Sys_Perf_Check : A tool to diagnose bottleneck component in a back end server

Progress Seminar - 2
Submitted in partial fulfillment of
the requirements for the degree of
Master of Science by Research
by

Jatin Lachhwani (21q050005)

Guide: Prof. Bhaskaran Raman

Department of Computer Science
Indian Institute of Technology Bombay
Mumbai 400076 (India)

Acknowledgements

I'm thankful to **Prof. Bhaskaran Raman** for guiding me throughout the thesis and providing valuable feedback. I would also like to thank **Debajyoti Saha** for actively participating with me in the thesis and **Kashmira Nagwaker** for her support with the safe server.

Jatin Lachhwani
IIT Bombay
December 1, 2022

Declaration

I declare that this written submission represents my ideas in my own words and where others' ideas or words have been included, I have adequately cited and referenced the original sources. I declare that I have properly and accurately acknowledged all sources used in the production of this report. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in my submission. I understand that any violation of the above will be a cause for disciplinary action by the Institute and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been taken when needed.

Jatin Lachhwani
21Q050005

Date: May 26, 2023

Contents

List of Figures

Abstract

The development of modern web services involves a division into two halves: front end and back end. The back end is typically responsible for complex processing and is composed of various components arranged in a stack. However, it is common for ill-configured components to go unnoticed until performance issues arise, significantly impacting user experience. Therefore, enhancing performance is crucial.
This project aims to address this challenge by introducing Sys_Perf_Check, a powerful tool for early detection of misconfigured components that can cause performance issues. Our tool systematically tests components under varying loads and analyzes their logs to identify potential bottlenecks. Specifically, we focus on response time as a key metric and observe any significant divergence in response time compared to preceding components.
By presenting the findings through informative graphs, we provide users with a clear and intuitive analysis, enabling easy identification of performance bottlenecks. Sys_Perf_Check offers a proactive approach to performance optimization, ensuring that issues are detected early and resolved efficiently, leading to an enhanced user experience for web services.

Chapter 1

Introduction

In the field of software development, identifying and addressing performance issues across the software stack is a crucial task. However, such issues often go unnoticed or are challenging to detect, resulting in degraded system performance and user experience. To tackle this problem, we have developed Sys_Perf_Check , a tool designed to detect and analyze bottleneck components in the software stack. Our aim is to provide developers and system administrators with an easy-to-use tool that can identify performance bottlenecks early on, allowing for timely optimization and improvement. In this report, we present an overview of Sys_Perf_Check and its application in the context of SAFE (Smart, Authenticated and Fast Exams), a smartphone application for conducting exams. By using Sys_Perf_Check to analyze the performance of SAFE's software stack, we demonstrate the effectiveness and utility of our tool in identifying and resolving performance issues. Through this project, we address the need for a comprehensive and efficient solution to performance analysis in software systems.

1.1 Background

1.1.1 Overview of SAFE

SAFE (Smart, Authenticated and Fast Exams) is a smartphone application that allows students to take exams on their phones. Students can bring their phones in place of examination and the question paper will be uploaded on the app inside their smartphones. It also provides additional features like marking attendance and live-time feed of student activity during the test. It eases the process of taking quizzes for students and uploading and evaluating quizzes for instructors. The idea of **Sys_Perf_Check** was conceived while doing performance testing of APIs used in SAFE. Its component stack is relevant in our discussion as we illustrate the usage of Sys_Perf_Check later.

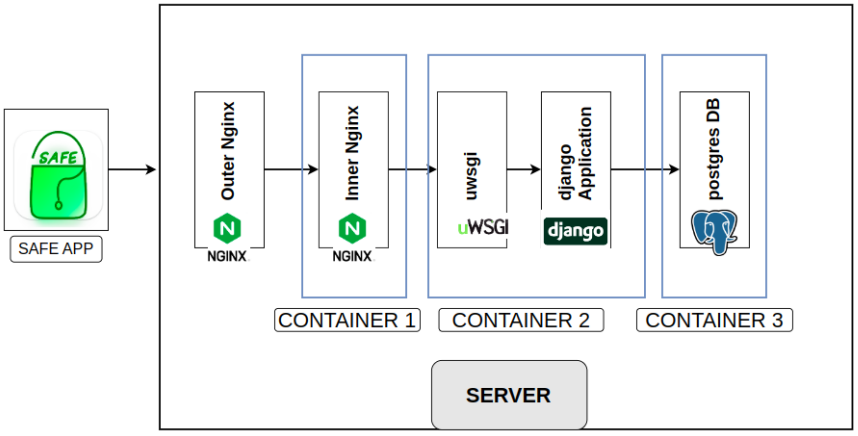

Figure 1.1: SAFE server architecture

Components of SAFE

1. **SAFE app**: The app which is used by students to attempt the quiz.
2. **Outer nginx**: The request by the user is sent to the outer nginx. It is generic, i.e., it is used to serve traffic for other web services hosted on the server. This acts as a reverse proxy.
3. **Inner nginx**: This nginx is specific to our application and is customized by us based upon our needs. It forwards traffic to uWSGI.
4. **uWSGI**: This is an implementation of the WSGI protocol used as a web server specific to python-based web frameworks.
5. **django**: The web framework used by us to write the application logic.
6. **postgres**: The database used in the SAFE project where most of the data generated in SAFE is stored.

1.1.2 Previous Work

APIs used when taking a quiz in SAFE

The core functionality of SAFE is its quiz functionality, as its goal is to move exams from paper to phone. In order to take quiz in SAFE, the instructor first publishes a quiz on SAFE using the website. Then there are 10 steps and corresponding APIs that are called while giving a quiz on SAFE till submission.

