I exhausted the memory capacity consuming 100% in the API3 container. This attack was carried out for 120 seconds between **4:16pm** and **4:18pm.**

Results and Evaluation

CPU and memory

Results for these metrics were similar for other memory attacks. See experiments 2.1 and 2.2.

Response times and error rates

As seen in figure 71 the response rate for the API3 endpoint increases significantly for the duration of the attack from only around 20ms to 300ms. The API2 POST increases, but to a lesser extent, note the timeout discussed earlier has been kept at 200ms which has limited reliance on API3. It is likely that response rate would have increased further if the timeout had been kept at 3000ms as in experiment 1.3. It can be clearly seen in figure 73 that current response times increase during the attack and then return to normal levels.

Figure 72 shows failure rate in API3 increases for the duration of the attack and then returns to normal as expected.

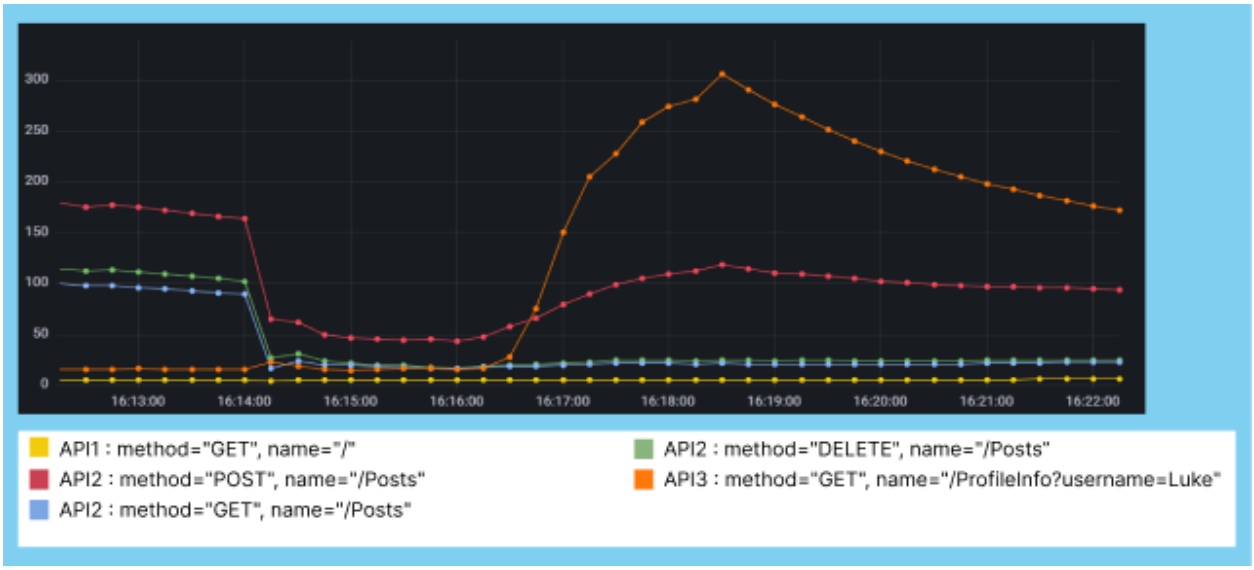


Figure 71: Average response times (ms) for endpoints during Experiment 2.3 across time

