SPC : A tool to diagnose bottleneck component in a back end server software stack.

BPS-2 presentation by

Jatin Lachhwani

(Guide : Prof. Bhaskaran Raman)

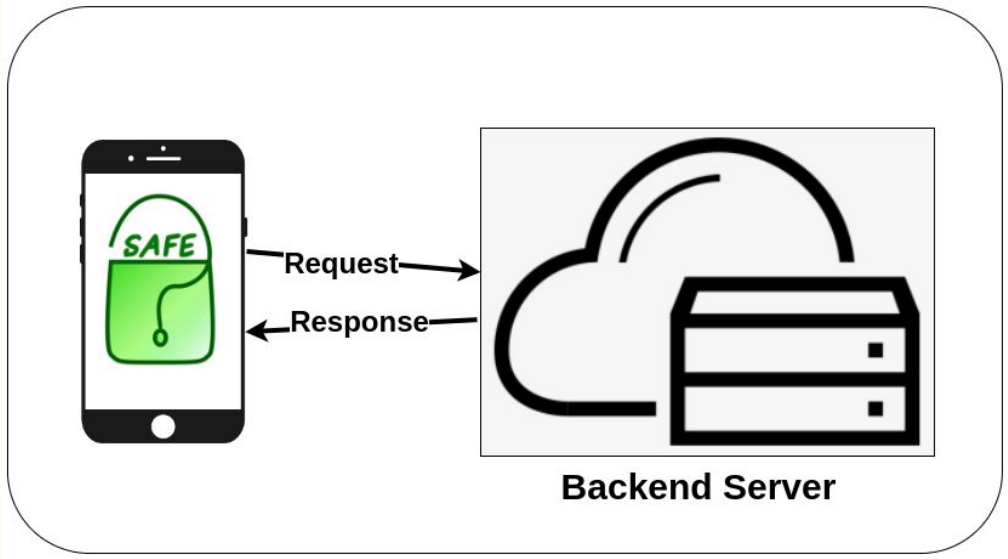
Contents

Introduction

Background | SAFE

SAFE - Smart Authenticated Fast Exams

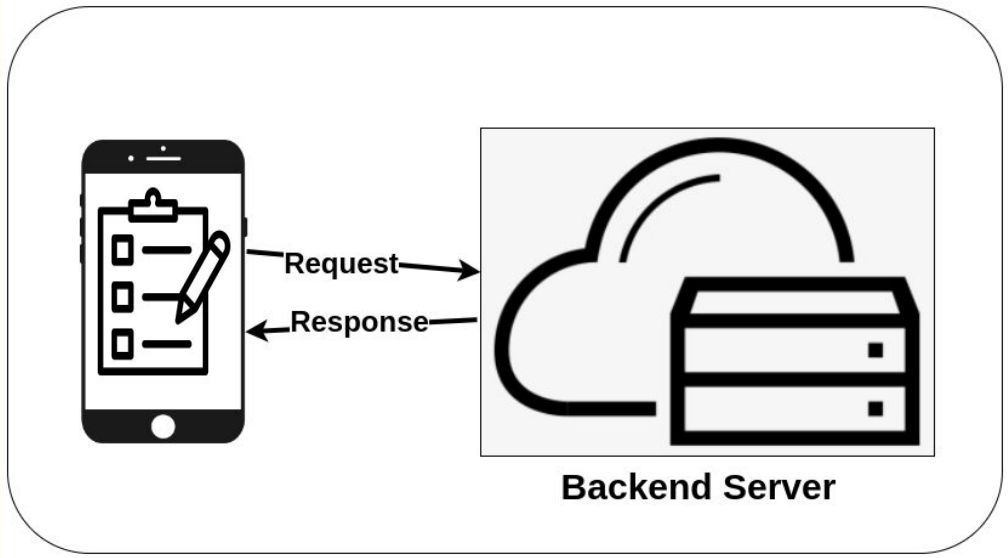
Take quiz in classroom using their own smartphones.



Background | SAFE | Core Functionality

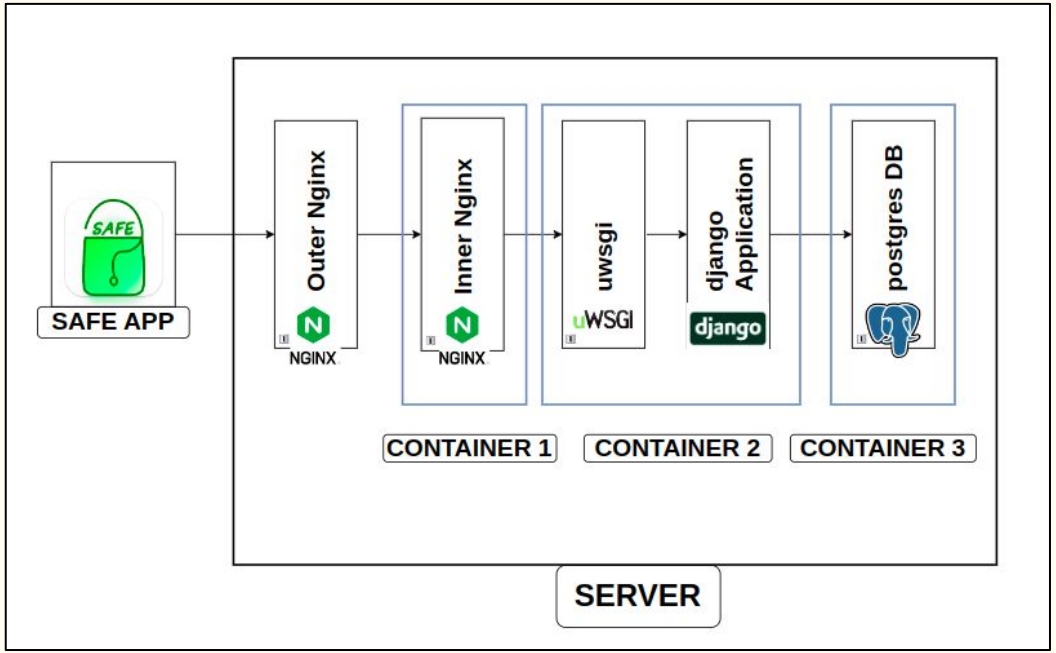
SAFE - Smart Authenticated Fast Exams

Take quiz in classroom using their own smartphones.



Background | SAFE | Architecture

- SAFE app
- Outer nginx
- Inner nginx
- uWSGI
- Django
- Postgres



Background | SAFE | Core Functionality

Steps : **Taking a Quiz in SAFE**

0. Instructor starts the quiz 
1. **Login** 
2. Get **Course List** 
3. Get **Quiz List** 
4. Get **Quiz Info** 
5. **Quiz Download** 
6. **Quiz Authenticate** 
7. **Quiz Submit** 

Background | Previous Work

To **identify bottleneck APIs** among them.

0. Instructor starts the quiz 
1. **Login** 
2. Get **Course List** 
3. Get **Quiz List** 
4. Get **Quiz Info** 
5. **Quiz Download** 
6. **Quiz Authenticate** 
7. **Quiz Submit** 

Background | Previous Work

Tried to find bottleneck APIs using Locust.

API	Avg. Resp. time(ms)
1. Login	4279
2. Get Course List	4378
3. Get Quiz List	4278
4. Get Quiz Info	9407
5. Quiz Download	7766
6. Quiz Authenticate	4278
7. Quiz Submit	5704

Test Conditions

Num of users : 100

Spawn Rate : 10/s

Background | Previous Work

Tried to find bottleneck APIs using Locust.

API	Avg. Resp. time(ms)
1. Login	4279
2. Get Course List	4378
3. Get Quiz List	4278
4. Get Quiz Info	9407
5. Quiz Download	7766
6. Quiz Authenticate	4278
7. Quiz Submit	5704

Too high for these conditions

Test Conditions

Num of users : 100

Spawn Rate : 10/s

Background | Previous Work

Tried to find bottleneck APIs using Locust.

API	Avg. Resp. time(ms)
1. Login	4279
2. Get Course List	4378
3. Get Quiz List	4278
4. Get Quiz Info	9407
5. Quiz Download	7766
6. Quiz Authenticate	4278
7. Quiz Submit	5704