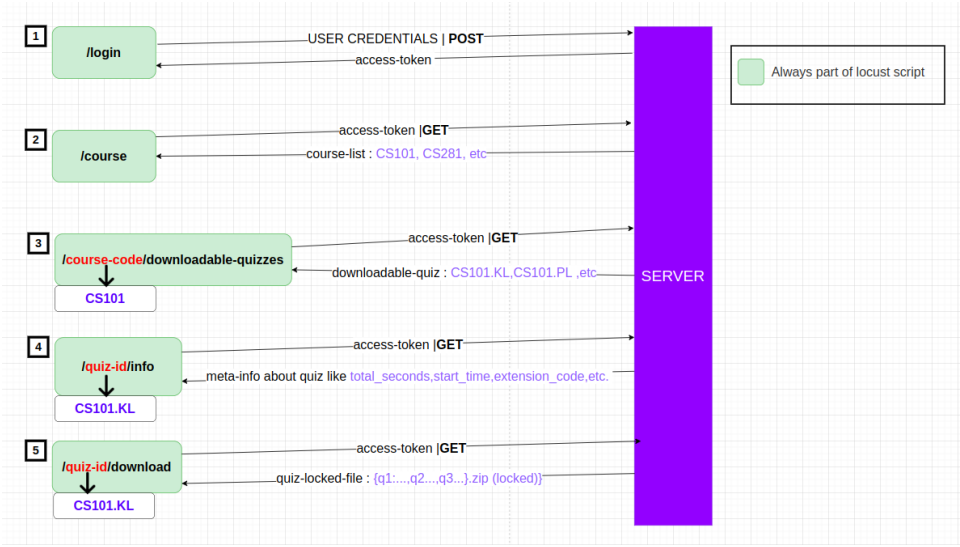

Figure 1.2: First five APIs

1. **Login** : The user provides its credentials to access the application.This generates a token which is used for further interaction. This step is performed only ones until the user logout. API - **/login**

2. **Fetch course List** : After login the application automatically fetches the list of courses user is enrolled in .API - **/course**

3. **Fetch downloadable quiz list** : Once the user taps on a particular course the list of downloadable quizzes is fetched from the server.API - **/course-code/downloadable-quizzes**

4. **Fetch quiz info** : This step involves fetching the meta data about the quiz like duration, quiz status,question id,etc.API - **/quiz-id/info**

5. **quiz download** : This step involves downloading the actual content of the quiz . API - **/quiz-id/download**

6. **quiz authenticate** : This step checks the status of quiz whether it is started by the instructor or not. API - **/quiz-id/authenticate**

7. **image upload** : In this step student uploads the solution images . It is performed every time a student clicks an image. API - **/quiz/upload_image**

8. **quiz partial submission** : This stores the answer the student has provided till a point in time and it runs periodically. API - **/quiz-id/partial_submission**

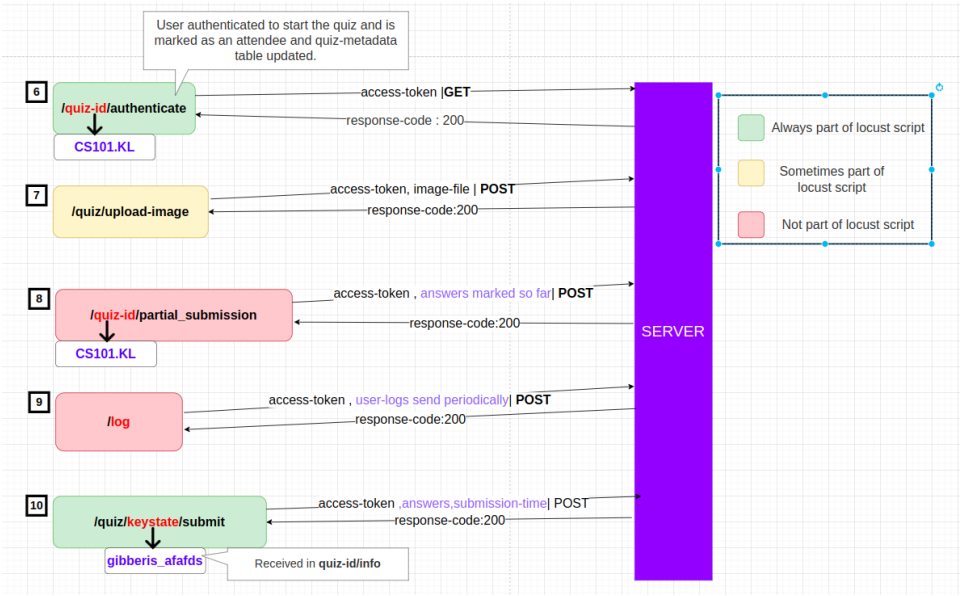

Figure 1.3: Last five APIs

9. **log** : This stores the log about the activity of student attending the quiz periodically. API - **/log**

10. **quiz submission** : This is the final step which marks the completion of the quiz .It is done by the student or automatically by the phone when time is over. API - **/quiz/keystate/submit**

Searching for bottleneck APIs and Component

We were trying to find bottleneck APIs in SAFE related to its quiz functionality. In order to do so, we used scripts written in locust[4] to simulate the quiz functionality and measure the response time of these APIs to identify any bottleneck APIs. We found that in the test where we simulated 100 users, the average response time of APIs was approximately 5.73 seconds as shown in figure 1.4, and also the CPU utilization of the server machine was nowhere close to 100 This led us to conclude that there might be a bottleneck component in our server stack. We analyzed the logs generated by different components in our stack and found uWSGI was not correctly configured, due to which the APIs were performing poorly. After correctly configuring it, the average response time of all APIs dropped to 0.68 seconds as shown in figure 1.5.