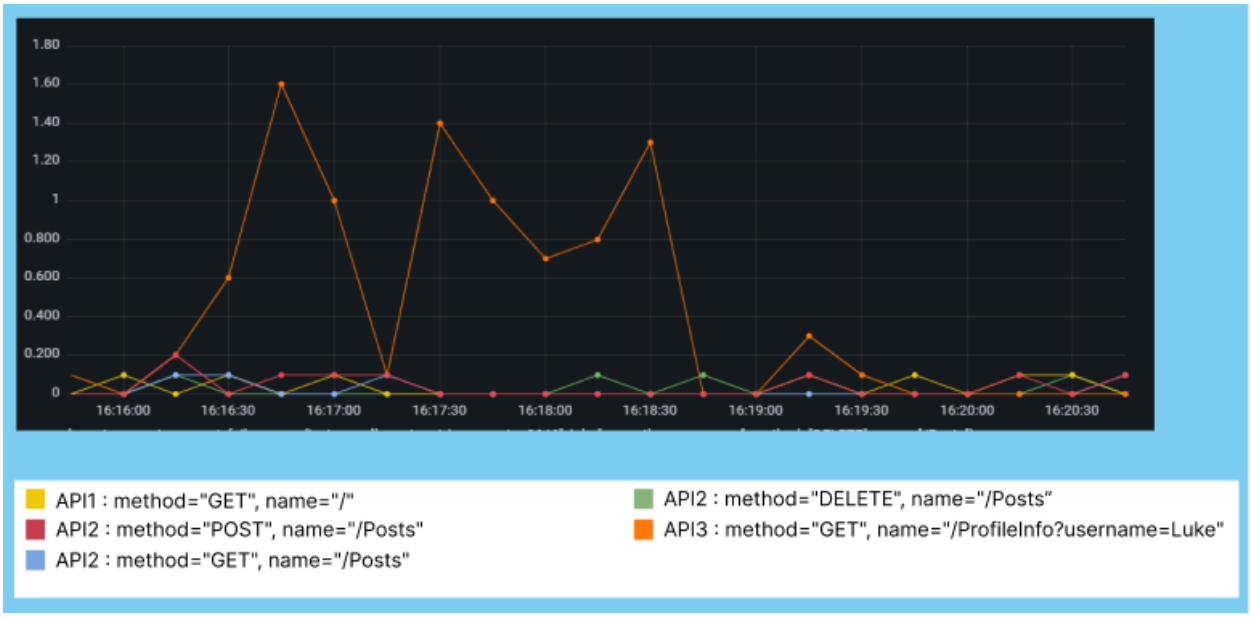


Figure 72: Error rate for endpoints (per second) during Experiment 2.3 across time

Figure 73: Current Response time for endpoints aggregated (ms) during Experiment 2.3 across time

Experiment 2.4

Hypothesis:

- Upon overloading memory usage of the mongo container, response and error rates go up in all endpoints involving database interaction. After the attack, response and error rates start returning to normal.

Attack Description:

- I exhausted the memory capacity consuming 100% of usage for the mongo container. This attack was carried out for 120 seconds between **7:26pm** and **7:27pm.**

Results and Evaluation

As seen in the figures and explanation below, this experiment caused a large number of unexpected results that could be considered emergent failures. As seen in figure 74 the attack was aborted one minute in, as the mongo container crashed.

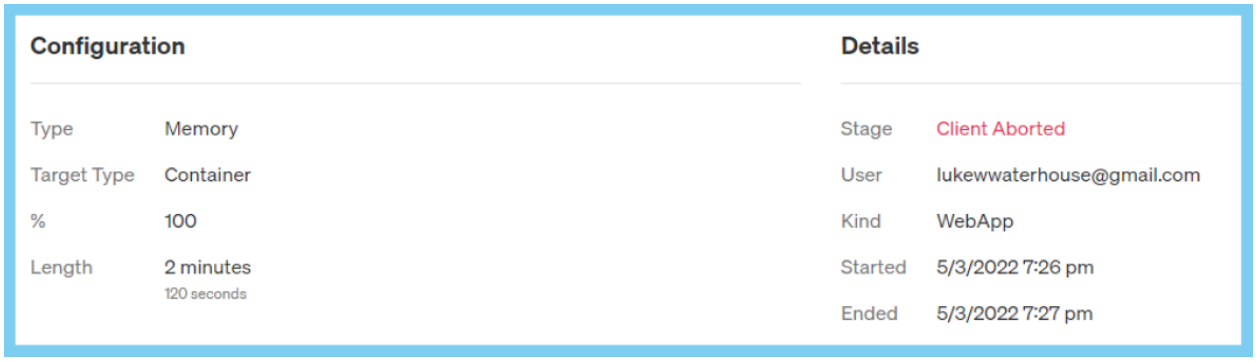


Figure 74: Gremlin UI displaying attack summary of Experiment 2.4

CPU

Shown in figure 75, CPU usage for API1 remained relatively constant for the duration of the attack. There is a spike in CPU activity in the client and API1 around 19:31, this is because when the container crashed I opened the front end to view the impact of the failures.

Around 19:27, CPU usage in the mongo container rapidly drops off, at this point a new mongo container is instantiated automatically by docker to replace it (seen in purple). CPU usage in this escalates and peaks at 10% where it is capped, then rapidly drops to a very low level of usage suggesting no data is being written or read from the mongo database.

At around the same time the API2 and API3 CPU usage rapidly starts to drop too, below the steady state level suggesting something is not functioning correctly.

The points I have made apply to the CPU throttle shown in figure 76, this mirrors the results from CPU usage.

Shown in figure 77 a similar pattern can be seen regarding memory usage, the mongo container crashes, shown by a discontinuation of data, followed by a new mongo container starting up, showing memory usage in purple. API2 and API3 memory usage also drops dramatically following this.