Too high for these conditions

Possibility of a system level bottleneck

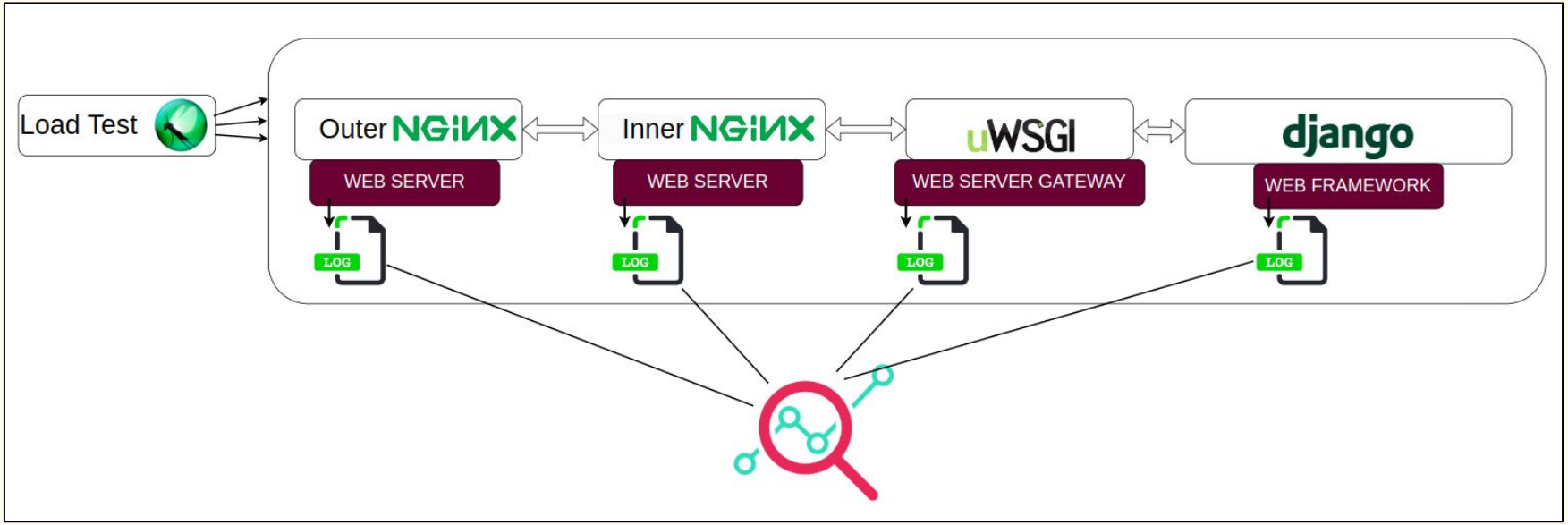
Test Conditions

Num of users : 100

Spawn Rate : 10/s

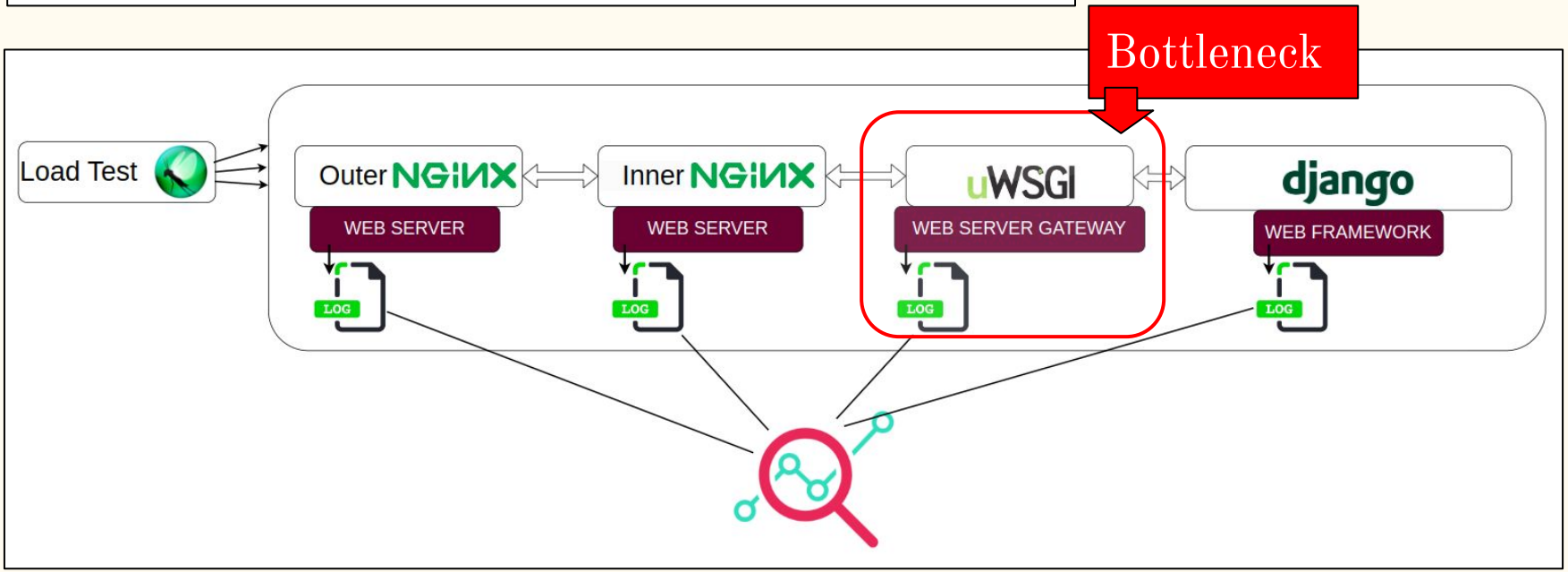
Background | Previous Work

Analysed the logs of each component in SAFE



Background | Previous Work

Analysed the logs of each component in SAFE



Background | Previous Work

After Properly configuring **uWSGI** (bottleneck component)

API	Avg. Resp. time(ms)	Avg. Resp. time(ms) (uWSGI configured)
1. Login	4279	1968
2. Get Course List	4378	438
3. Get Quiz List	4278	519
4. Get Quiz Info	9407	306
5. Quiz Download	7766	325
6. Quiz Authenticate	4278	405
7. Quiz Submit	5704	783

Test Conditions

Num of users : 100

Spawn Rate : 10

Background | Previous Work Conclusion

System level bottleneck component(s)

are hard to detect

can stay unnoticed for too long

have a significant impact on overall performance

Problem Statement

To create a tool that is

able to detect System level bottleneck component(s).

independent of any application specific APIs

easy to use and configure

visual feedback for analysis of components.

SPC Overview

Assumptions

Possible to customize the log format for all components in the software stack.

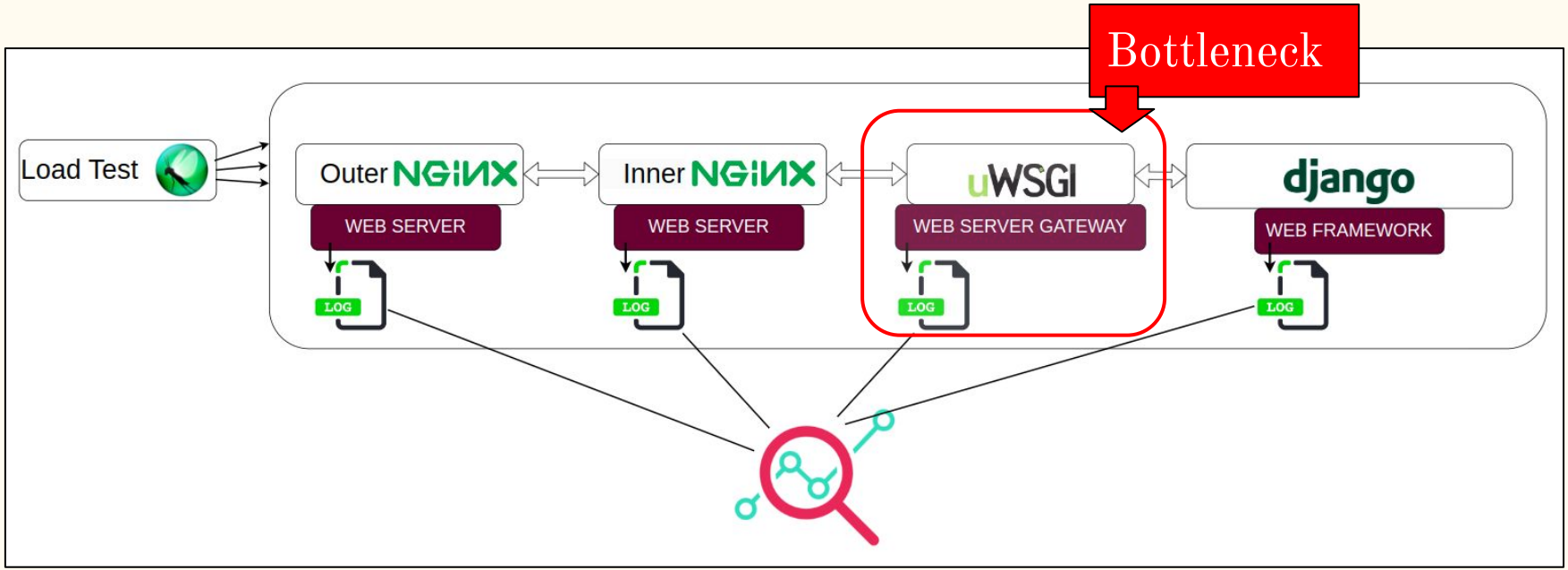
Components are stacked linearly (like nginx-uwsgi-django)

URL of the request is visible at all components.

Mainly designed for backend web application stacks.

Test for SPC

Good test for SPC : It can detect the uWSGI bottleneck we detected in our previous work.



SPC Design Brainstorm

How can we detect a **bottleneck like ill-configured uWSGI?**

while ensuring

independent of any application specific APIs

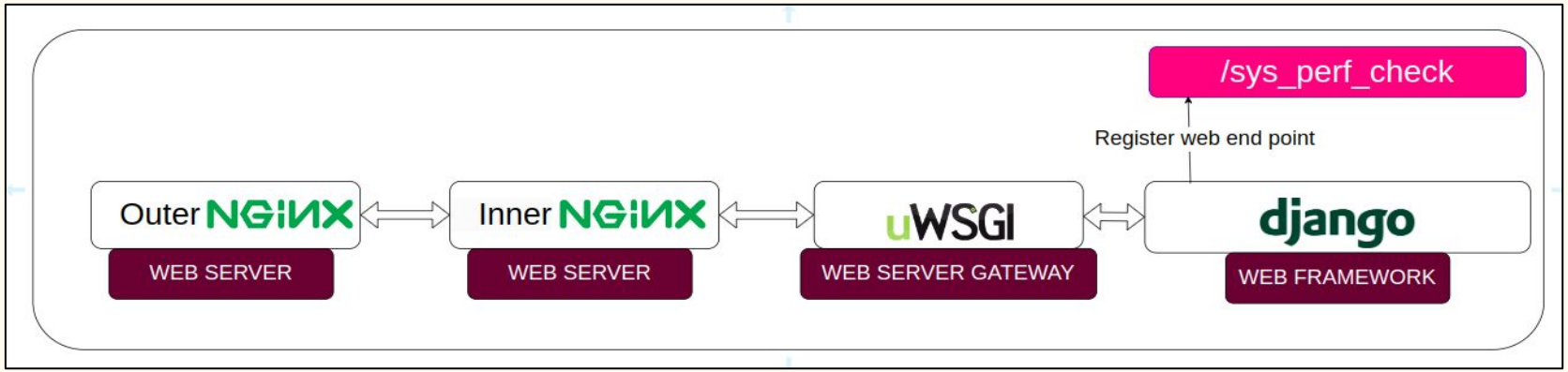
easy to use and configure

visual feedback for analysis of components.

SPC Design Brainstorm | Idea 1

Add a specific API endpoint at application code independent of other APIs.

Say : sys_perf_check

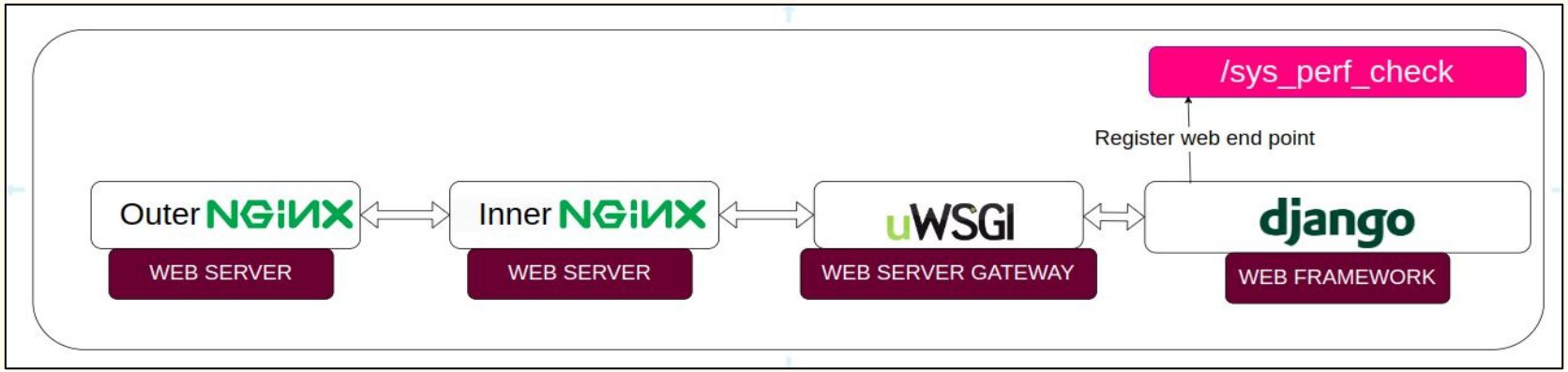


SPC Design Brainstorm | Idea 1

Add a specific API endpoint at application code independent of other APIs.

Say : sys_perf_check

Why?



