

**Simple Task Queue Processing Service**

# **Performance Test Report**

**Senior QA Engineer  
Elina Nurgaleeva**

## **Table of Contents**

<b>Simple Task Queue Processing Service</b>	<b>3</b>
<b>Performance Test Goal</b>	<b>3</b>
<b>Performance Test Strategy</b>	<b>3</b>
Tool	3
System under test	4
Test Scenario	4
Load Profile	5
Monitoring	5
<b>Performance Test Results</b>	<b>6</b>
Interesting cases	7
<b>Summary</b>	<b>8</b>
Optimal service operating conditions, throughput, and latency distribution	9
Service limits	9
Stress testing result	9
<b>Future Performance Testing plan</b>	<b>9</b>

## Simple Task Queue Processing Service

A naive task queue processing system service. The service accepts different tasks and executes them asynchronously, ideally depending on the available resources. The task executions and results are persisted (by default on the local file system).

There are only two types of tasks in the system:

- **Compute factorial:** The result of this task is just a factorial.
- **Fetch content by provided URL using HTTP:** The result of this task is raw HTTP resource content.

The service exposes three API endpoints over the HTTP protocol:

- `POST /tasks/submissions`. This endpoint accepts (or rejects) a new task submission.
- `GET /tasks/submissions/{id}`. Returns the current status of the submitted task.
- `GET /tasks/{id}`. Returns the final result of the submitted task when it completes. The task is considered completed when it has transitioned into a `SUCCEEDED` or `FAILED` status.

## Performance Test Goal

The goal of this exercise is to run a black box performance test against the provided RESTful service instance and answer the following questions:

- In a steady state, what is the throughput for the service?
- In a steady state, what is the latency distribution for the service?
- What are the optimal service operating conditions?
- What are the service limits?

## Performance Test Strategy

This task involves performing non-functional performance testing of the service. The functionality and accuracy of the service responses are not tested.

## Tool

I've selected the Locust tool (see Locust documentation) for performing this task. It allows you to create a test scenario as a Python script, has a demonstrative web interface, and is easy to manage. Locust can monitor metrics such as RPS, Response Time (average, median, 90th percentile), and CPU usage. However, for more complex systems, I would likely use other tools that allow monitoring additional metrics.

The test script is located in the GitHub repository. The README file contains the test description and a guide on how to run the test.

## System under test

The load test and the service itself are both running on a local machine.

### Hardware Overview:

Model Name:	MacBook Pro
Model Identifier:	MacBookPro15,2
Processor Name:	Quad-Core Intel Core i5
Processor Speed:	2,4 GHz
Number of Processors:	1
Total Number of Cores:	4
L2 Cache (per Core):	256 KB
L3 Cache:	6 MB
Hyper-Threading Technology:	Enabled
Memory:	8 GB

## Test Scenario

Under normal conditions, tasks from a particular client are expected to arrive every 5 seconds. Most tasks are expected to involve HTTP fetching. The endpoints to get task statuses are expected to be called frequently (at least one call per task in progress every second). The endpoints to get task results are expected to be called rarely (at least once per task upon completion).

The Locust file with the test script emulates user behavior:

1. Create a new task ('http-fetch' or 'factorial' in a ratio of 5 to 1).
2. Check the task status until it is either SUCCEEDED or FAILED. If the GET /tasks/submissions/{id} response contains a task status of RUNNING or QUEUED, another request is sent in 1 second to check the task status again. The new status is recorded in the log file.
3. Check the task result when the task is completed.

Locust allows specifying task weight using the `@task` decorator. In this case, I used `@task(1)` for 'factorial' tasks and `@task(5)` for 'http-fetch' tasks.

### Requests distribution by type

Request Type	Weight
POST /tasks/submissions - 'http-fetch' type	25%

POST /tasks/submissions - 'factorial' type	5%
GET /tasks/submissions/{id}	45%
GET /tasks/{id}	25%

## Load Profile

To define the service's capacity and throughput in a stable state, I selected a stepping load profile. It allows monitoring service performance with varying user loads and probing the maximum RPS that can be achieved. Locust's web interface allows you to manually adjust the number of users and the spawn rate during a test. I started with 40 users and every 10 minutes added 40 more users to monitor service behavior and metrics.

The test is run in distributed mode—4 slaves for each processor core.

The test continues until one of the following conditions occurs:

- The service throws errors (on any of the POST or GET requests).
- CPU (load average) exceeds 3.00.
- RPS stops growing or decreases with increased load.
- Response time increases (even if RPS is growing and there are no errors).