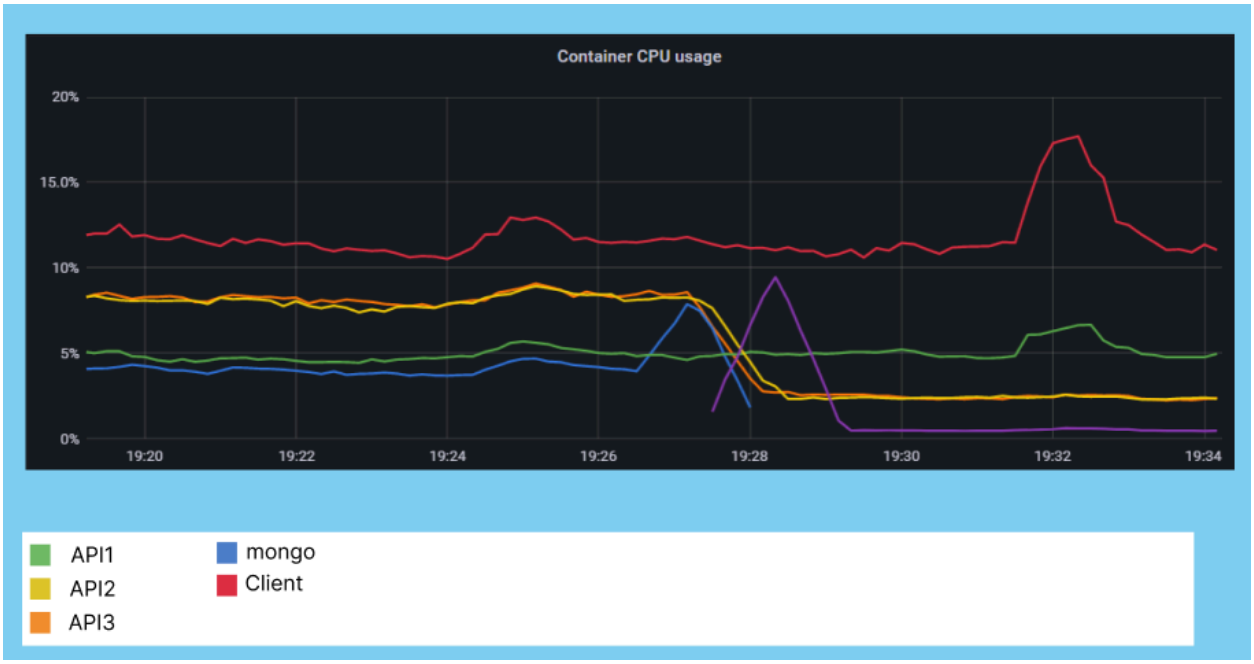


Figure 75: CPU Usage (% of host) for each docker service during Experiment 2.4 across time

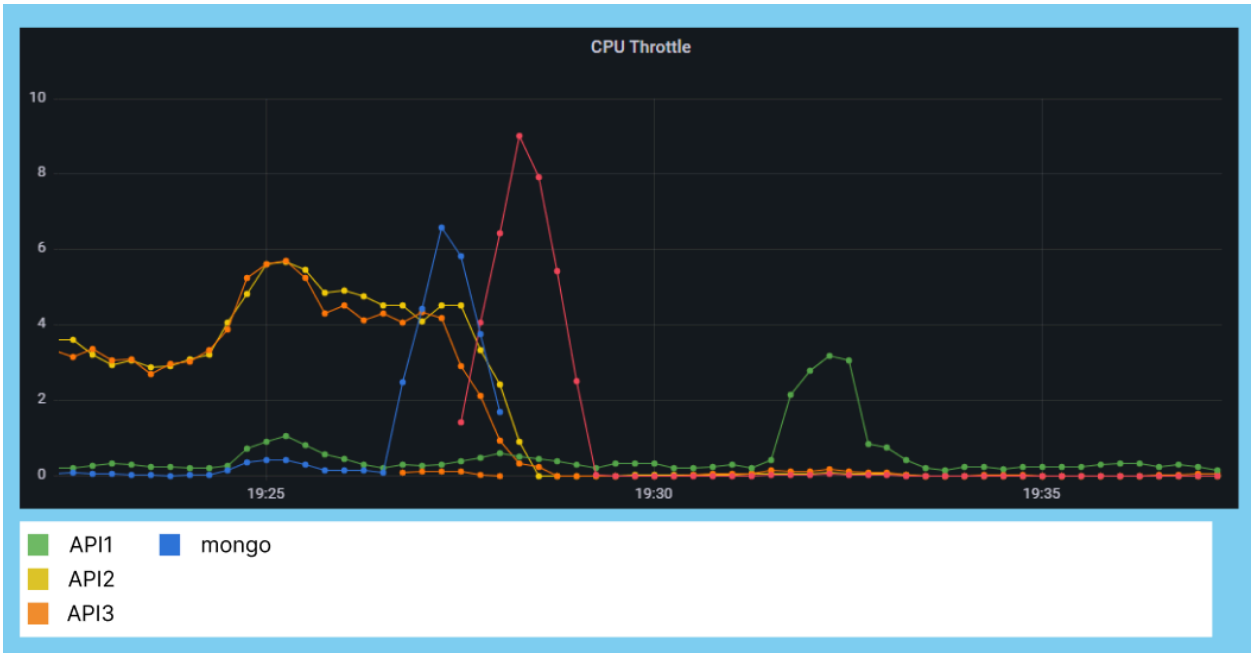


Figure 76: CPU Throttle for each docker service during Experiment 2.4 across time