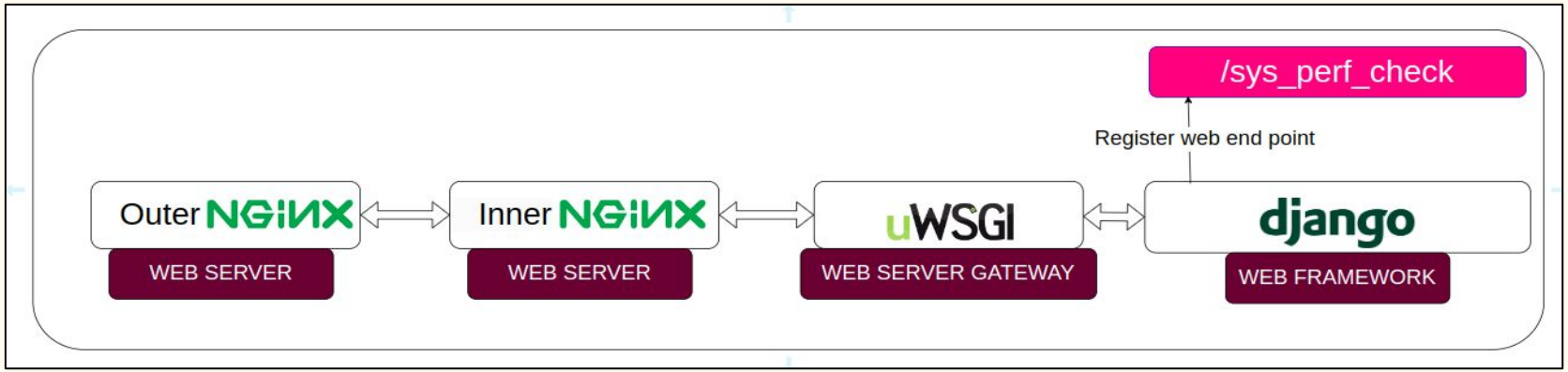
SPC Design Brainstorm | Idea 1

Add a specific API endpoint at application code independent of other APIs.

Say : sys_perf_check

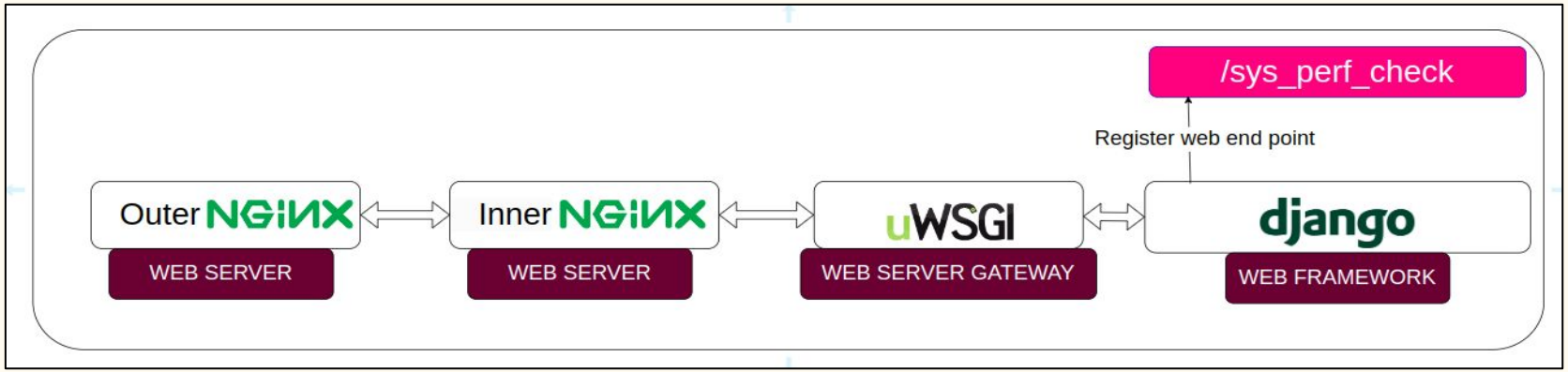
Why?

independent of any application specific APIs



SPC Design Brainstorm | Idea 1

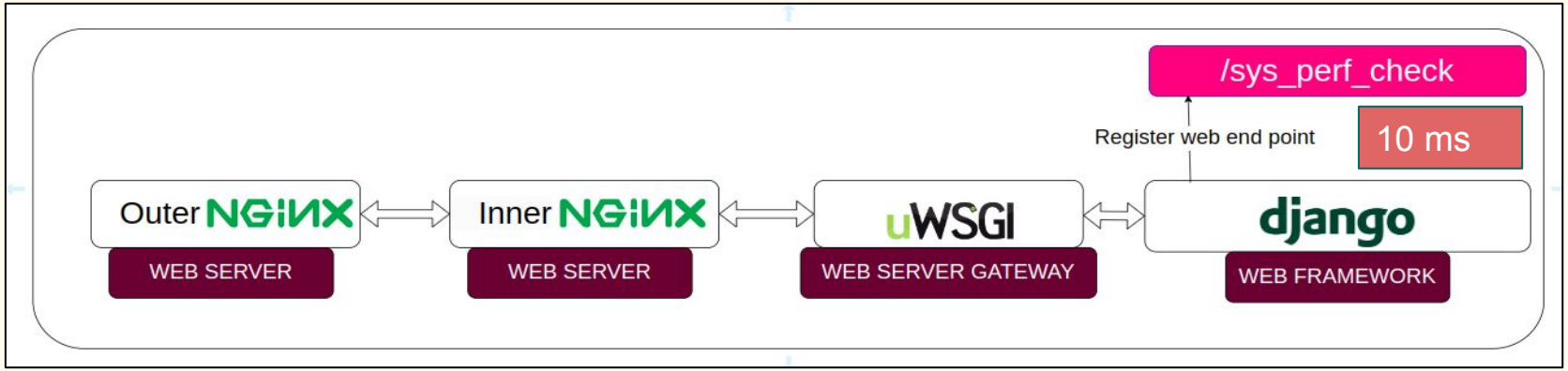
What will /sys_perf_check do?



SPC Design Brainstorm | Idea 1

What will /sys_perf_check do?

It will execute for a bounded time then exit. (say **10ms**)

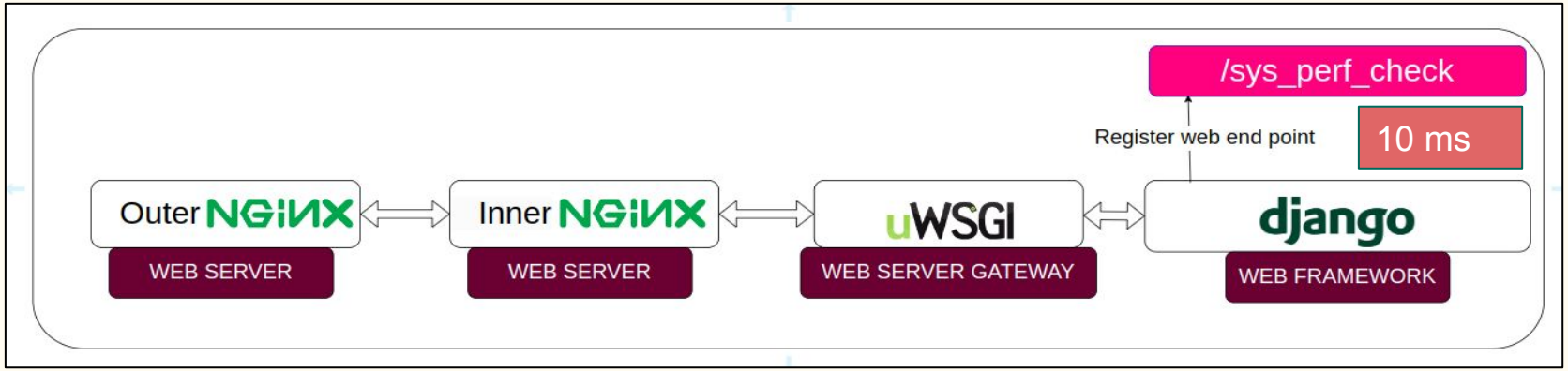


SPC Design Brainstorm | Idea 1

What will /sys_perf_check do?

It will execute for a bounded time then exit. (say **10ms**)

Why?



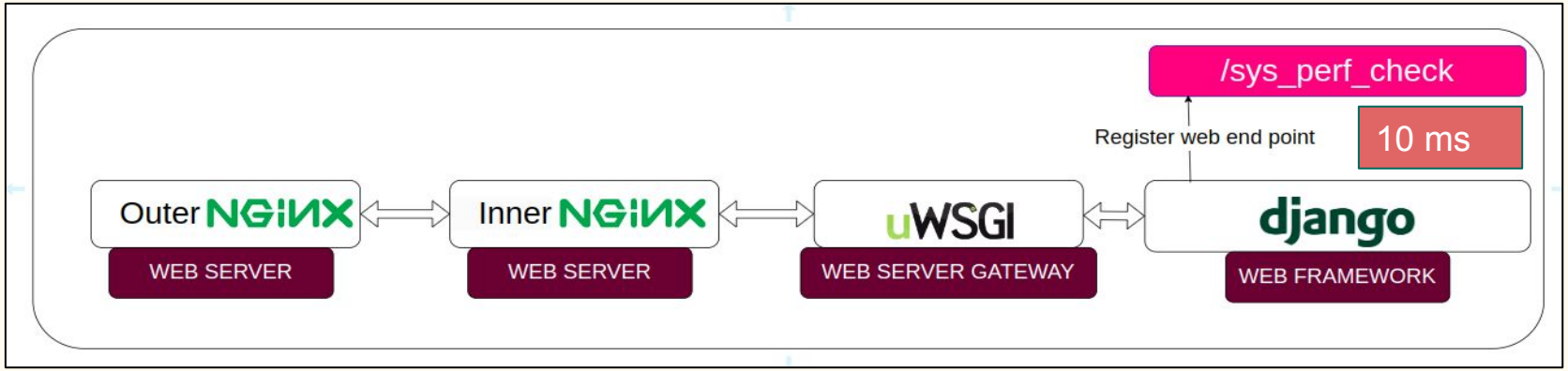
SPC Design Brainstorm | Idea 1

What will /sys_perf_check do?

It will execute for a bounded time then exit. (say **10ms**)

Why?

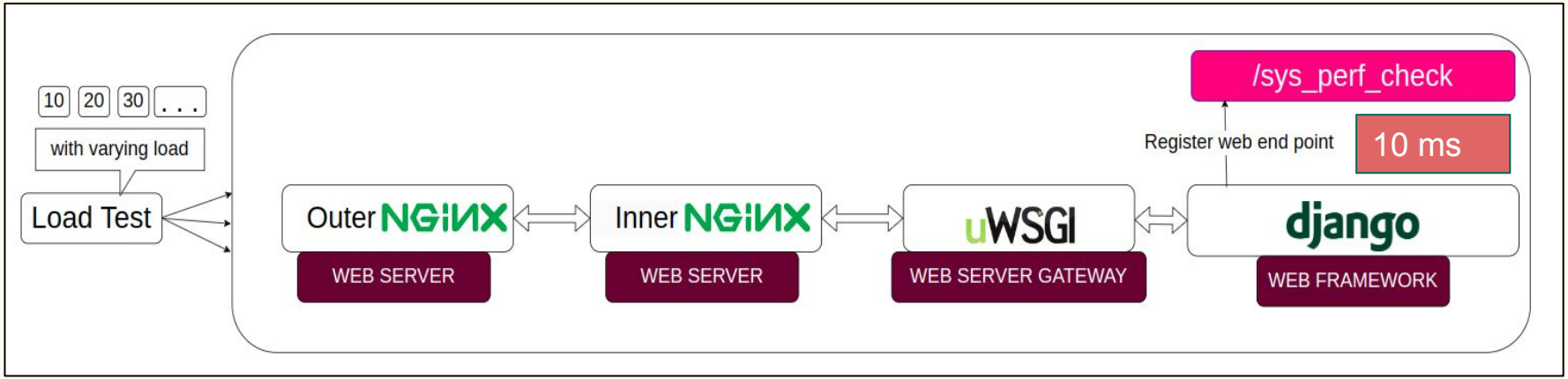
We observed in our previous work, **django code** would **execute** for roughly **XY ms.** (empirical). (Adjusted based upon the system under test.)