Type	Name	# Requests	# Fails	Median (ms)	90%ile (ms)	99%ile (ms)	Average (ms)
POST	1.login	100	0	1400	11000	12000	4279
GET	2.course_list	100	0	4100	10000	11000	4385
GET	3.quiz_list	100	0	3800	8000	11000	4278
GET	4.quiz_info	100	0	9300	15000	18000	9467
GET	5.quiz_download	100	0	7200	13000	16000	7766
GET	6.quiz_authenticate	100	0	3800	9000	17000	4278
POST	7.quiz_submit	100	0	3300	15000	17000	5704
	Aggregated	**700**	**0**	**5300**	**12000**	**17000**	**5737**

Figure 1.4: Performance result in ill-configured uWSGI

Type	Name	# Requests	# Fails	Median (ms)	90%ile (ms)	99%ile (ms)	Average (ms)
POST	1.login	103	0	900	6100	7100	1968
GET	2.course_list	103	0	63	120	6000	438
GET	3.quiz_list	103	0	46	90	6100	519
GET	4.quiz_info	103	0	110	220	5000	306
GET	5.quiz_download	103	0	44	72	5600	325
GET	6.quiz_authenticate	103	0	100	270	6300	465
POST	7.quiz_submit	103	0	690	1100	1500	783
	Aggregated	**721**	**0**	**95**	**1300**	**6500**	**686**

Figure 1.5: Performance result in properly configured uWSGI

1.2 Problem Statement

From the work we did on performance analysis, mentioned in the previous section, we realized that performance issues spread across the software stack can go unnoticed for too long and are also hard to detect. So, we decided to pursue a new project in which we would create a tool that will make it easier for us to detect such issues easily and early on. Thus, we decided to create a tool with the following objectives:

- Detect bottleneck components in the software stack (similar to SAFE).
- Independent of application-specific APIs.
- Easy to use and configure.
- Provides visual feedback for the analysis of components in the software stack.

Since our tool allows us to detect performance issues across the software system stack, we have named it Sys_Perf_Check .

1.3 Scope of Sys_Perf_Check

Sys_Perf_Check can be used in web applications that have an architecture similar to SAFE. Applications that have a linear stack, e.g., nginx-uWSGI-django. It can be used to analyze the configuration and performance issues across the stack. It uses locust [6] scripts for performance testing. In general, it can work in stacks where requests are visible across all the components and the components are capable of logging response time of the requests without a significant impact on performance.

1.4 Report Structure

From the next chapter onwards, we will describe the general architecture of Sys_Perf_Check in chapter 2. Then we will get into the implementation details of Sys_Perf_Check in chapter 3. After that, we will demonstrate the use of Sys_Perf_Check on SAFE for performance analysis and tuning in chapter 4. We will also see the results generated by that can be used for analysis. We will end it with a succinct conclusion and future work in chapter 5.

Chapter 2

Sys_Perf_Check overview

2.1 Key Features of Sys_Perf_Check

- **Bottleneck Detection:** Sys_Perf_Check is designed to detect bottleneck components in the software stack of web applications. By analyzing the response time of each component and identifying divergences, it helps pinpoint the components causing performance issues.
- **Early Detection:** The tool enables early detection of misconfigured components that may lead to performance degradation. By identifying these issues at an early stage, developers and system administrators can take timely measures to optimize and improve system performance.
- **Independent of Application-specific APIs:** Sys_Perf_Check is designed to be independent of application-specific APIs. It focuses on analyzing the performance of the underlying components in the software stack, regardless of the specific APIs used in the application.
- **Ease of Use and Configuration:** The tool is designed to be user-friendly and easy to configure. Developers and system administrators can easily integrate Sys_Perf_Check into their existing software stack and configure it to analyze the performance of their components.
- **Visual Analysis:** Sys_Perf_Check provides visual feedback in the form of graphs, making it easier to analyze the performance of different components. The graphs generated by Sys_Perf_Check help visualize the response time of each component, enabling users to quickly identify bottleneck components.
- **Applicable to Linear Stack Architectures:** Sys_Perf_Check is applicable to web applications with linear stack architectures, such

as nginx-uWSGI-django. It works in stacks where requests are visible across all components and the components can log response times without significant performance impact.

- **Supports Performance Tuning:** By detecting bottleneck components and providing insights into their performance, Sys_Perf_Check facilitates performance tuning. It enables developers and system administrators to optimize the performance of their software stack by identifying and resolving performance issues.
- **Flexible and Extensible:** Sys_Perf_Check is designed to be flexible and extensible, allowing users to adapt it to their specific needs. It provides customizable settings and options to fine-tune the performance analysis process according to the requirements of the application and infrastructure.

These key features make Sys_Perf_Check a valuable tool for developers and system administrators in identifying and addressing performance issues in web applications, ultimately leading to improved system performance and user satisfaction.

2.2 Sys_Perf_Check Architecture

2.2.1 Introduction

When we started composing the tool, we only created a single script which would generate load for varying number of users and would also assign an id to the test being conducted. After the test was over we would go to the system being tested then we would extract the logs specific to a particular test from each component and then we would do the analysis.
Now one can make this easier by making sure that if the script and the system being tested are on the same physical server then there is no need of transferring logs across machines. Keeping our tool script and the server being tested makes each test non uniform. As for different loads the script would consume different amount of resources and thus the resources available for the server being tested would not be same for each test. So, we decided to write a script in such a way that could run it from a different machine. But still we required a means to easily fetch the generated logs from the machine where our server is hosted.
In order to automate log extraction from server being tested we decided to write a separate script that would allow us to easily extract the necessary logs from each component and thus make the extraction process easier.
We created our tool in two pieces a client end script that runs the performance test and display graphs for each test and server end script that would extract the logs from each component specific to a test and then send them

for analysis to client end script.
The two parts of our scripts :

1. **Client end script:** This contains the script that is run on the machine from which the server is being tested.

2. **Server end script:** This script runs on the server being tested.