## Monitoring

### Main monitoring sources:

#### - Test script logs

I added additional messages to the test script to monitor test performance. One of the most useful logs is monitoring task status: if the GET /tasks/submissions/{id} response contains a task status of RUNNING or QUEUED, another request is sent in 1 second to check the task status again. The new status is recorded in the log file, allowing for monitoring of the service queue.

#### - Errors

I used Locust's built-in option to mark GET /tasks/submissions/{id} requests as failing when the task status is FAILED, even if the response code is 200. Also, GET /tasks/{id} is marked as failing if the task result is FAILED, with the result added to the error message. Error messages can be seen in the Locust Web UI Failures tab.

#### - RPS (Requests Per Second)

RPS is one of Locust's main metrics and is shown in the Web UI in real-time when the test is running.

## - Response Time (ms)

Response time is also shown in the Web UI and is measured by percentiles: 50% (Median), 95%, and Average. More percentiles (99%, 99.9%, 99.99%) can be found in the CSV report available for download in the Download tab.

## - CPU

I used `htop` in the Terminal command line and the Activity Monitor application for CPU monitoring.

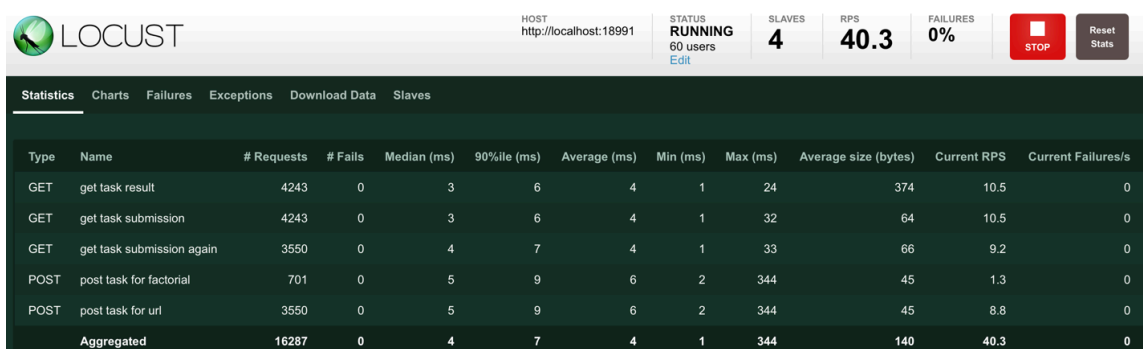
## Performance Test Results

I've added the test results to an aggregated table.

Test duration (min)	# of concurrent users	RPS (Avg)	Median (50%) Response time (ms)	95% Response time (ms)	Failures (%)	CPU (LA)
10	20	14	3	6	0	1.50 1.68 1.74
10	40	26	3	6	0	1.95 1.86 1.77
10	60	38	3	6	0	1.76 1.95 1.89
10	80	54	4	8	0	2.27 2.37 1.96
5	100	64 => 0.1	4	9	Queued tasks, no errors. Stopped test	2.77 2.38 1.64
5	120	78 => 0	5	8	Queued and failed tasks. Stopped test.	3.25 2.32 1.84
1	80 => 200 => 60	60 => 90 => 0 => 40	7	8	Queued tasks (not failed if decrease load)	2.15 2.17 2.06

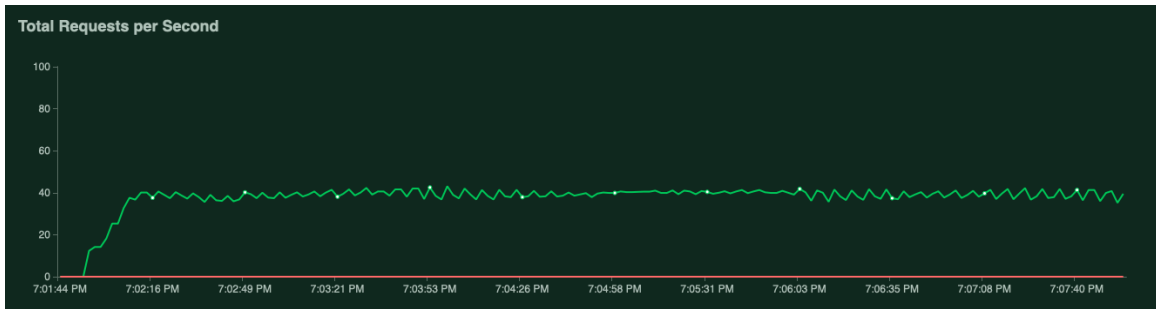
Based on the performance testing results, I would say the optimal service throughput is 40-50 RPS, which can be achieved with 80-100 concurrent users. More details are provided below.