SPC Design Brainstorm | Idea 1

Now we can do **a load test** with **varying number of users.**

We can **analyze** the **component logs** and **find the bottleneck.**

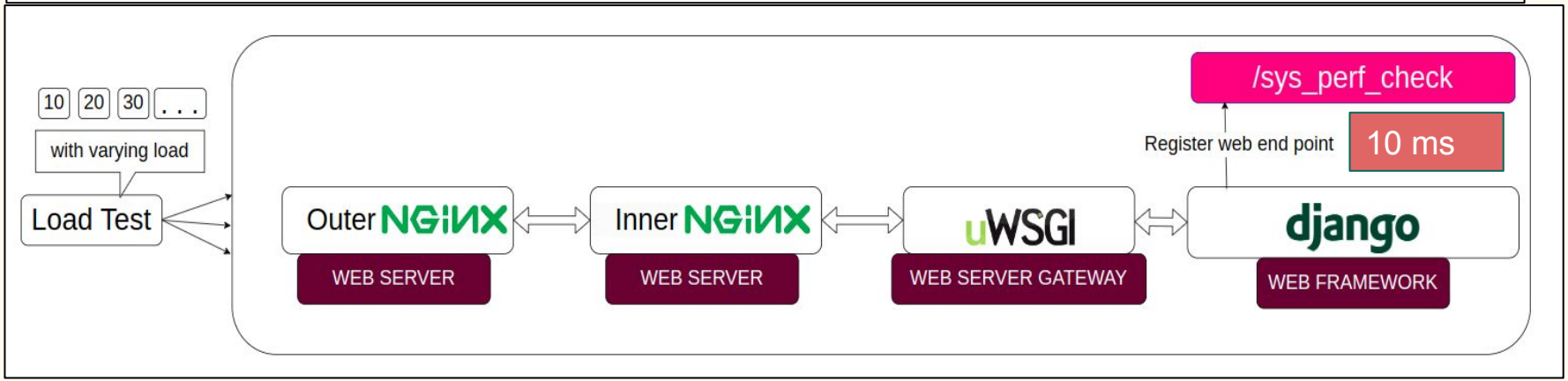


SPC Design Brainstorm | Idea 2

Now we can do **a load test** with **varying number of users.**

We can **analyze** the **component logs** and **find the bottleneck.**

Wait. But **there is a catch.** How to identify the number of users for a request entry in a log?

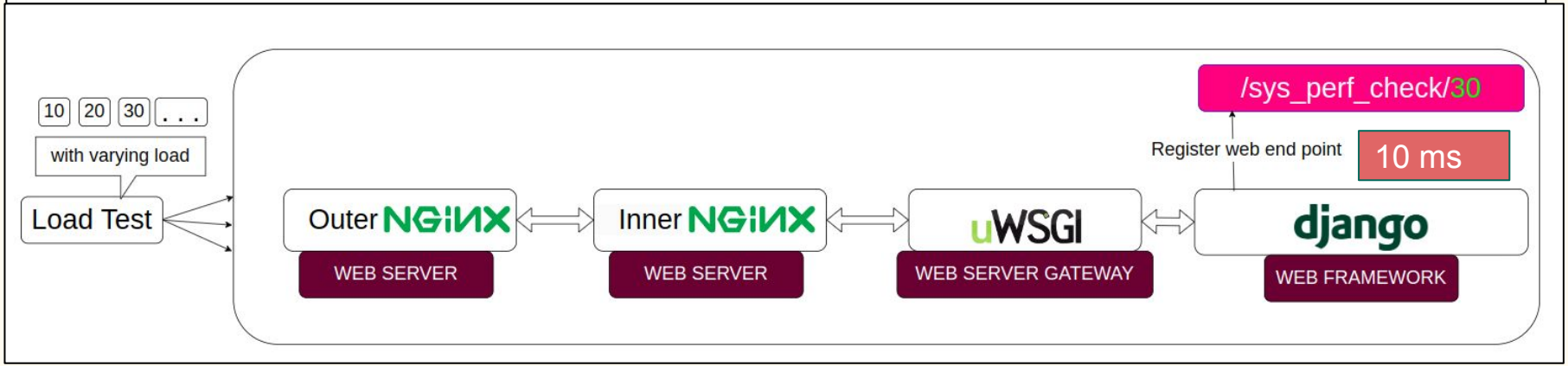


SPC Design Brainstorm | Idea 2

Now we can do **a load test** with **varying number of users.**

We can **analyze** the **component logs** and **find the bottleneck.**

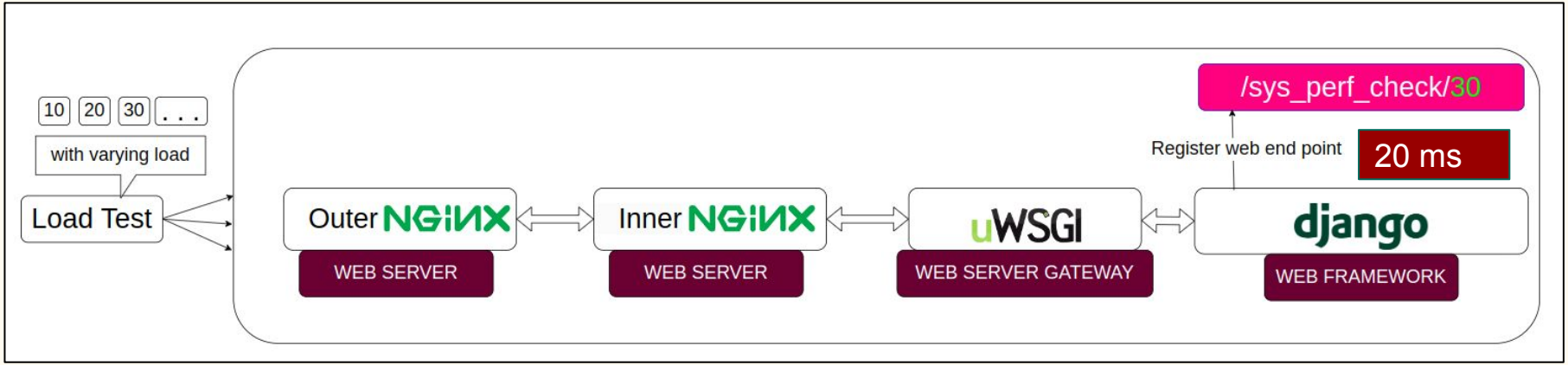
Wait. But **there is a catch.** How to identify the number of users for a request entry in a log? By **appending number of users as a part of request.**



SPC Design Brainstorm | Idea 3

Suppose **we change something** for a **new test.**(say **endpoint time to 20 ms**)

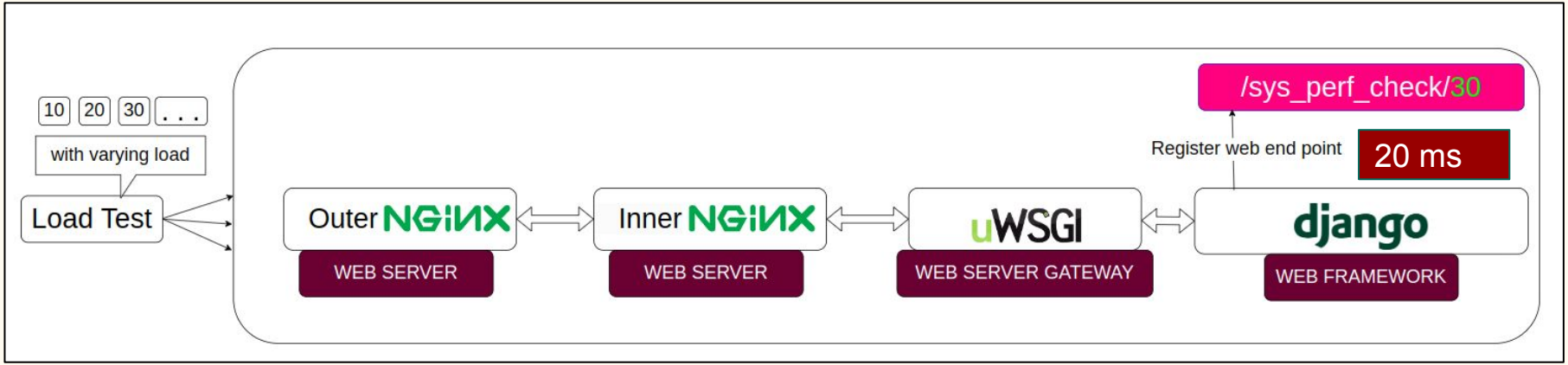
After the load test when **we extract the logs. We might get logs from the previous test.**



SPC Design Brainstorm | Idea 3

Suppose **we change something** for a **new test.**(say **endpoint time to 20 ms**)

After the load test when **we extract the logs. We might get logs from the previous test.** What now?

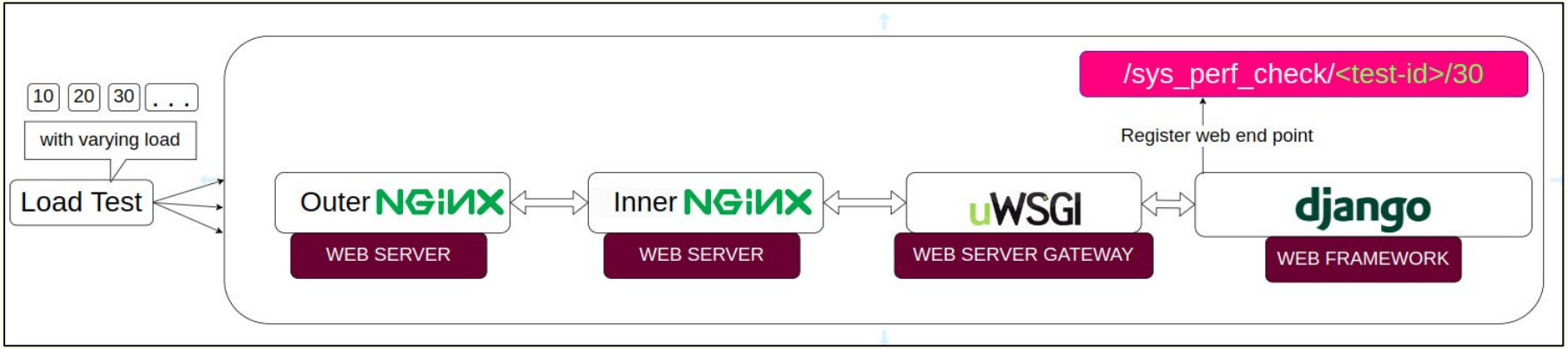


SPC Design Brainstorm | Idea 3

Suppose **we change something** for a **new test.**(say **endpoint time to 20 ms**)

After the load test when **we extract the logs. We might get logs from the previous test.** What now?. We need a way to distinguish tests uniquely

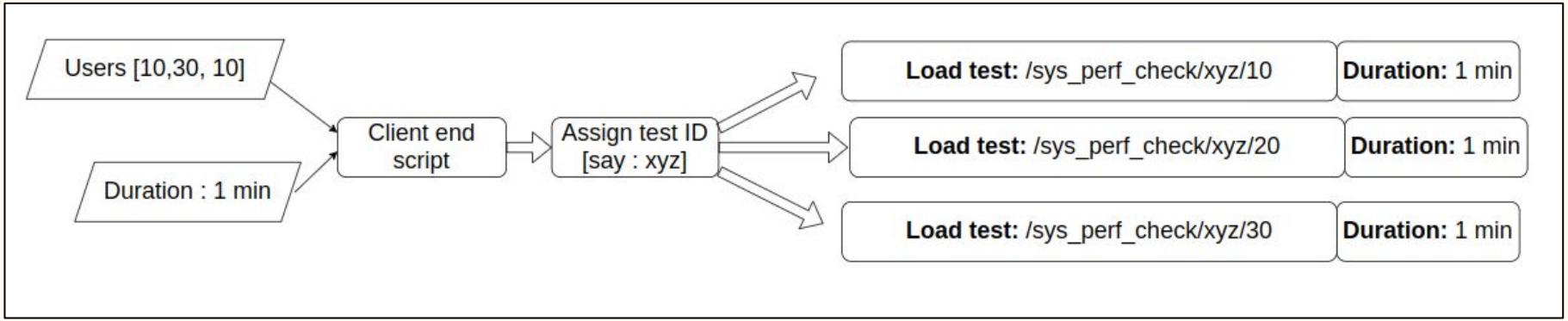
We will add a unique test id in the request for each varying load test.