Figure 77: Memory usage for each container (MiB) during Experiment 2.4 across time

Network Output

Shown in figure 78, network output drops to zero for all services other than API1 when the mongo container crashes, mirroring the previous figures for this experiment. It can be seen that a new container is starting for mongo in purple however its network output remains zero.

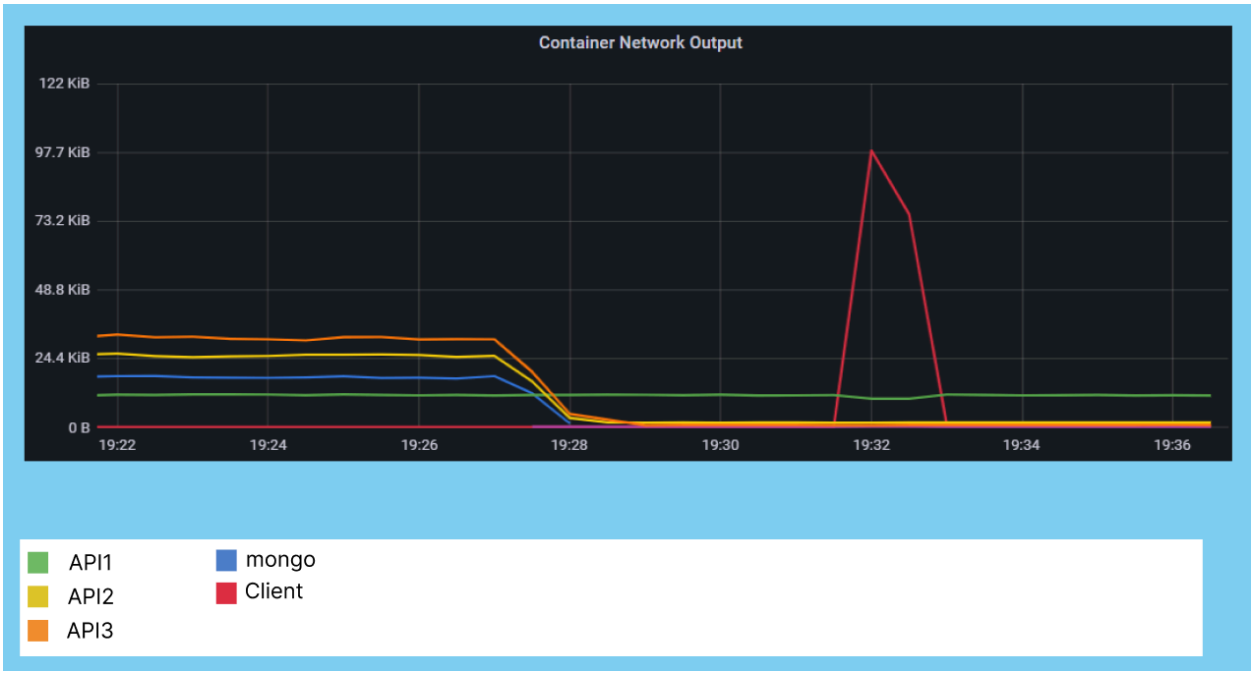


Figure 78: Network Input for containers (KiB) during Experiment 2.4 across time

Requests per second

In the majority of previous experiments, request rates from Locust have remained constant across attacks, however it seems the memory attack on the mongo container led to some disruption.

As seen in figure 79, a short time after the attack begins, when the mongo container crashes, the request rates to API2 and API3 endpoints sharply drop to zero before quickly returning to steady state levels regarding API2, however API3 requests per second only climb back up to around 15 which is about half of steady state levels which is unexpected as I predicted this would not be affected as these requests come from the locust container.