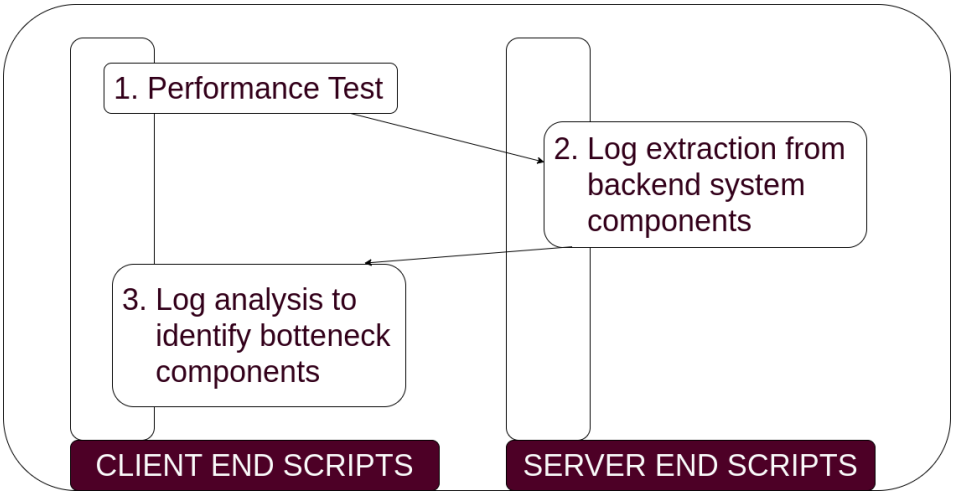

Figure 2.1: Tool Architecture

2.2.2 Purpose of each script

Client end script

The client end script has multiple responsibilities. It is responsible for giving an ID to the performance test. Based on settings, it generates the load of the desired number of users for a specific duration. Once the test is over, it sends a signal to the server end script to fetch the logs for the particular test. Then it generates a graph using the logs of each component, which is used to identify the bottleneck component in the stack.

Server end script

The server end script listens for requests from the client end script, which contains information about the test ID. Based on this ID, it extracts the logs of each component specific to that test. Then it compresses the logs and starts an FTP server, which is used to transfer these logs to the client end script.

2.3 Sys_Perf_Check Request endpoint

In order to use Sys_Perf_Check , one needs to register the web endpoint with the following pattern:
https://xyz.com/**sys_perf_check/[test-id]/[num-users]/**

The above-mentioned pattern needs to be registered to ensure that each test can be uniquely named by the client end script, which is used later to identify the logs corresponding to that test. The number of users is inserted by the client end script corresponding to different user loads to identify the response time corresponding to a specific user load within a test. It is assumed that the request is visible to all the components being tested, allowing them to push their names into the logs. Details about the contents of the endpoint are in Chapter 3, and information on how to register the web endpoint is in Chapter 4.

Chapter 3

Sys_Perf_Check implementation details

3.1 The Request end point

The end point https://xyz.com/**sys_perf_check/[test-id]/[num-users]**/ needs to be added as a part of the system stack being tested. We have defined it like this because it makes it easy for us to identify the test id and number of users for which this test was run.
Now, we will see the contents of the end point. The end point we have defined contains a loop that runs for a specific period of time then returns a string containing the response time. In our case we have kept it 10 ms. This allows us to bound the execution time for the last component in our stack (in our case it is django). This makes it easy for us to estimate the execution time of the final code. The code is shown in 3.1 Once the above

```
def sys_check_fun():
    y=0
    x=timeit.default_timer()
    while timeit.default_timer() - x < 0.01:
        y=y+1
```

Figure 3.1: Request end point code

mentioned end point is registered it is now possible to use Sys_Perf_Check for testing and analysis.

3.2 Client end script

The client end script contains the part to do the following things

- Takes necessary command line arguments
- Generate test-id and test-id folder
- Run the locust script for varying load
- Share test info and fetch test specific logs
- Extract the info from logs and display as results

3.2.1 Takes the necessary command line arguments

The client end script is passed the following arguments -l : lower bound of test users , -u : upper bound of test users, -s : step size to increase users , -t : duration of test for each test. The code uses the **argparse module**[1] in python3 to accomplish this task. The load one needs to generate is specific to their application backend setup. Thus, they need to determine it empirically. We provide a simple character user interface to do so.

```
λ › ./client_end_script.py -h                                          thesis/locust_fi
usage: ./client_end_script.py [-h] -l LOWER_BOUND_USERS -u UPPER_BOUND_USERS -s STEP_SIZE
                              [-t RUN_TIME]

To figure out system level bottleneck component for REST API based services

optional arguments:
  -h, --help            show this help message and exit
  -l LOWER_BOUND_USERS  Specify the lower bound of the number of users.
  -u UPPER_BOUND_USERS  Specify the upper bound of the number of users.
  -s STEP_SIZE          Specify the step size for incrementing the number of users.
  -t RUN_TIME           Specify the runtime for each user number being tested
```

Figure 3.2: client end script : arguments

3.2.2 Generate test-id and test-id folder

To assign each test a unique id we have used **MetroHash64** a non-cryptographic hash function that efficiently computes a 64 bit random value which we use as the id for the particular test.For this purpose we have used a public library metrohash[3]. This id is used to filter the logs generated at each component. It is given the current time and date in byte string format as input. It also generates a folder by the same name where the logs specific to this test are placed for analysis.

Reasons for using hashing

- **Unique Identification**: Hash functions generate a unique hash value for each input. It allows us to assign a unique ID to each test based on the current time and date. This ensures that each test is identified uniquely and can be easily distinguished from others.

- **Filtering and Analysis**: Using the generated hash IDs allows you to filter and analyze logs at each component easily. By associating a specific folder with each test ID, you can store the logs specific to that test separately. This organization makes it simpler to track and analyze the logs for individual tests without mixing them up with logs from other tests.

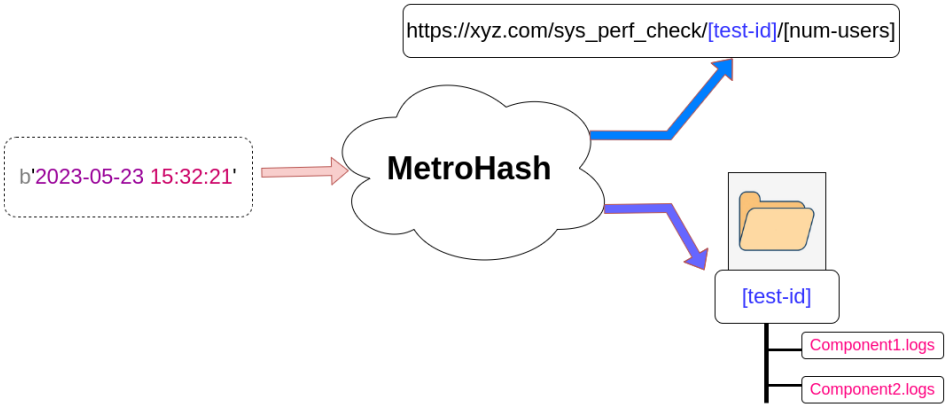

Figure 3.3: MetroHash Use

3.2.3 Run the locust script for varying load

Locust script is run in headless mode (as a command line tool) to generate the load . It is run by the client end script . It takes the arguments number of users, spawn rate and duration for each run. The locust script run multiple times depending upon the command line arguments passed in the step 3.2.1. In the locust script host address needs to be configured before starting the client end script.