### 1. Statistics table with metrics counted in Locust Web UI.

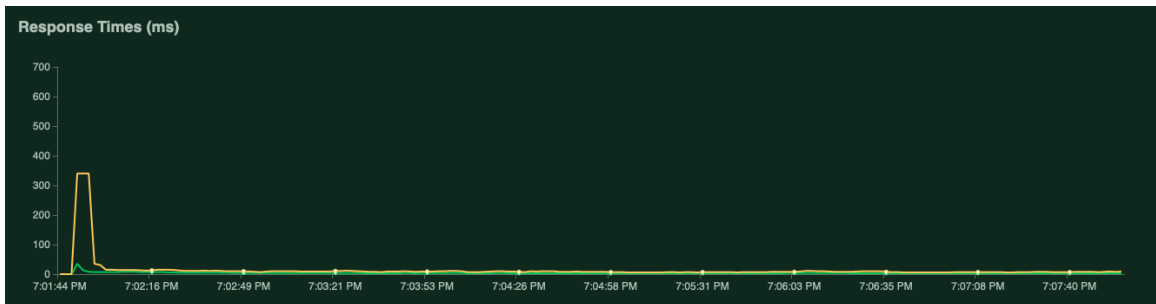


Type	Name	# Requests	# Fails	Median (ms)	90%ile (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS	Current Failures/s
GET	get task result	4243	0	3	6	4	1	24	374	10.5	0
GET	get task submission	4243	0	3	6	4	1	32	64	10.5	0
GET	get task submission again	3550	0	4	7	4	1	33	66	9.2	0
POST	post task for factorial	701	0	5	9	6	2	344	45	1.3	0
POST	post task for url	3550	0	5	9	6	2	344	45	8.8	0
Aggregated		16287	0	4	7	4	1	344	140	40.3	0

## 2. RPS graph



## 3. Response Time graph



## Interesting cases

### - Memory leak issue

During the test, I discovered a memory leak problem. After running the test for a long period, all tasks started to fail with the result: "too many open files in the system." I tried different load profiles:

- Stepping load
- Sharp load increase
- Endurance testing with 60% of the max handled load

All of these strategies eventually led to a memory leak problem. Therefore, this issue needs to be resolved before repeating the performance testing.

### - Running the test only for 'factorial' tasks

In this scenario, the service worked without issues with 300+ concurrent users. No memory leak issue was found.

### - Running the test only for 'http-fetch' tasks

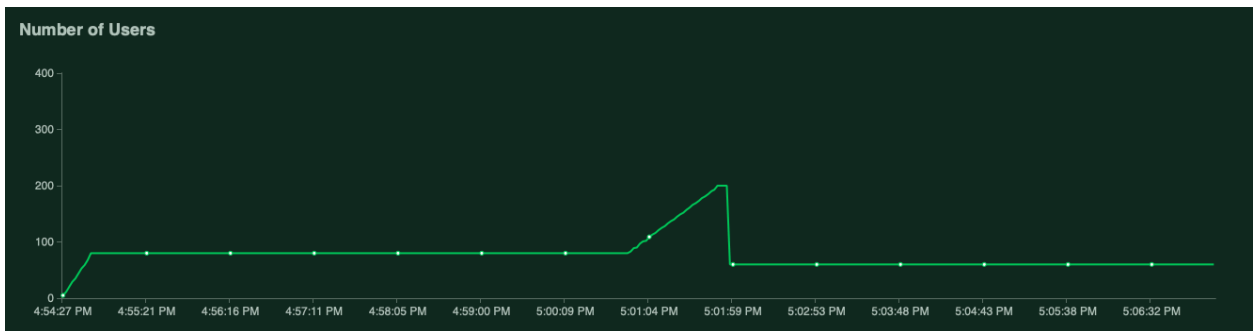
In this case, the service failed with a memory issue.