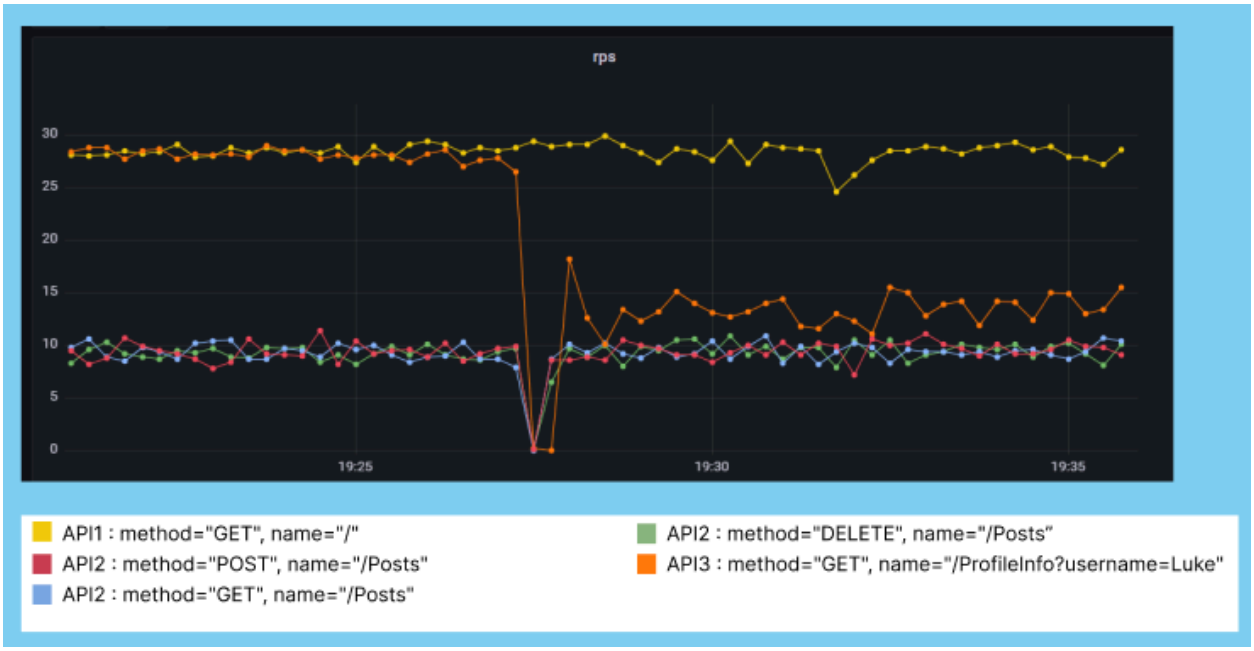


Figure 79: Requests per second shown for each endpoint with 300 users spawned for Experiment 2.4 over time

Response times and failure rate

Shown in figure 80, shortly after the memory attack began on the mongo container, API2 response times briefly spiked to around 75ms higher than steady state before slowly declining. It's shown that the API3 endpoint response time spiked quickly from steady state around 50ms up to around 300ms and then continually climbed for the next 7 minutes up to 1400ms at which point I decided to stop the experiment as the trend seemed constant.

Regarding failure rate as seen in figure 81 soon after the attack began, failure rates for all API2 and API3 endpoints rose dramatically. API2 endpoint failure rates all rose to around 10 per second indicating 100% failure rate. API3 failure rate rose to around 15 per second which is roughly the request per second rate after the attack discussed in figure 79 suggesting all requests are failing.