3.2.4 Share test info with server end script and fetch test specific logs

The server end script is responsible for extracting the test specific logs. In order to do so it needs info about the name of the test and a rough estimate of number of lines in the log. This info is passed to server end script by the client end script before log extraction. Once the logs are extracted by the server end script it notifies the client end script . Then the client end script downloads it from the server end script using ftp. This is then used for analysis.

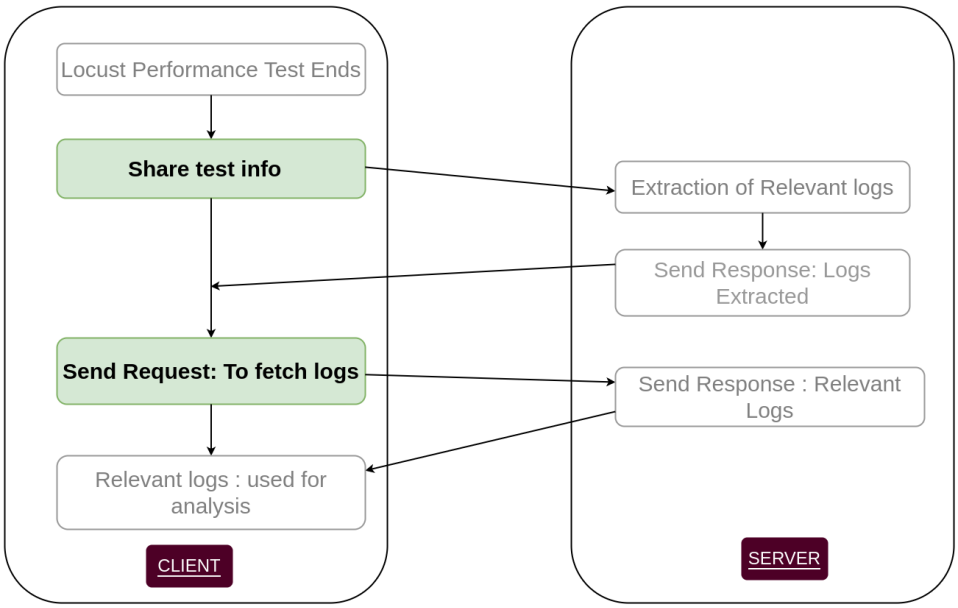

Figure 3.4: Share test info and fetch test specific logs

3.2.5 Extract the info from logs and display the results

Once the logs are received from the server end script . The information in them is extracted to find the response time information.
There is a file called components.json, it contains information about each component. To the client end script, unit of time each by each component is relevant info. More details about components.json are in the section3.3. Once the necessary information is extracted it is used to generate the relevant graphs which can be used to find the bottleneck component. A graphs is shown in figure 3.5. This picture corresponds to an old test. We have a GUI now which will be shown in chapter 4.

3.3 Server end script

The server end script contains the part to do the following things

- Extracting relevant logs
- Creating an ftp server for transferring logs

3.3.1 Extracting relevant logs

In order to extract logs the server needs to know the test id which is shared by the client end script once it has completed the test. Also it needs a rough idea about the number of lines it need to look in the logs which is also shared

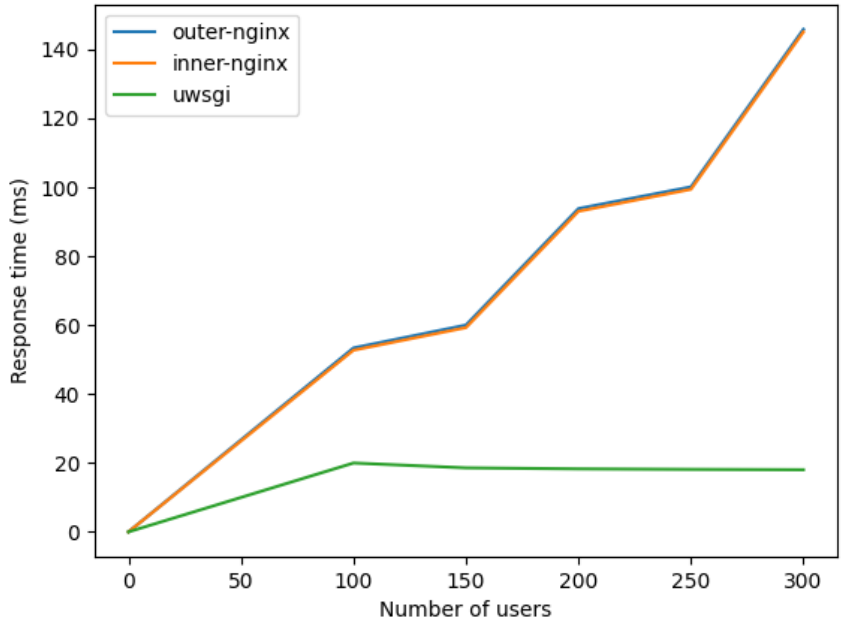

Figure 3.5: Graph generated from a test

by the client end script.
To know the location of logs of each component and the name of each component a file components.json is used . An example of its content is shown in fig3.6. Even after knowing the location of the log files it is not practical

```
[
    {
        "componentName":"outer-nginx",
        "logPath":"/var/log/nginx/safev2.cse-proxy-access.log",
        "timeUnit":"s"
    },
    {
        "componentName":"inner-nginx",
        "logPath":"/home/safev2admin/kashmira/safe_server_v2/logs/inner-nginx/safe-access.log",
        "timeUnit":"s"
    },
    {
        "componentName":"uwsgi",
        "logPath":"/home/safev2admin/kashmira/safe_server_v2/logs/uwsgi/uwsgi-daemonize.log",
        "timeUnit":"ms"
    }
]
```

Figure 3.6: Example of components.json

to send the whole file as it is bulky. So in order to get the relevant parts from the logs tail and grep are used in combination, tail command is used

to access the last few lines based upon the test and then grep is used to filter the records specific to the test. This is possible because of the structure of the request **/sys_perf_check/[test-id]/[num-users]** which contains the test-id which makes it possible to uniquely identify the relevant parts from the logs. Then based upon the name of the component and the test the file names are given to the component of each logs. once all the logs are extracted they are compressed into a single tar file placed in the public folder which will be available through ftp.