SPC Design Brainstorm | Idea Aggregation

1. Register endpoint sys_perf_check
2. Script that generates load for varying number of users
3. Assigns Id to each instance of test.

These things will be done by a script : **client end script**



SPC Design Brainstorm | Client end script

client end script

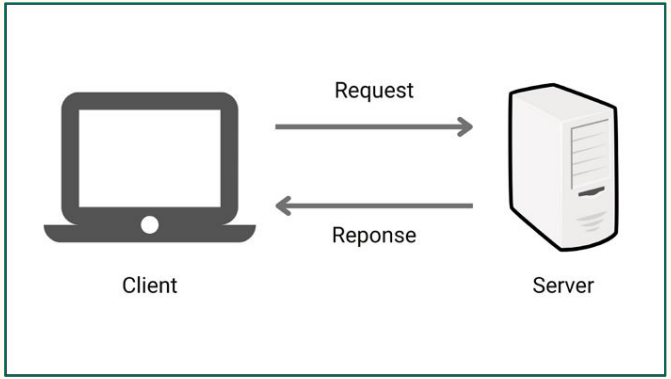
Why?

SPC Design Brainstorm | Client end script

client end script

Why?

Generally **load generation for testing** happens on a **separate machine.**



To keep the **server machine** same for all tests.

SPC Design Brainstorm | Idea 4

Once the client end script executes

We need something to

extract the logs

transfer them to client end script for analysis

SPC Design Brainstorm | Idea 4

Once the client end script executes

We need something to

extract the logs

transfer them to client end script for analysis

Server end script

SPC Design Brainstorm | Idea 4

Once the client end script executes

We need something to

extract the logs

transfer them to client end script for analysis

Server end script

SPC: easy to use and configure

SPC Design Brainstorm | Idea 5

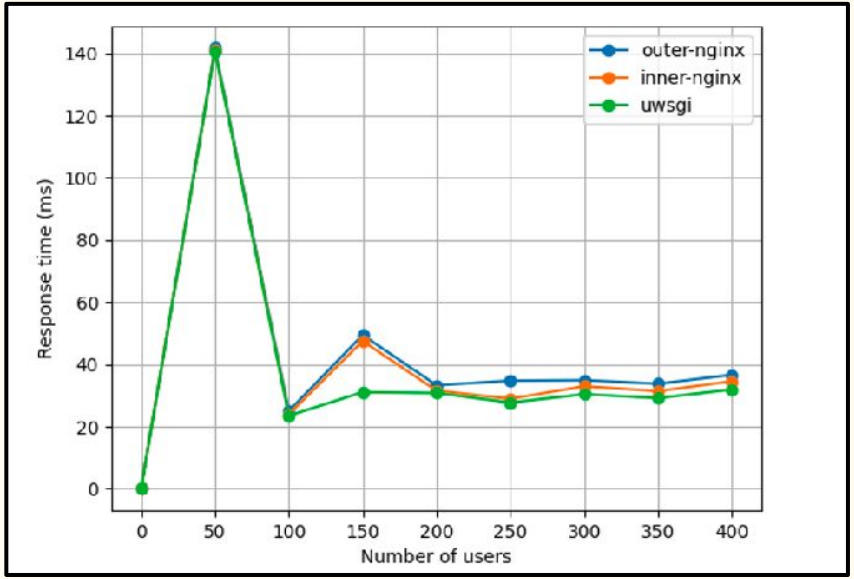
Once logs are received : Analyze them

SPC Design Brainstorm | Idea 5

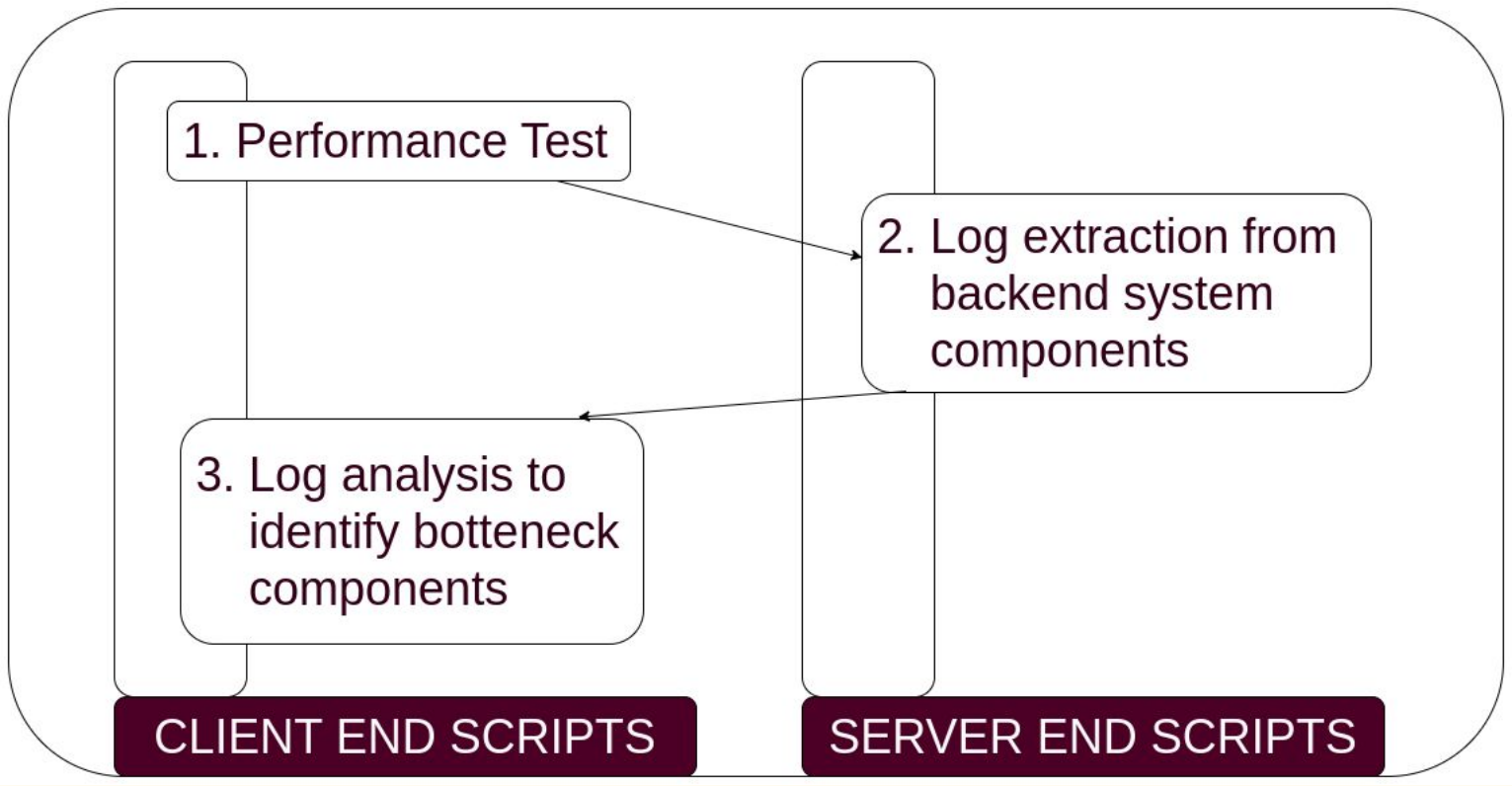
Once logs are received : Analyze them

Display graphical results for analysis.

visual feedback for analysis of components.

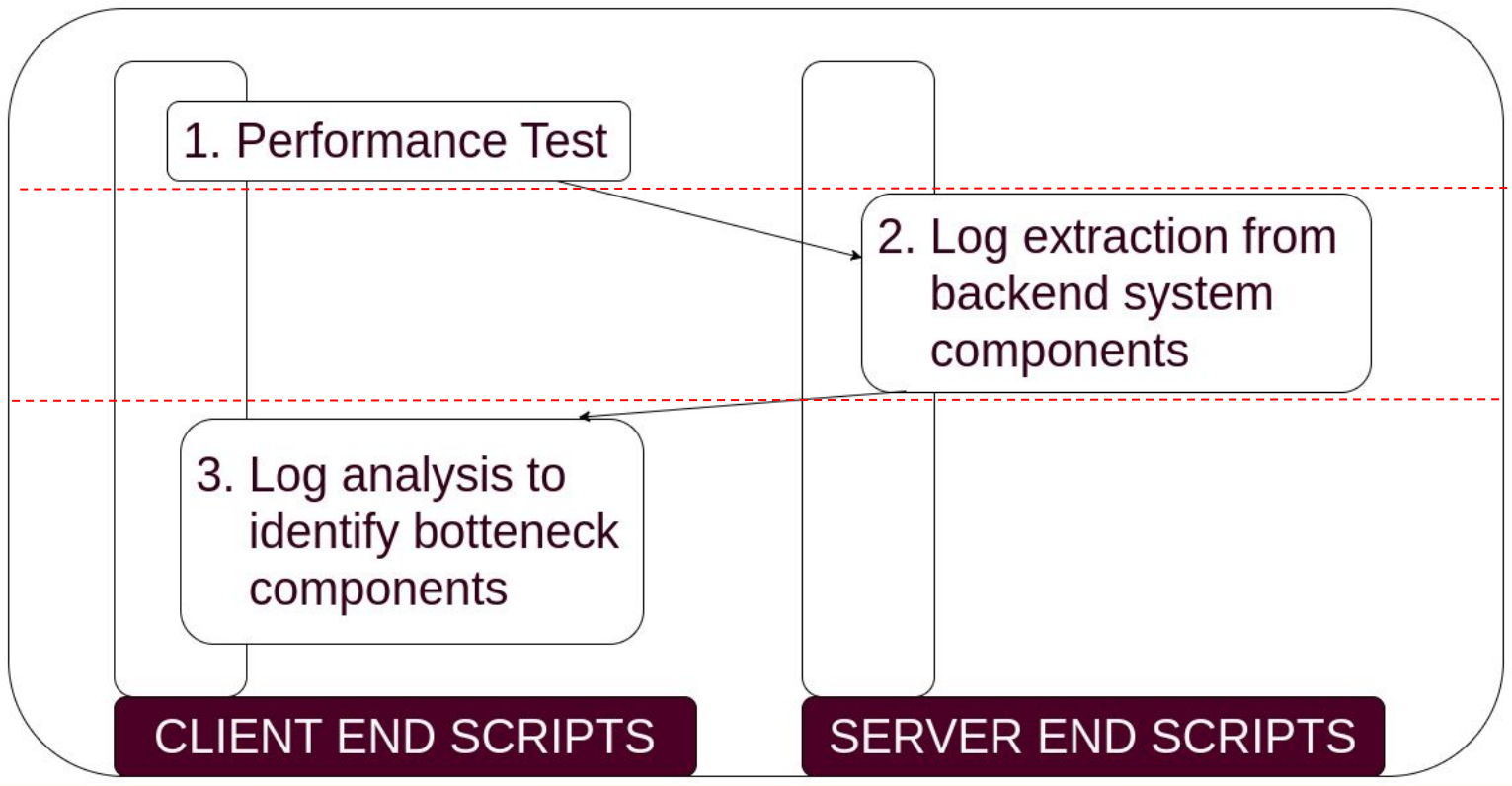


SPC Design


1. Performance Test
2. Log extraction from backend system components
3. Log analysis to identify botteneck components
CLIENT END SCRIPTS
SERVER END SCRIPTS