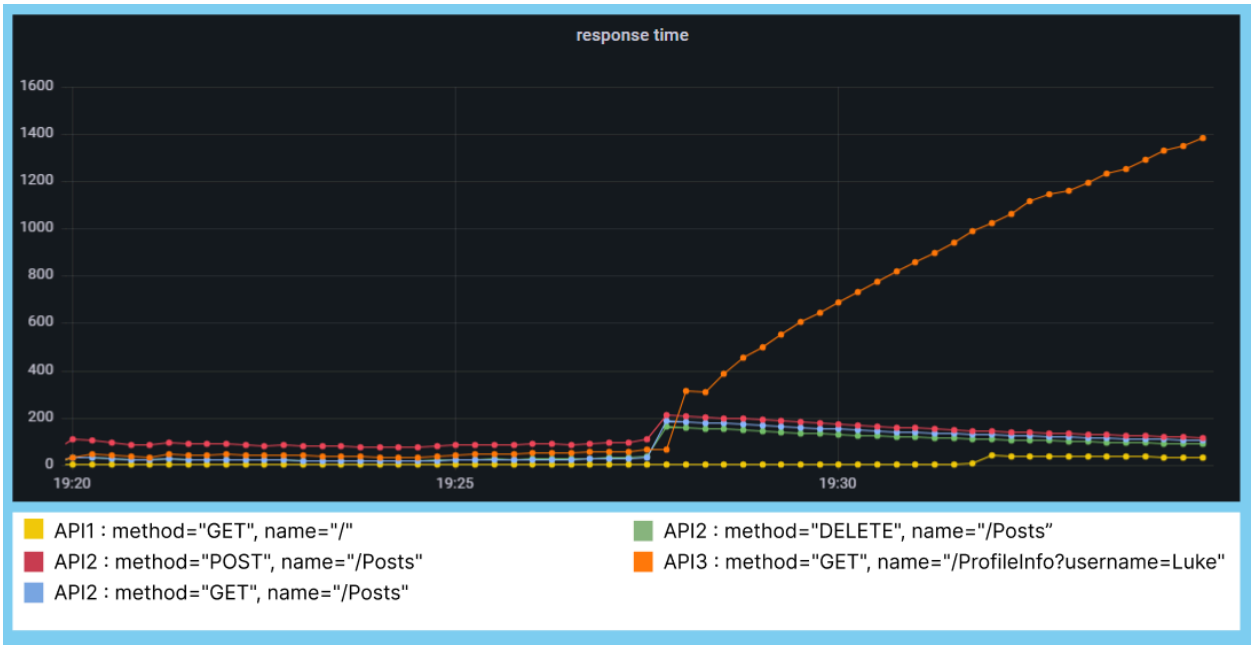


Figure 80: Average response times (ms) for endpoints during Experiment 2.4 across time

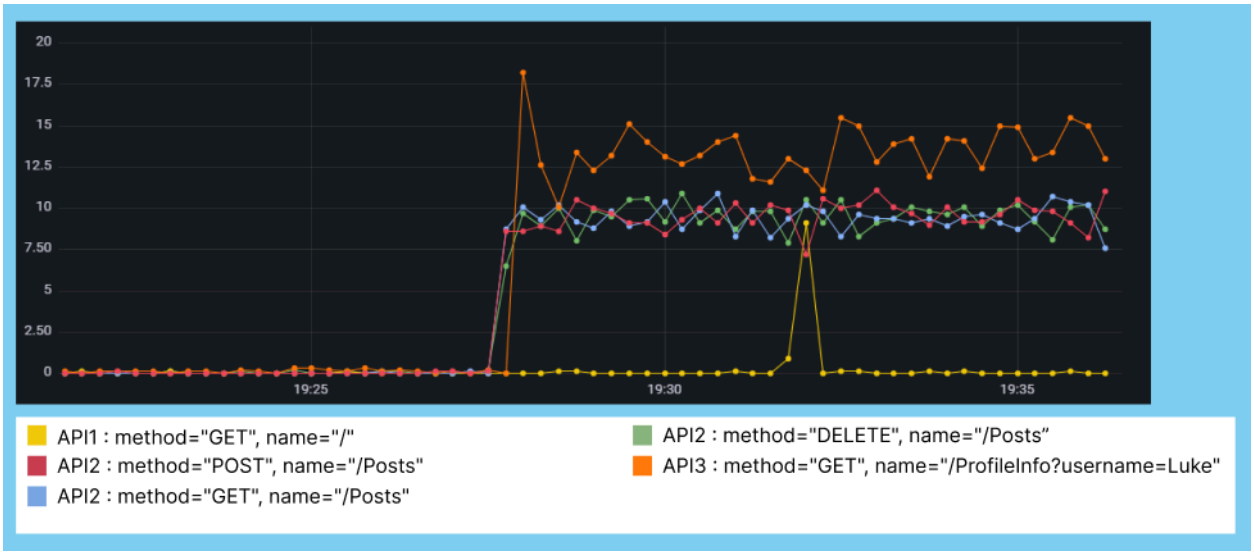


Figure 81: Error rate for endpoints (per second) during Experiment 2.4 across time

Conclusion

Reflection on initial objectives and deviations

Regarding the first objective involving building awareness of tooling and approaches I believe this was reasonably met. This was mainly addressed in background research. I outlined the effectiveness of Gremlin as a chaos engineering platform as well as mentioning pumba. I detailed chaos engineering approaches implemented in other papers and organizations. Before discussing chaos engineering I went into some detail about causes and repercussions of emergent failures and the ways one could monitor them.

There is probably more research that can be done regarding these topics however there's only so much I can explore in the time I have. Regarding 'becoming aware of tooling', some of this objective was met through the use and setup of Gremlin. I gained a good amount of knowledge about monitoring and load injection tools I used throughout my project and more general tools such as Docker, node, express and React whilst building the system.

For the second objective regarding building the system I believe this was somewhat met. Three separate backend APIs were completed which interacted with a mongo database. I got decent results and some interesting emergent failures and patterns arose when testing my system, however I had issues testing some API1 endpoints as discussed in the results.