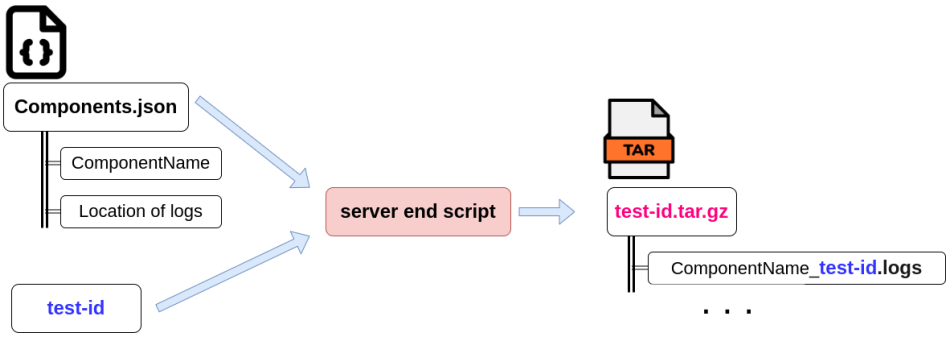

Figure 3.7: server end script extracting logs

3.3.2 Creating an ftp server for transferring logs

The logs extracted in the above step need to be shared with the client machine. In order to do so the server end script call the fork a new process which creates an ftp server in a directory name public which contains the tar file containing all the logs . These logs are then fetched by the clent end script . In order to create ftp server we have used ftplib[2] internally (python module).

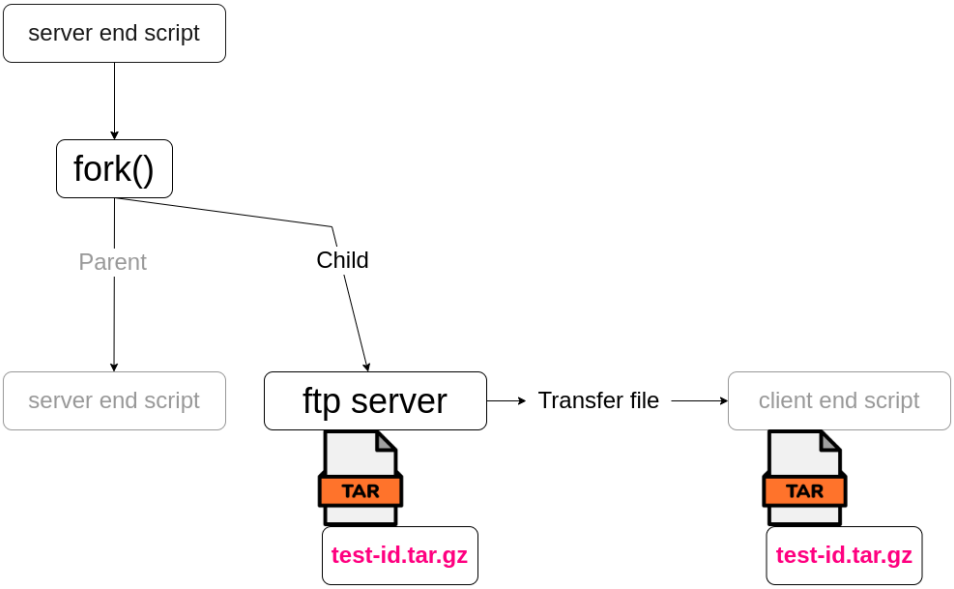

Figure 3.8: using ftp to transfer logs

Reasons for chosing ftp

We have chosen ftp for file transfer due to following reasons :

- **Ease of Implementation:** The ftplib module in Python provides a convenient and straightforward way to implement an FTP server and client. It abstracts away the complexities of the FTP protocol, allowing developers to focus on the file transfer logic.
- **Efficiency:** FTP is designed for efficient file transfer, making it suitable for transferring large files or a collection of files, such as the tar file containing logs in this scenario.

Chapter 4

Sys_Perf_Check usage and Results

4.1 Introduction

In this section, we will illustrate the use of our scripts through an example on SAFE. Also, we will see how it can be used to diagnose bottlenecks and tune the performance by analyzing the graphs displayed by it. The components in SAFE are mentioned in subsection 1.1.1.

4.2 Initial setup

4.2.1 Sys_Perf_Check endpoint

It is essential to set up Sys_Perf_Check endpoint in order to use the tool. Also, the request name should be visible in the logs of each component. In our case, we have registered it in Django with the following format: /sys_perf_check/[test-id]/[num-users]/. Also, make sure to turn on the logs at each component. The content of the endpoint is mentioned in section 3.1.

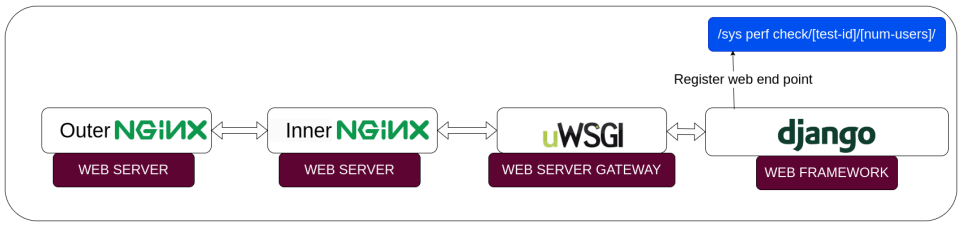

Figure 4.1: create Sys_Perf_Check web endpoint

Special log format

Make sure that the logs generated follow the special format such that the response time of the request is wrapped in ***[response-time]***. This allows us to identify the response time uniquely in each test.

```
[25/May/2023:10:12:13 +0000] GET /sys_perf_check/ba921a6be74ff7a3/100/ HTTP/1.0***0.027***
[25/May/2023:10:12:13 +0000] GET /sys_perf_check/ba921a6be74ff7a3/100/ HTTP/1.0***0.029***
[25/May/2023:10:12:13 +0000] GET /sys_perf_check/ba921a6be74ff7a3/100/ HTTP/1.0***0.034***
```