SPC Implementation

3 Phases


1. Performance Test
2. Log extraction from backend system components
3. Log analysis to identify botteneck components
CLIENT END SCRIPTS
SERVER END SCRIPTS

Phase 1 - Performance Test | Client End Script

1. Client end script input

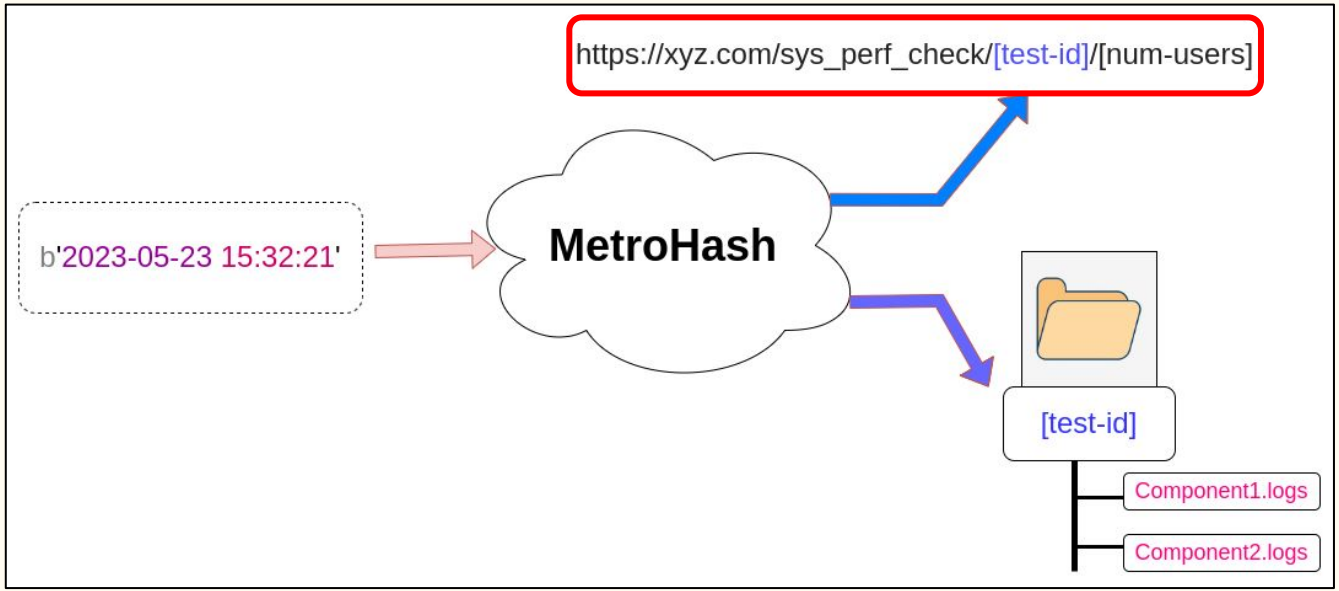
```
λ › ./client_end_script.py -h                                          thesis/locust_fi
usage: ./client_end_script.py [-h] -l LOWER_BOUND_USERS -u UPPER_BOUND_USERS -s STEP_SIZE
                              [-t RUN_TIME]

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
```

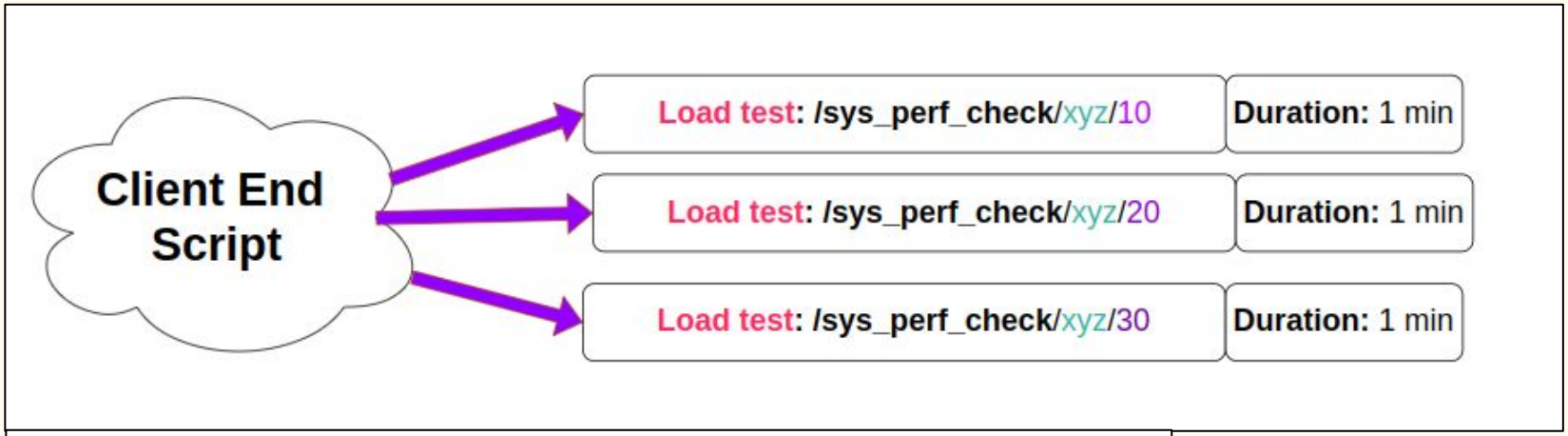
Phase 1 - Performance Test | Client End Script

2. Generating unique id for this instance of script.



Phase 1 - Performance Test | Client End Script

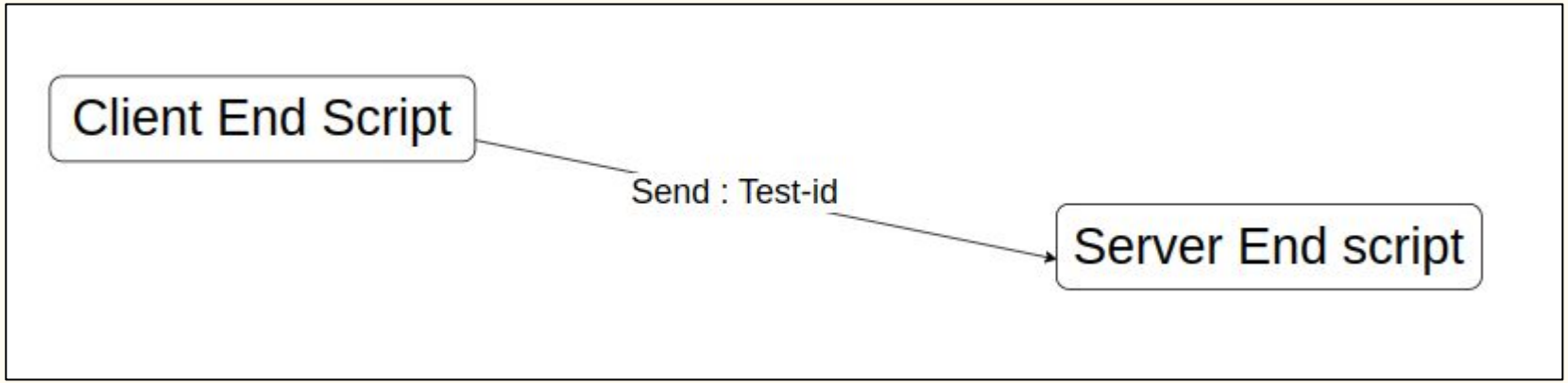
3. Performance Test (uses locust)



say : l: 10, u:30, s:10, d:1 min | test-id : xyz

Transition Phase 1 to Phase 2

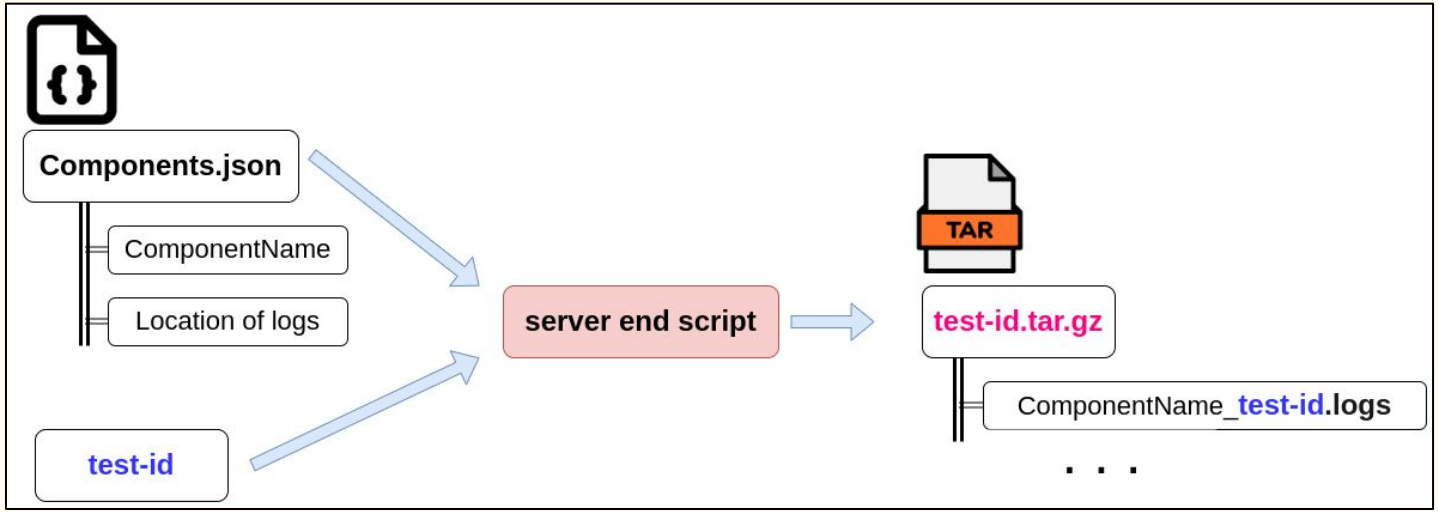
After Phase 1, client end script will **send the test-id** to server end script.