The time spent on the front end could have been spent better. I didn't appreciate that most of the front end would not end up being tested in the same way the backend APIs were and the code I wrote for instance, displaying user information and popup editors, as well as error displays when microservices ceased responding. It seems this somewhat faded into irrelevance as my results consisted of tests on the backend services instead with the client service only ever really fluctuating in CPU usage when opening the application in the browser.

I think this project could have been done effectively using no front end, I could have gained more valuable insights if I had used this time building more complex interactions between backend APIs. I think some inefficiencies are inevitable when exploring a new topic as it is not till the end that I can formulate better ways to carry it out. I think I could have done a better job including more endpoints in my locust test files especially regarding API1 as it only ever had a basic empty get response.

I think some of this came down to wanting to start gathering results, as I didn't want to risk not completing that stage. Setting up monitoring was a larger task than I anticipated and sourcing the most appropriate tools whilst setting them up to work in harmony took a substantially long time however I learned a lot about monitoring and how to query meaningful metrics about my system, which is an important part of chaos engineering.

My other objectives included the use of overloading tools (Locust) and chaos engineering tools (Gremlin). I fulfilled my objective to use Docker constraints on my services and went further in some experiments, investigating the effects this had on attacks. I also used Cadvisor and went further using Prometheus and Grafana to monitor my system which I didn't fully appreciate when writing objectives. I also mentioned Docker visualiser in my objectives, I found that this wasn't really useful once my Grafana dashboard was set up as I could tell from this when containers went down anyway.

I set an objective to carry out 2 overloading experiments with Locust. I did carry out 2 preliminary investigations with Locust investigating API interactions however this is not necessarily an overloading test per se. I realized this would be more appropriate to do once Gremlin was set up which I did later in the project.

Regarding aims for the chaos engineering experiments themselves I believe I met my objectives mostly. I carried out a large amount of experiments, however many of them were quite simple and did not yield any unexpected results. However I think there were enough to grasp some fundamental issues that could arise in a locally deployed system like mine. In order to grasp more complex failure causes I would have to do an investigation into an inherently more complex system.

I ran into many issues when attempting to carry out network attacks which was one of my original objectives. I tried for a long time to fix these issues but there were some quite fundamental obstacles. The Gremlin agent makes use of netem which is a linux module for testing protocols by emulating things like latency, packet loss and reordering. I tried for a while to manually install this module in the containers I was attacking however eventually realizing that the Windows subsystem for Linux which Docker Desktop builds on does not support netem. The only potential fix I found for this would be to recompile the kernel enabling tc however by this point time constraints became an issue. Another fix would be to run the system on a Linux based host OS, this could be something to be implemented in future work as Network attacks are an important component of chaos engineering attacks.

Reflection on learning

Initially most learning included technical use of Docker as well as creating node/express API endpoints.

Coding wise this mainly consisted of using online resources and documentation to write code that accepted requests, interacted with the mongoose library in different ways to store and read data, format this and send it back as a response. I had some experience with node and express before this project so it wasn't completely new but there was still much I had to refresh myself on and learn. I also had to learn how to write appropriate database schema for mongo. A decent amount of the learning curve during this was the appropriate use of promises/asynchronous programming, catch statements and API programming principles.

I gained more experience in React through the front end development, increasing my competency in the use of states, props, conditional rendering and events. I gained more experience in the use of third party libraries such as material ui as well.

A substantial part of learning regarded the use of Docker. To get my system working I containerised all the components. This involved writing dockerfiles for each of the services I built to copy code to the container, install packages and run commands etc. It involved writing up docker compose files to expose ports, coordinate volumes, environments, dependencies and deployment settings. It also took a decent amount of research and time to set up all my monitoring systems (cadvisor, prometheus, grafana) as they all had to interact and scrape metrics from each other, this was mainly through reading online blogs and documentation in order to correctly write the docker compose file and configuration files.

I learned about some of the important different metrics involved in monitoring distributed web systems and how to query them in Prometheus, as well as setting up Grafana dashboards to display a collage of useful metrics in graph format in real time.

I learned how to set up and use Locust to load endpoints by writing tasks in a Locust file and ways to simulate real user behavior.

Finally I learned how to set up and use Gremlin to carry out attacks on my system, giving me some competence with this tool.

I also think there is something to be said for piecing together a set of tools and a methodology in order to inject load, carry out attacks and monitor my system. I could not find an explicit guide for how to do all this so I had to come up with a system myself which took some time however I think this has increased my knowledge substantially regarding chaos engineering tools and approaches, monitoring techniques and tools and CPU, memory and network interactions regarding a distributed backend web system even If my methodology has weaknesses.

Future work and weaknesses

A natural step for extending this project would involve deploying the system across multiple nodes. This could be done using AWS or Azure cloud systems and would better mirror how most systems in production are deployed, potentially leading to more meaningful results. It would also lead to meaningful results if experiments were done in a real system in production with real users however this may be difficult to do in practice as I would need access to a company system.

