YouTrack Performance Task

# Application setup

Server: MacBook Pro 2,3 GHz Quad-Core Intel Core i7 (Hyper-Threading enabled), 32GB RAM  
Load Generator (LG): MacBook Pro M3 (5 performance / 6 efficiency cores), 18GB RAM

YouTrack: ZIP installation (ARM arch not supported, Docker installation on Mac is not easy to backup/restore due to filesystem access limitation)

# Users setup

99 users (as 1 user is reserved for admin and license limit is 100) created using k6 users.js script

# Task #1: Issues upload

Data parametrization  
  
10.000 descriptions (up to 200 symbols) / summaries (up to 10 words) text prepared and stored in CSV format ensure data variety (e.g. search performance might be dramatically impacted by low data cardinality).

## Creation scenario

Although issue creation scenario executed from UI (using web-browser) triggers bunch of different API calls (e.g. /drafts, /sprints, /issueWatchers, etc), there’s a programmatic way to create a new issue using simple REST API call:  
  
<https://www.jetbrains.com/help/youtrack/devportal/resource-api-issues.html#create-Issue-method>

**As far as task challenges us to ‘upload 100k issues as fast as possible’, we’re going to stick to that simplistic scenario.**  
  
Please, note that real-world scenario would imply much more intensive load model to replicate browser behavior.

## Open vs Closed Load model

Since we need to ensure stability of issues creation (not to overload system causing failures), **closed load model better fits the case as it provides sort of back pressure mechanism** (each thread iteration will be delayed in case of service time degradation, stalling further requests initiation).

## Capacity

Ramp-up load test executed to check potential maximum throughout for issues creation.

A graph with a line

Description automatically generated

Maximum throughput reached ~30rps, caused by CPU exhaustion. It may look not completely convincing due to CPU utilization ups & downs as there’s a visible pattern of CPU spikes every ~1m or ~1.5m (potentially caused by some background scheduled task):  
  
- Last 7m of the test:

A screenshot of a graph

Description automatically generated

Thus, to ensure stable rate of issues creation (even during CPU spikes), we should stick to a rate of slightly less than **30rps**.

**Theoretical time** in that case would be 100\_000 r / 30rps = 3333s ~= **1h**

However, there’s no guarantee that system will demonstrate similar performance as number of issues in the system grows (e.g. indexes update and data insertion might become more expensive). Let’s find it out by trying to create 100\_000 issues.

## Upload 100\_000 issues

In order to create exactly 100k issues we’re going use purpose-built k6 executor: <https://k6.io/docs/using-k6/scenarios/executors/shared-iterations/>

I’ve started from ~27k issues in the system with a test of 35 virtual users (1s think time on each iteration), resulting in ~30rps:

A close-up of a number

Description automatically generated

A graph of a graph

Description automatically generated with medium confidence

My initial assumption of system degradation depending on data volume was right. As we can see, service time increases linearly as data grows in the system. That means each issue creation operation becomes more and more CPU intensive:

* How it started:  
  A screenshot of a graph

  Description automatically generated
* How it ended:

A screenshot of a graph

Description automatically generated

So I was forced to stop the test and decrease number of virtual users to decrease pressure on the system (to avoid failures due to timeout and system outage):

A graph with blue and green lines

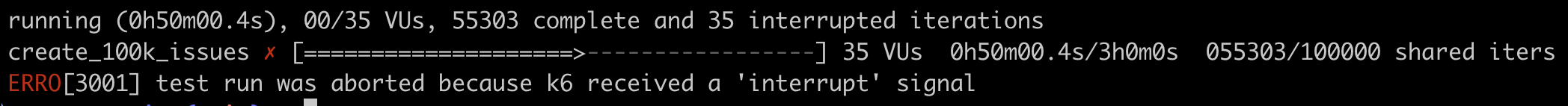
Description automatically generated

Overall, we can see that **issue creation throughput has decreased** from **~30rps** when there were **27k issues** to **~5rps** when there were **127k issues**.

It took 50m + 110m = **170m in total to create 100k issues** (from 27k to 127k), which is ~3 times higher than estimated based on initial throughput.

A close-up of a number

Description automatically generated





# Task #2: Users load scenario

## Load model

|  |  |  |  |
| --- | --- | --- | --- |
| **Operation** | **requests #, 1 user / h** | **requests #, 100 users / hour** | **requests #, 100 users / min** |
| Create issue | 1 | 100 | 2 |
| Update issue | 9 | 900 | 15 |
| View issue | 30 | 3000 | 50 |
| Search | 10 | 1000 | 17 |

As each user spends major part of the time in idle, it’s more efficient from load generator utilization standpoint to rely on requests throughput rather than number of users in the system (though, generally there’re cases when we need to ensure total amount of concurrent user connections/session, especially in stateful services). For a test task purpose only I’m going to stick to throughput-based model.

Open load model executor provided by k6 <https://k6.io/docs/using-k6/scenarios/executors/ramping-arrival-rate/> is a good fit to ensure specific load rate per operation.

Thus, reaching desired load rates will indicate that system can handle 100 users operating concurrently. We can also increase load rates linearly by using *X\_LOAD* multiplicator to find out system capacity (additional buffer).