Phase 2 - Log Extraction | Server End Script

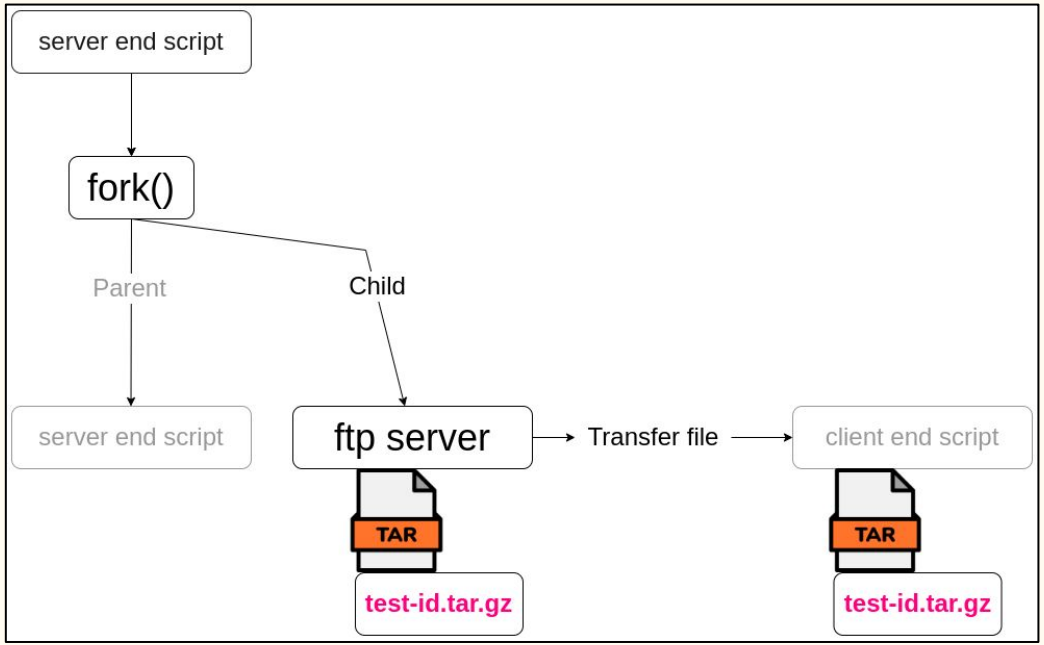
4. Log Extraction from Component logs.

It requires a **components.json** file to describe the location of component logs.



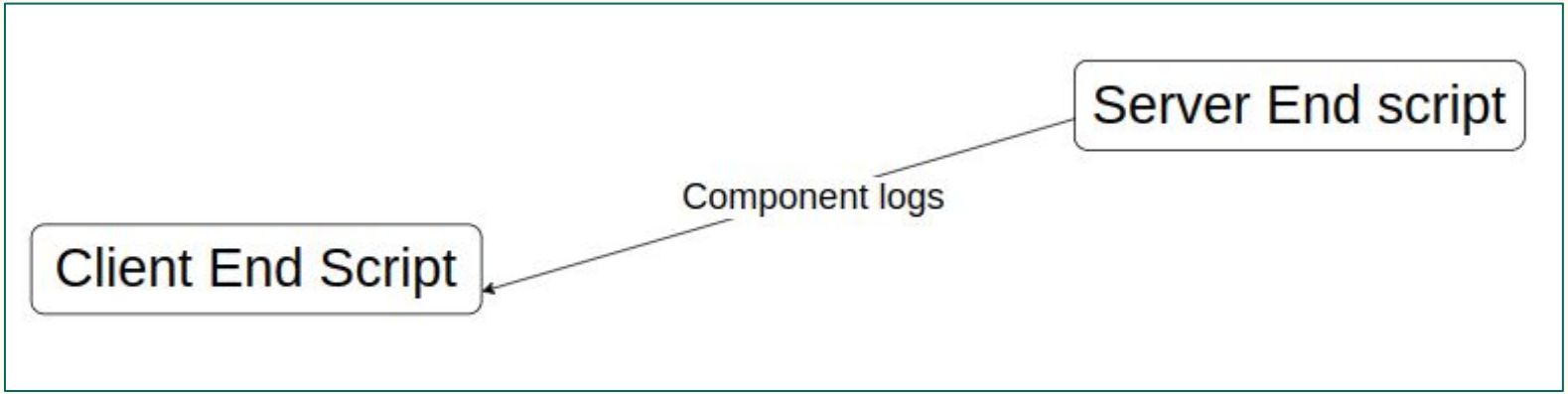
Phase 2 - Log Extraction | Server End Script

5. Creating an ftp server to transfer logs



Transition Phase 2 to Phase 3

After Phase 2, Components logs are transferred to client end script.



Phase 3 - Graph for analysis| Client End Script

6. Component logs are used to create graph for analysis.

Image and Table Presentation

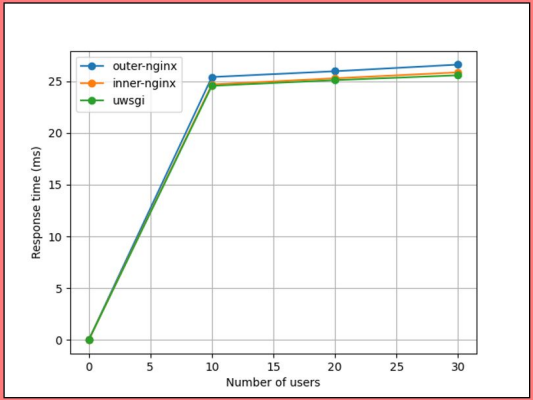


Table containing Response Time(ms) vs number of users

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

This can be used to detect the bottleneck.

SPC Usage Testing

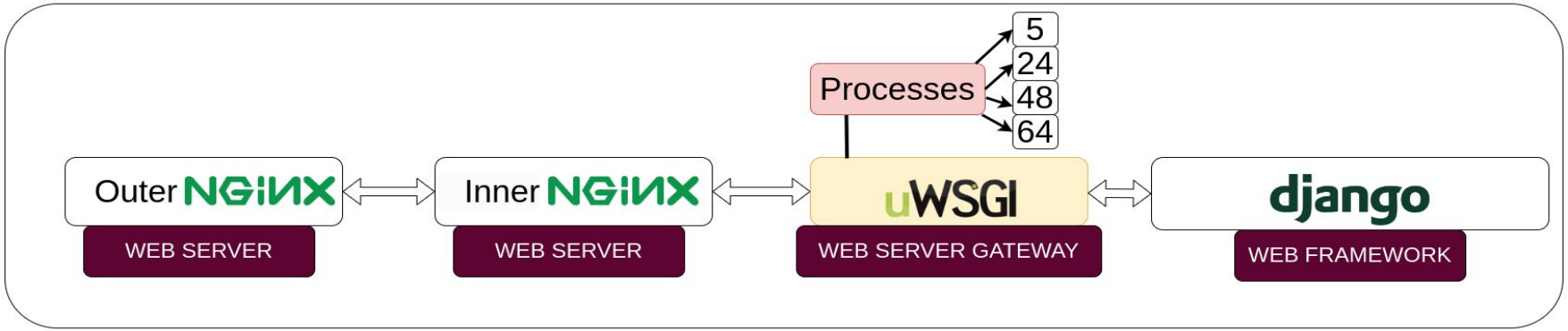
Test Setup | Machine Specifications

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

Machine : safev2 is hosted

Test Setup | uWSGI configurations

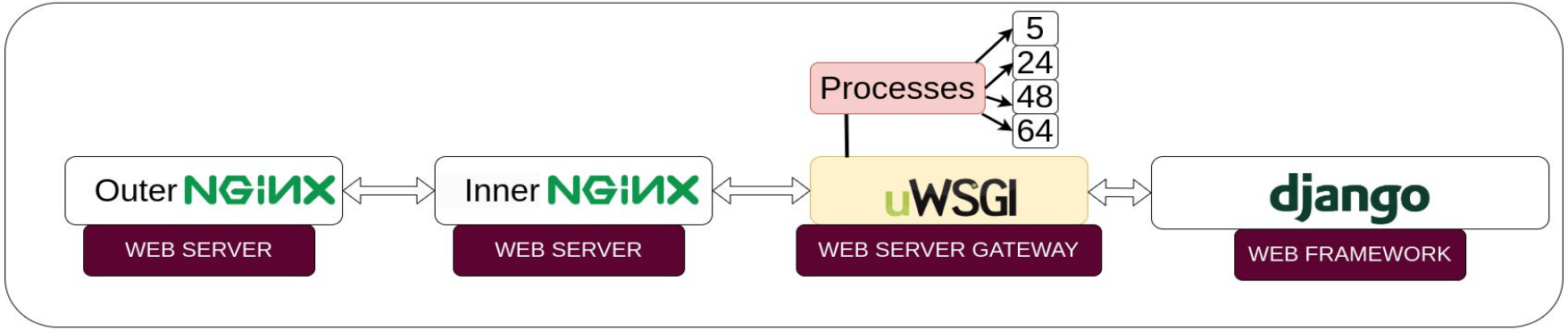
We tried **four configurations** for number of processes in **uWSGI**



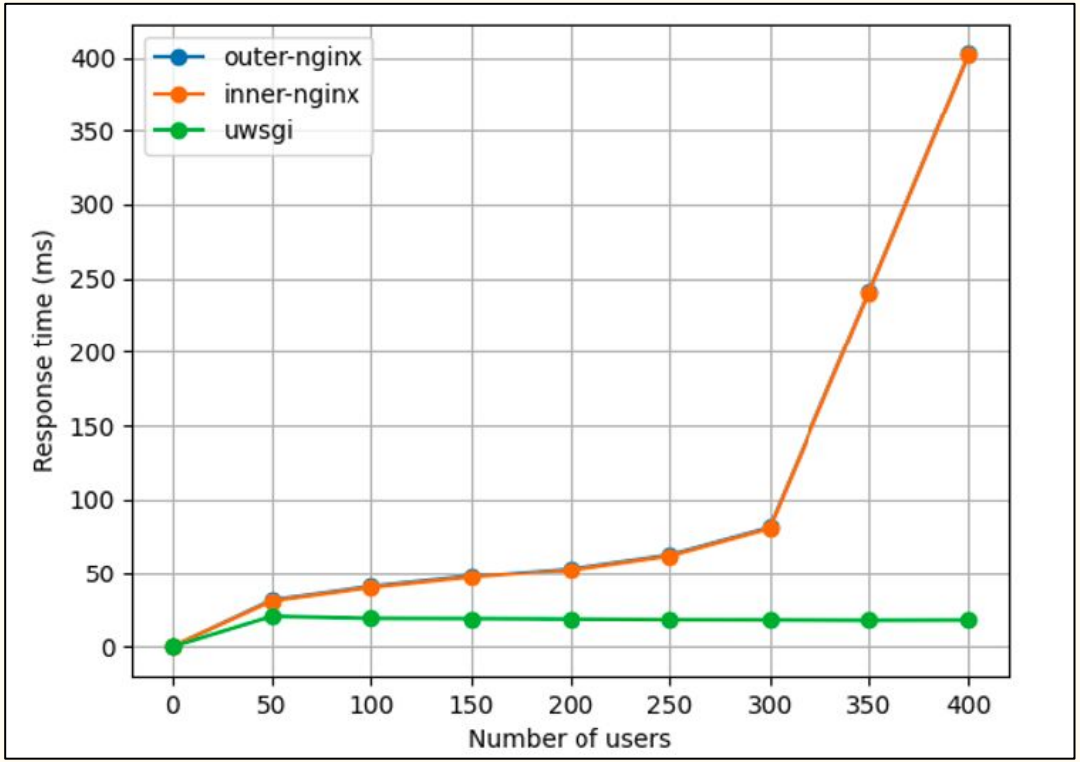
Test Setup | uWSGI configurations

5: lower than ideal config.	24: equal to num of cores
48: ideal config. (documentation)	64: higher than ideal config.