Now that I have the foundation for the tools and approaches one can take for carrying out chaos experiments, it would be useful to create a more rigorous testing system. Each experiment

could potentially be carried out multiple times to ensure anomalous results could be identified. It would also be interesting to carry out experiments but with a greater variety of attack settings. Most of my attacks were carried out for 2 minutes with 100% intensity. It would be interesting to see the effects of carrying out the attacks for longer periods of time, for example a 30 minute attack. I could also see if less intense attacks were handled differently. I would also like to carry out both Network attacks and Disk I/O attacks which I have not covered in this project.

Total word count: 15447
Bibliography word count: 451

Calculated word count: **14996**

Bibliography

1. Rosenthal, C. and Jones, N. (2020). *Chaos engineering : system resiliency in practice*. Beijing O'reilly® April.
2. Basiri, A., Behnam, N., de Rooij, R., Hochstein, L., Kosewski, L., Reynolds, J. and Rosenthal, C. (2016). Chaos Engineering. *IEEE Software*, [online] 33(3), pp.35–41. Available at: https://ieeexplore.ieee.org/abstract/document/7436642/ [Accessed 10 Mar. 2020].

3. Garraghan, P., Ouyang, X., Yang, R., McKee, D. and Xu, J. (2019). Straggler Root-Cause and Impact Analysis for Massive-scale Virtualized Cloud Datacenters. *IEEE Transactions on Services Computing*, [online] 12(1), pp.91–104. Available at: https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=7572191&casa_token=H0PNSPpxtTEAAAAA:6FH5MX5jT4Z9Kh64_vR1tHq2d4NTxHL0SvAmEIOxPelTbdR47VI0hO1H7Qj-eEadM1I5zOI&tag=1 [Accessed 30 Mar. 2022].
4. Garraghan, P., Yang, R., Wen, Z., Romanovsky, A., Xu, J., Buyya, R. and Ranjan, R. (2018). Emergent Failures: Rethinking Cloud Reliability at Scale. *IEEE Cloud Computing*, 5(5), pp.12–21.
5. Gill, S.S., Ouyang, X. and Garraghan, P. (2020). Tails in the cloud: a survey and taxonomy of straggler management within large-scale cloud data centers. *The Journal of Supercomputing*, 76(12), pp.10050–10089.
6. www.gremlin.com. (n.d.). *4 Chaos Experiments to Start With*. [online] Available at: https://www.gremlin.com/community/tutorials/4-chaos-experiments-to-start-with/.
7. Mogul, J.C. (2006). Emergent (mis)behavior vs. complex software systems. *ACM SIGOPS Operating Systems Review*, 40(4), pp.293–304.
8. Jha, D.N., Garg, S., Jayaraman, P.P., Buyya, R., Li, Z. and Ranjan, R. (2018). A Holistic Evaluation of Docker Containers for Interfering Microservices. *2018 IEEE International Conference on Services Computing (SCC)*.
9. Heyman, H. (2013). *Examensarbete 15 hp Implementing dynamic allocation of user load in a distributed load testing framework*.
10. Natu, J., Ghosh, R. and Shyamsundar, R. (2016). *Holistic Performance Monitoring of Hybrid Clouds: Complexities and Future Directions*.
11. Casalicchio, E. and Perciballi, V., (2017). Measuring Docker Performance. *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*,.
12. Jernberg, H., Runeson, P. and Engström, E. (2020). Getting Started with Chaos Engineering - design of an implementation framework in practice. *Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*.

13. Heorhiadi, V., Rajagopalan, S., Jamjoom, H., Reiter, M.K. and Sekar, V. (2016). Gremlin: Systematic Resilience Testing of Microservices. *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*.
14. DevExchange, C.O. (2020). *Continuous Chaos - Introducing Chaos Engineering into DevOps Practices*. [online] Capital One Tech. Available at: https://medium.com/capital-one-tech/continuous-chaos-introducing-chaos-engineering-into-devops-practices-75757e1cca6d [Accessed 4 Apr. 2022].
15. prometheus.io. (n.d.). *Metric types | Prometheus*. [online] Available at: https://prometheus.io/docs/concepts/metric_types/.
16. Nielsen, J. (1993). *Usability Engineering*. Morgan Kaufmann.
17. GitHub. (2021). *Locust Exporter*. [online] Available at: https://github.com/ContainerSolutions/locust_exporter [Accessed 8 May 2022].
18. Deland-Han (2019). *Introduction to the page file*. [online] Microsoft.com. Available at: https://docs.microsoft.com/en-us/windows/client-management/introduction-page-file.