YouTrack Performance Task Report

# 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**.

Note: there are 4 physical cores available on the server machine, however monitoring detects 8 vCPUs due to Intel’s hyper-threading mechanism.

**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

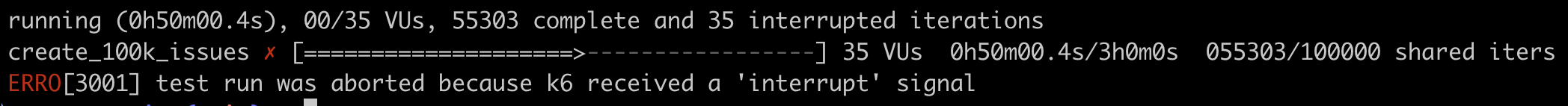
Description automatically generated

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 = **160m in total to create 100k issues** (from 27k to 127k), which is almost 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

Based on task description, I’m going to use following operations distribution as a base (with only assumptions that 9/10 issue ‘write’ operations are updates and 1/10 is new issue creation, and for each viewed issue user navigates to Issues page first).

|  |  |  |  |
| --- | --- | --- | --- |
| **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/issues | 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 operations 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).

## Endpoints selection

Although the task clearly defines limited load model we should stick to, there’s still a problem of endpoints selection to be included into a test scenario. Generally, it’s easier to build load model based on business scenarios/actions rather than specific endpoints, but similar criteria could be applied to individual endpoints as well.  
  
Usually there are several key factors to consider when creating a load model:

* **Frequency of occurrence**: most used operations/endpoints tend to make more performance impact;
* **Business criticality**: any business-critical actions should be load-tested to ensure normal product operation (e.g. if payment in online shop doesn’t work, it makes almost no sense; thus, it should be checked). Examples of such operations in case of issue tracker may include, but not limited to, issue creation / update / view;
* **Load intensity**: some operations may cause heavy load impact on the system even if they are used not as frequently, e.g. reports & statistics generation.

However, there might be additional minor factors to keep in mind, such as:

* **Maintainability & complexity of the load script**: whenever we’re trying to reach higher test coverage, we’re increasing complexity of the script and decreasing it’s maintainability. In the worst case, performance team may spend too much time during release cycles doing script updates rather than valuable work such as analysis & optimizations;
* **Operations/data “balance”**: some modifying actions made in the system should be counter-balanced by applying reverting actions to avoid malfunctioning, e.g. closing recently opened issues per user as users do not tend to open 1k+ issues tabs. Although it might be useful to know if such behavior can really lead to a system instability or outage, it makes load model unrealistic and it makes practical conclusions from tests harder to make (will such a problem ever occur in production or not?). That means if we add recently opened issue we should probably close it later to avoid data collecting over test iterations;
* **Caching**: some resources (not only html/css/js static files) might be cached on client side (example – *GET /api/config*). For the systems where there aren’t many ‘fresh’ users such calls might be skipped.

From a practical standpoint, we should balance between coverage and script maintainability/complexity with help of the key factors described above.

Let’s try to apply that criteria and define crucial calls/endpoints within our main user operations in YouTrack (as it tends to send a bunch of API requests per each transaction/operation).

### Business criticality

Specific endpoints that are playing main role in user action should be included into a script.

Examples of such endpoints:

* *GET /api/sortedIssues, POST /api/issuesGetter* while viewing Issues page;
* *GET /api/issues/<issue\_id>* while viewing/previewing particular issue;
* *POST /api/users/me/drafts, GET /api/users/me/drafts, POST /api/users/me/drafts/<id>, POST /api/issues?draftId=<id> for creating new issue;*
* *POST /api/commands?fields=issues(idReadable)* while applying commands;
* *PUT /api/issues/<issue\_id>/draftComment, POST /api/issues/<issue\_id>/draftComment, POST /api/issues/<issue\_id>/comments* while commenting issues (or using commands);
* *GET /api/sortedIssues?query= , POST /api/issuesGetter* while doing search;

### Frequency of occurrence

YouTrack UI seems to send a bunch of different requests (and some of them are repeated multiple times) for every user action:

* *GET /api/issues/<issue\_id>/sprints, GET /api/issues/<issue\_id>/links, GET /api/issues/<issue\_id>/activitiesPage, GET /api/issues/<issue\_id>/issueWatchers, POST /api/issuesGetter, POST /api/users/me/recent/issues, POST /api/issuesGetter/counts* while viewing/previewing particular issue;
* *POST /api/search/assist* while doing search;
* *POST /api/commands/assist* while applying commands;

There are many other calls made from UI while doing user actions (some of them are triggered from different pages, e.g. *GET /api/inbox/folders, GET /api/permissions/cache, GET /api/filterFields/values, GET /api/users/me* and others. For the task solution purpose, I’m not going to include them, **but on** **real projects they still should be at least considered**. Even if every particular call doesn’t seem to make any difference, it’s the total amount of them which may impact performance. Generally, we can either check their load intensity by providing synthetic test with such calls, or we can increase load rates of other operations in test to counter-balance the fact of partially missing load/traffic.

### Load intensity

As mentioned above, let’s try to assess an impact from some calls that seem to be neglectable. To do that, we can create a simple synthetic load test containing some of these calls to see how many of them system can handle:

* *GET /api/inbox/folders?$top=-1&fields=id,lastNotified,lastSeen,enabled&ignoreLicenseErrors=true&start=<timestamp>*
* *GET /api/permissions/cache?fields=global,permission(key),projects(id)*
* *GET /api/filterFields/project/values?$top=-1&fields=id,presentation,query&prefix=&query=&type=Issue*
* *GET /api/admin/widgets/general?fields=id,key,appId,description,appName,name,collapsed,indexPath,extensionPoint(),iconPath,appIconPath,expectedHeight,expectedWidth*

*A graph with a line and a line

Description automatically generated with medium confidence*

System can **easily handle >800rps** with such ‘minor’ calls. Given that 100 users are viewing issues with a rate of ~1rps, there should be hundreds of these calls per single view to make any impact, which **allows us to ignore them**.

Interestingly, an impact of presumably a background task running every ~1mI mentioned earlier is visible even on such test by impacted response times.

In conclusion, I’d like to mention that I wasn’t following a goal of providing the most realistic/accurate load model, but rather tried to come up with a general framework of assessing endpoints/operations coverage.