### - Stress testing

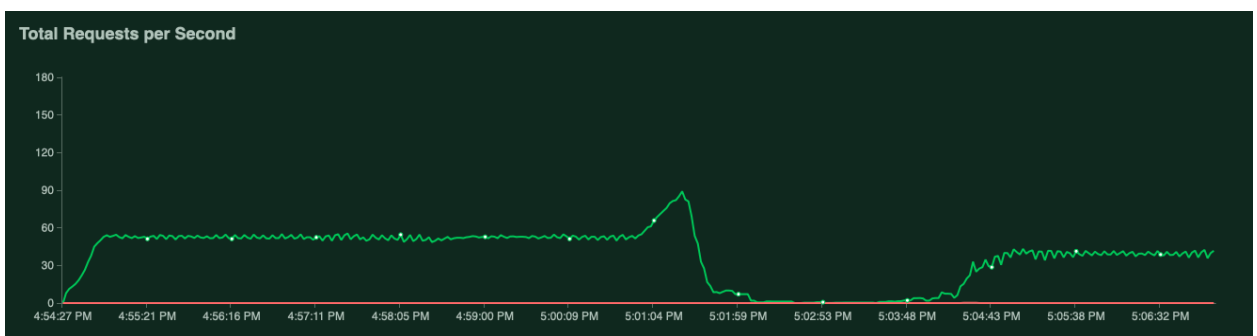
I started with 80 concurrent users (60 RPS) and then increased the load to 200 users. As a result, tasks were queued, and RPS initially increased from 60 to 90, then decreased to 0. I decreased the number of users to 60, and after some time, the service recovered, with RPS stabilizing at 40 (which is normal for 60 users).

**Main takeaway:** The load was increased for only about 1 minute. If I had maintained 200 users, tasks would have started to fail, and errors would have appeared. However, response time did not increase significantly.

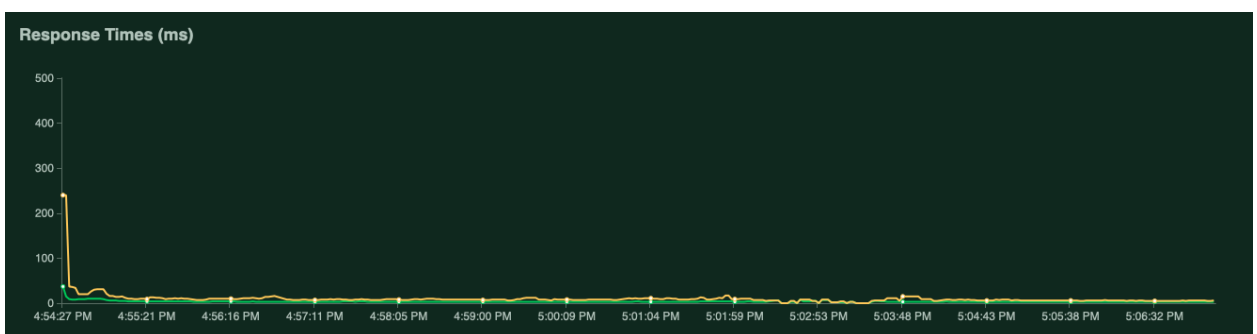
### 1. Number of users graph



### 2. RPS graph



### 3. Response Time graph



## Summary

Based on the performance testing results, here are the answers to the goal questions:



## Optimal service operating conditions, throughput, and latency distribution

- **Service throughput in a steady state:** 40-60 RPS (60-100 concurrent users). However, I wasn't able to verify how the service handles such a load over an extended period due to the memory leak issue.
- **Latency distribution (aggregated for all request types) in a steady state:**
  - 50th percentile – 3 ms
  - 95th percentile – 6 ms

To summarize, the service is stable when no more than 100 concurrent users are creating tasks and retrieving task results. The service can also handle short-term workload increases (up to 110-200 users for 1-5 minutes).

## Service limits

- With RPS > 70 (100+ concurrent users), tasks are queued, and the service cannot handle all of them. Eventually, tasks receive a FAILED status.

## Stress testing result

When the load is extremely increased (e.g., from 80 to 200 users), the following errors occur:

- Created tasks are queued and receive a FAILED status if the load persists.
- New tasks are not created – a 503 status code is returned in the response.

## Future Performance Testing plan

### - Endurance testing

Check how the service handles constant load (approximately 60% of average load in production or the maximum defined load in staging) over a long period (usually 2-3 days of continuous running).

### - Scalability testing

Validate the service's capability to scale up or down in response to the number of user requests or other performance attributes. This can be conducted at a hardware level (by using a more powerful machine) or a software level (with code optimization).

### - Failover testing

Validate whether the system can allocate extra resources and shift operations to backup systems during a server failure.