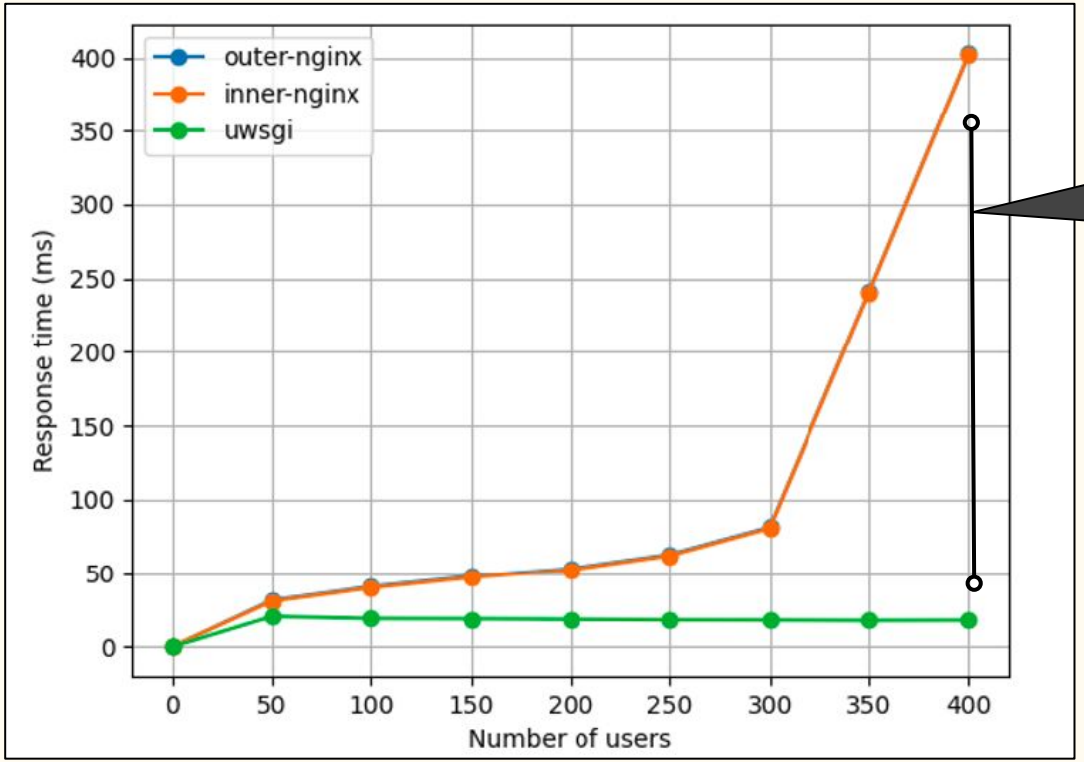
Test: `$ ./client_end_script -l 50 -u 400 -s 50 -t 60`



Final Test | uWSGI Processes 5

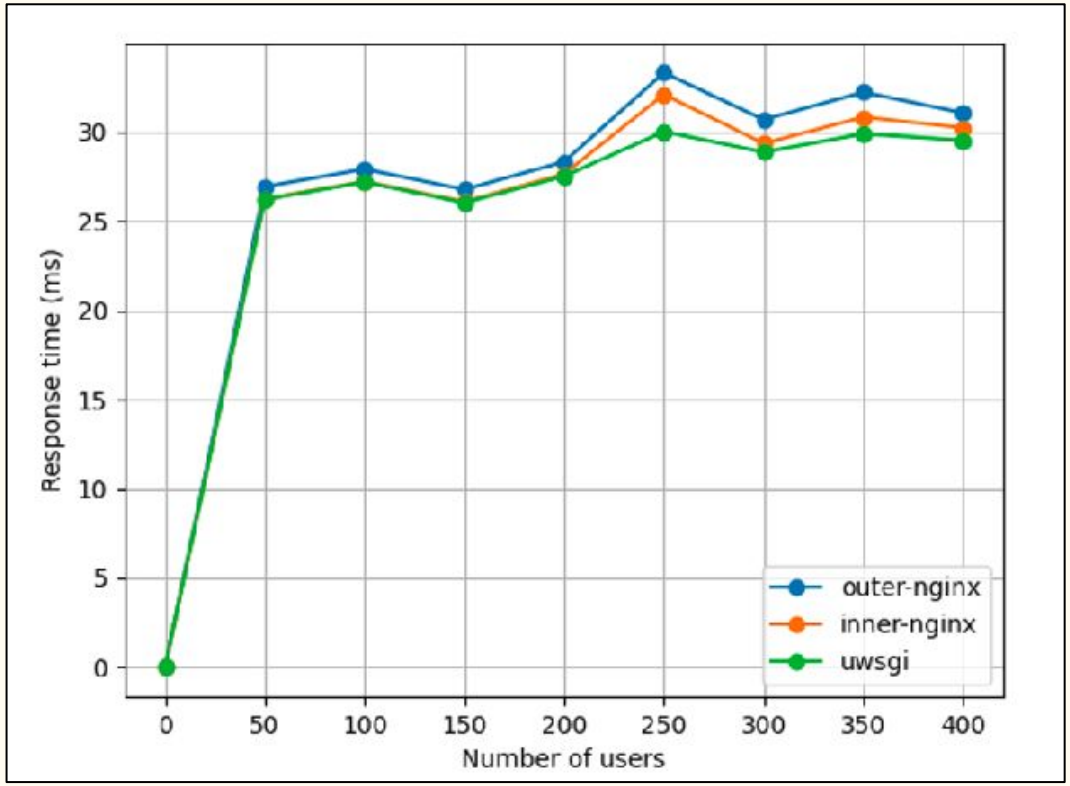


Final Test | uWSGI Processes 5

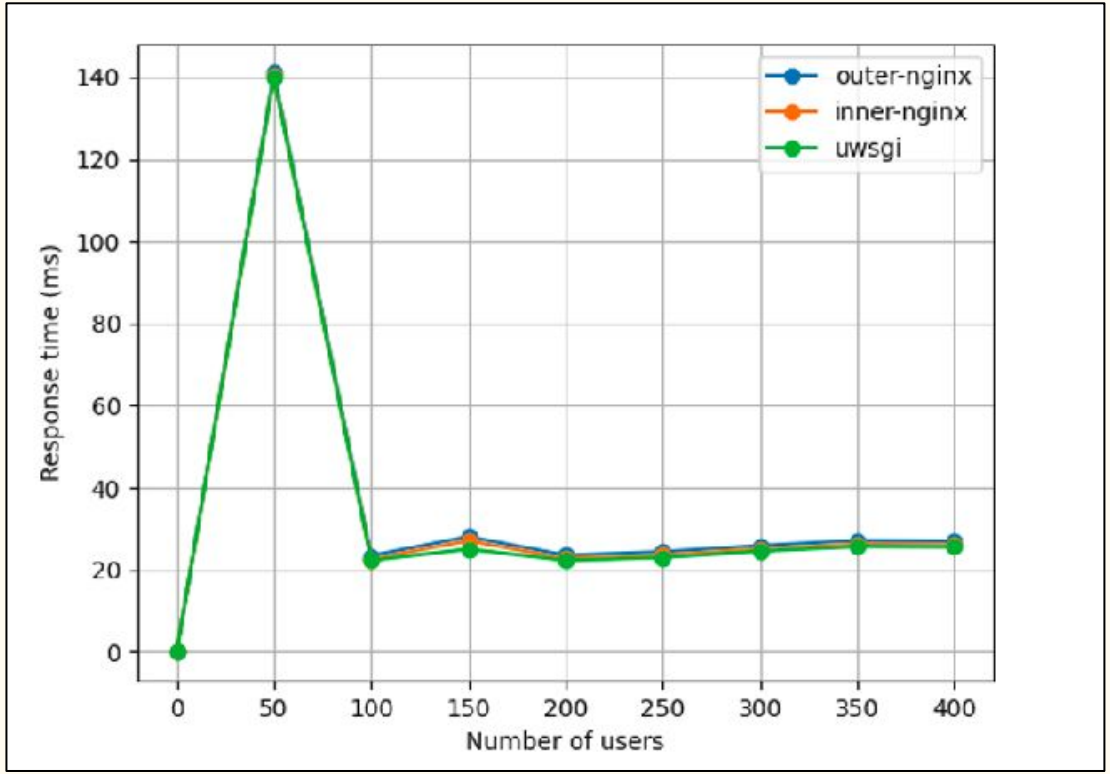


This wide gap indicates a bottleneck.

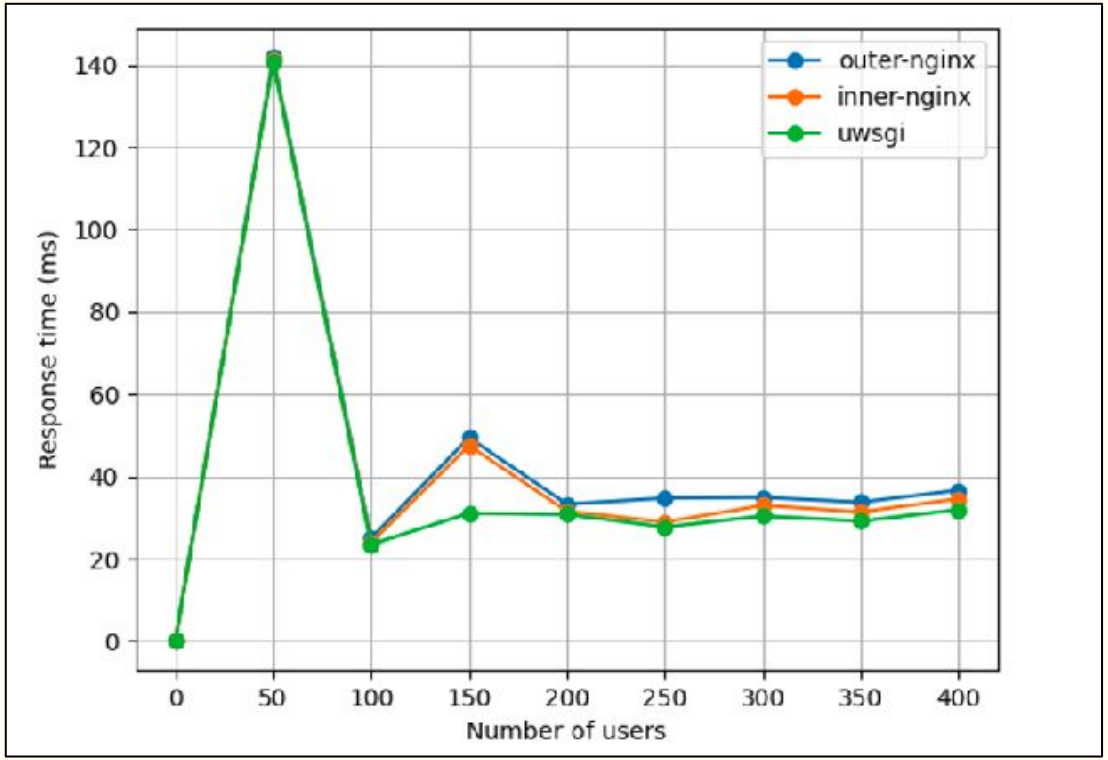
Final Test | uWSGI Processes 24



Final Test | uWSGI Processes 48



Final Test | uWSGI Processes 64



Final Test | Results

RESULTS	Worst Performance	Best Performance
Low load(50 users)	uWSGI Processes : 64 Response time : 141.7 ms	uWSGI Processes : 24 Response time : 26.9 ms
High load(400 users)	uWSGI Processes : 5 Response time :402 ms	uWSGI Processes : 48 Response time : 27 ms

Conclusion & Future Work

Conclusion

SPC works and it helps in bottleneck detection

It satisfies the following conditions :

independent of any application specific APIs

easy to use and configure

visual feedback for analysis of components.

Conclusion | Other use cases

To **fine tune application performance**

To check **impact of certain API endpoints by modifying /sys_perf_check API**

Future Work

Figure out ways to **extend SPC for more use cases.**

Experimenting with **different implementations of sys_perf_check Endpoint.**

More **user friendly naming for test-id** names instead of metrohash

Extend it **for non Linear stacks** as well.

The End