Figure 4.2: Example log format

4.2.2 Configuration file for components

The file **components.json** is required by both the client-end script and server-end script. The server-end script uses it for the component name and log path. The client-end script uses it for the component name and unit of time in the logs. It contains an array of JSON objects where each object contains the fields **componentName, logPath, timeUnit**. An example JSON object is shown in figure 4.3.

```
{
    "componentName": "outer-nginx",
    "logPath": "/var/log/nginx/safev2.cse-proxy-access.log",
    "timeUnit": "s"
}
```

Figure 4.3: Example JSON object

4.3 Running the main scripts

4.3.1 Server-end script

```
λ › ls
components.json  server_end_script.py
λ › ./server_end_script.py
```

Figure 4.4: Server-end script running and ensuring components.json in the same folder

Make sure to run the server-end script before running the client-end script; otherwise, the client-end script won't work. Run it on the machine

where the software being tested is deployed. Make sure to place the components.json file in the same folder as the server-end script. The server-end script uses two ports: 5000 and 5001. One for transmitting messages with the client-end script (5000) and the other for hosting the FTP server (5001). Thus, make sure the two ports are available.

4.3.2 Client-end script

While running the client-end script, make sure the components.json file is also present in the same folder. Also, it takes command-line arguments with the following flags:

- -l: lower bound of users
- -u: upper bound of users
- -s: step size
- -t: duration for each test for that number of users.

```
λ › ./client_end_script.py -l 10 -u 30 -s 10 -t 30
```

Figure 4.5: Running client-end script

e.g.: Running the command in figure 4.5. This will create 3 tests with the number of users 10, 20, and 30. Each test will run for 30 seconds. Once the tests are complete, the client-end script will fetch the logs and show a graph displaying the results of each test. A graph is shown in figure 4.6.

Image and Table Presentation

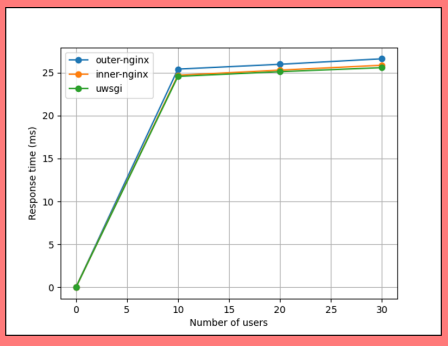

Table containing Response Time(ms) vs number of users

Numusers	outer-nginx_time(ms)	inner-nginx_time(ms)	uwsgi_time(ms)
10	25.41	24.69	24.56
20	25.97	25.29	25.11
30	26.61	25.86	25.58

Figure 4.6: Graph displaying test results at the end of the client-end script

4.4 Using Sys_Perf_Check on SAFE to find bottleneck components

4.4.1 Introduction

We will be conducting our testing on a machine with the configurations shown in table 4.1. We will try out three scenarios where we will be modifying the configuration of uWSGI to adjust the performance, which will in turn impact the performance of the whole system. We will use Sys_Perf_Check to identify the bottleneck and evaluate the system's performance under different configurations.

Parameter	Value
RAM	128 GB
Architecture	x86_64
No of CPU	24
Model name	Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20GHz
Cores per socket	12
Thread per core	2

Table 4.1: **Machine configuration**

4.4.2 Different configurations of uWSGI

We will modify a configuration parameter called "Processes" in the uWSGI configuration file. We will try four different values: significantly lower than the ideal configuration, equal to the ideal configuration, slightly higher than the ideal configuration, and significantly higher than the ideal configuration. The ideal configuration for uWSGI processes is 2 times the number of CPU cores [5], which in our case is 48. We will test the system for the following four cases: 5, 24, 48, and 64 processes and observe how Sys_Perf_Check can help us analyze the system's performance.

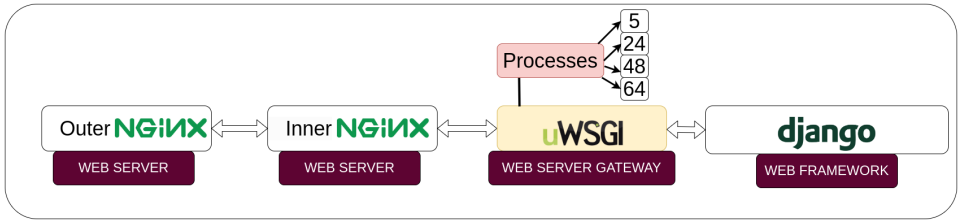

Figure 4.7: Different configurations of uWSGI

4.4.3 Results for different uWSGI configurations using Sys_Perf_Check

uWSGI with Processes: 5

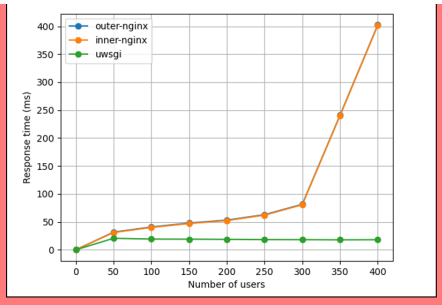

Table containing Response Time(ms) vs number of users

Numusers	outer-nginx_time(ms)	inner-nginx_time(ms)	uwsgi_time(ms)
50	31.61	30.91	20.65
100	40.82	40.03	19.19
150	47.91	47.03	19.03
200	53.18	52.35	18.72
250	62.88	62.08	18.27
300	81.48	80.67	18.15
350	240.83	240.24	17.86
400	402.55	401.92	18.04

Figure 4.8: Graph of Sys_Perf_Check for uWSGI Processes 5

uWSGI with Processes: 24

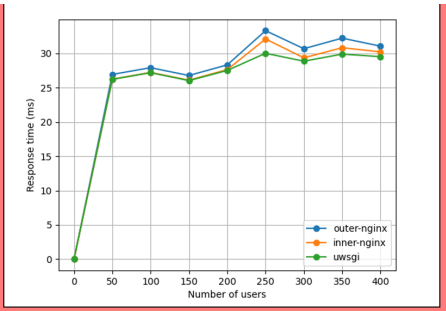

Table containing Response Time(ms) vs number of users

Numusers	outer-nginx_time(ms)	inner-nginx_time(ms)	uwsgi_time(ms)
50	26.94	26.25	26.22
100	27.92	27.23	27.19
150	26.78	26.1	26.04
200	28.33	27.64	27.52
250	33.32	32.1	30.02
300	30.69	29.36	28.87
350	32.23	30.83	29.9
400	31.06	30.23	29.53

Figure 4.9: Graph of Sys_Perf_Check for uWSGI Processes 24

uWSGI with Processes: 48

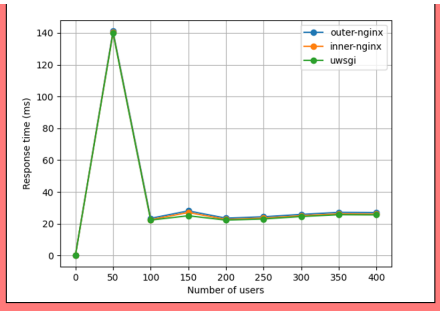

Table containing Response Time(ms) vs number of users

Numusers	outer-nginx_time(ms)	inner-nginx_time(ms)	uwsgi_time(ms)
50	141.03	140.26	140.01
100	23.42	22.57	22.32
150	28.09	27.09	25.03
200	23.53	22.73	22.31
250	24.39	23.54	23.02
300	25.88	25.05	24.54
350	27.17	26.22	25.68
400	27.04	26.06	25.57

Figure 4.10: Graph of Sys_Perf_Check for uWSGI Processes 48

uWSGI with Processes: 64

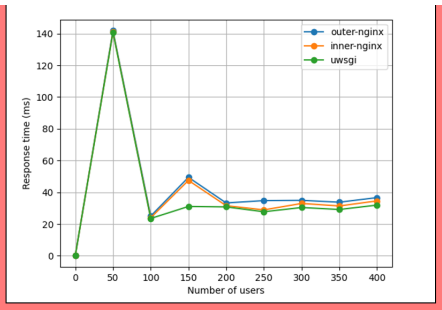

Table containing Response Time(ms) vs number of users

Numusers	outer-nginx_time(ms)	inner-nginx_time(ms)	uwsgi_time(ms)
50	141.78	141.07	140.81
100	24.98	23.93	23.48
150	49.41	47.48	31.05
200	33.26	31.54	30.77
250	34.74	28.9	27.71
300	34.91	32.95	30.39
350	33.73	31.38	29.14
400	36.64	34.55	31.92

Figure 4.11: Graph of Sys_Perf_Check for uWSGI Processes 64

4.4.4 Comparison

- The **worst** performance at **low load (50 users)** is observed when **uWSGI processes is set to 64**. Figure 4.11 (**Response Time = 141.7 ms**).
- The **worst** performance at **high load (400 users)** is observed when **uWSGI processes is set to 5**. Figure 4.8 (**Response Time = 402 ms**).
- The **best** performance at **low load (50 users)** is observed when **uWSGI processes is set to 24**. Figure 4.9 (**Response Time = 26.9 ms**).
- The **best** performance at **high load (400 users)** is observed when **uWSGI processes is set to 48**. Figure 4.10 (**Response Time = 27 ms**).
- The overall **best performance** (most of the time) is observed when **uWSGI processes is set to 48**. Figure 4.10.

Conclusion

The **best configuration at low load is 24 processes**, and for **medium to high load is 48 processes**, which aligns with the ideal configuration mentioned in the uWSGI documentation (2 * number of cores = 2 * 24 = 48).
It is possible to identify **uWSGI or the layer above** as the bottleneck when **uWSGI is poorly configured with 5 processes**. This demonstrates the use case of our tool.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

In this project, we developed a tool, Sys_Perf_Check , to identify bottlenecks in the system stack of a web application (SAFE). We provided an overview of the different components of our project, including the client-side script and server-side script. We discussed the implementation details of Sys_Perf_Check , covering both the client-side script and the back-end script.

We demonstrated how to use Sys_Perf_Check , starting from the initial setup to the order in which the scripts need to be executed. Currently, Sys_Perf_Check focuses on CPU-bound functions and does not specifically address IO devices such as disk IO. However, we successfully applied Sys_Perf_Check to the SAFE web application and showed how it can be used to identify bottleneck components in the stack.

5.2 Future Work

There are several areas for future work and improvements for Sys_Perf_Check :

- Extend the framework to include IO devices, particularly disk IO, to analyze the performance impact of IO operations on the system.
- Develop a graphical user interface (GUI) to run the scripts and provide a more user-friendly experience.
- Implement an automated detection mechanism within Sys_Perf_Check to identify bottlenecks without requiring manual analysis of the generated graphs.
- Include database components in the system stack to analyze their performance impact and identify potential bottlenecks.

By addressing these future work areas, Sys_Perf_Check can become a more comprehensive and user-friendly tool for identifying and resolving performance bottlenecks in web applications.

Bibliography

[1] argparse — parser for command-line options, arguments and sub-commands. https://docs.python.org/3/library/argparse.html.

[2] ftplib - ftp protocol client. https://docs.python.org/3/library/ftplib.html.

[3] Metrohash, a fast non-cryptographic hash function. https://pypi.org/project/metrohash/.

[4] Performance testing script by jatin. https://gitlab.com/dvramana/safe_server_v2/-/tree/jatin_performance_testing.

[5] Things to know (best practices and "issues") read it !!! https://uwsgi-docs.readthedocs.io/en/latest/ThingsToKnow.html.

[6] What is locust? https://docs.locust.io/en/stable/what-is-locust.html.