YouTrack Performance Task Report

Table of Contents

Application setup

Two separate machines are used to host Service/Server and Load generator.

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

Second machine with ZIP installation was used for server since ARM arch is not supported (although it started successfully on Apple M3 chip ☺) and 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
https://github.com/yurysup/jetbrains-youtrack-test/blob/main/setup/users.js

Users data (username, email) are generated using Python Faker package https://faker.readthedocs.io/en/master/ in https://github.com/yurysup/jetbrains-youtrack-test/blob/main/setup/fake_users_data.py

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).

Same Faker package is used for data generation in https://github.com/yurysup/jetbrains-youtrack-test/blob/main/setup/fake_issues.py

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).
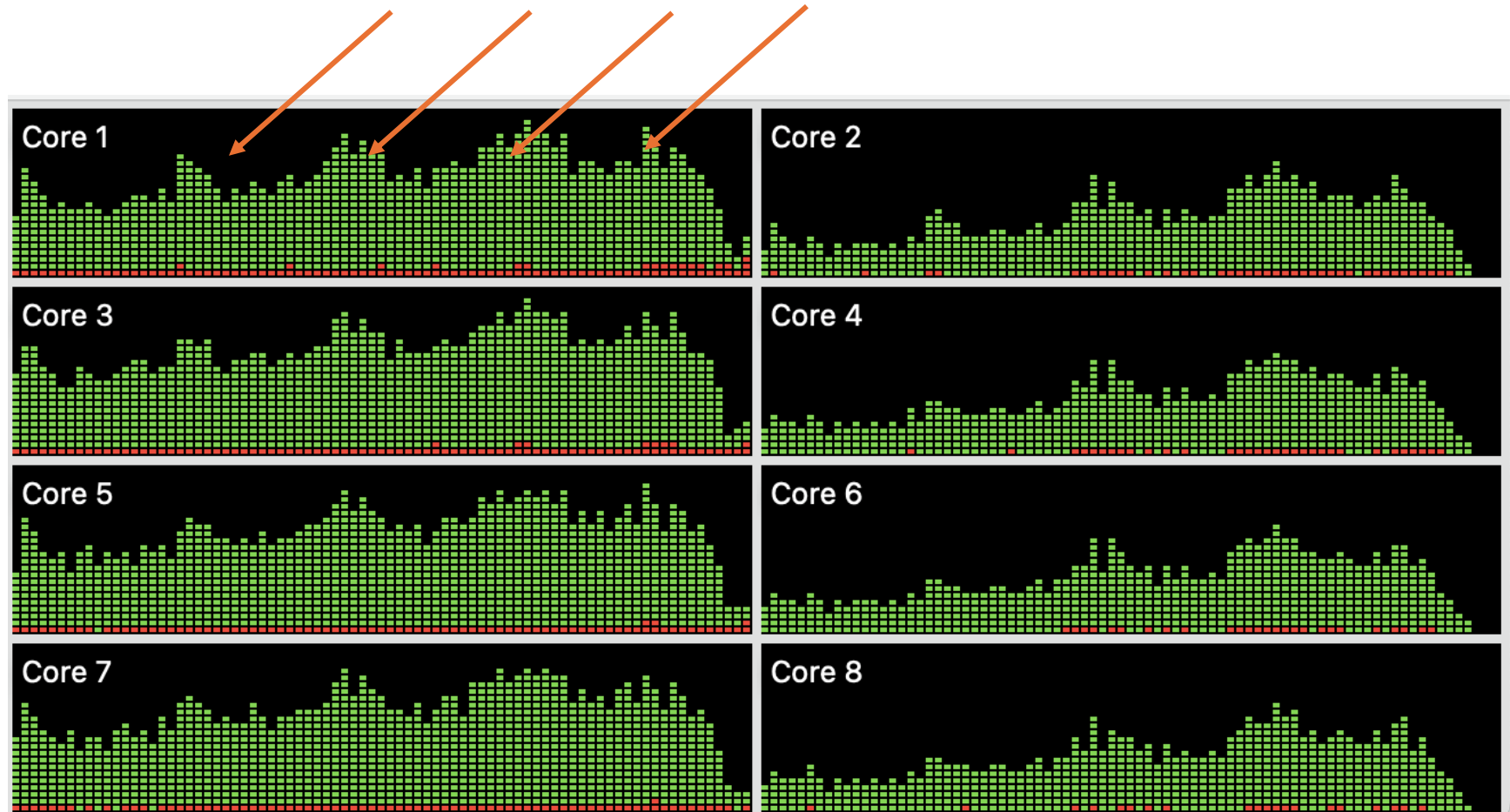
Throughput

Ramp-up load test executed to check potential maximum throughout for issues creation.
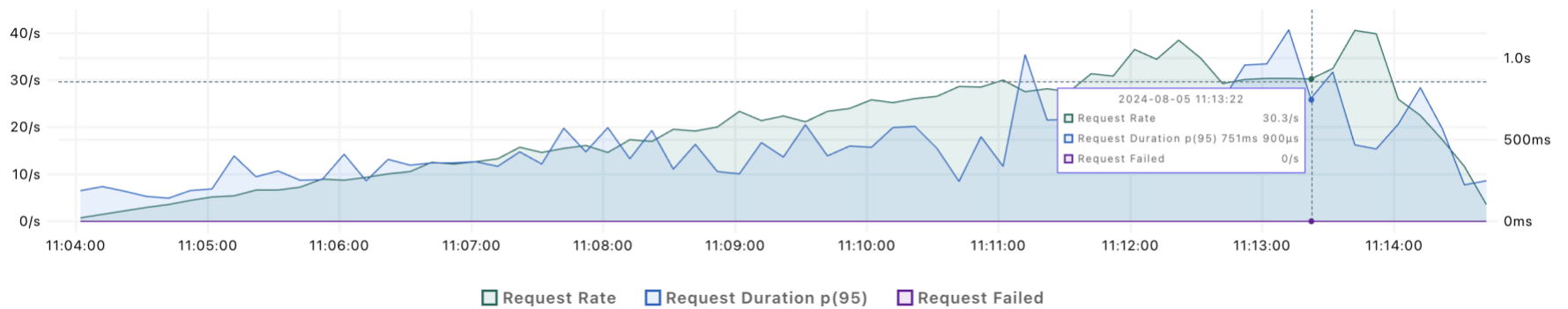
Maximum throughput reached ~30rps, caused by CPU exhaustion. It may look not completely convincing due to CPU utilization ups & downs as there's a visible pattern of CPU spikes every ~1m or ~1.5m (potentially caused by some background scheduled task):

- Last 7m of the test:

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/ , see https://github.com/yurysup/jetbrains-youtrack-test/blob/main/setup/issues.js

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:
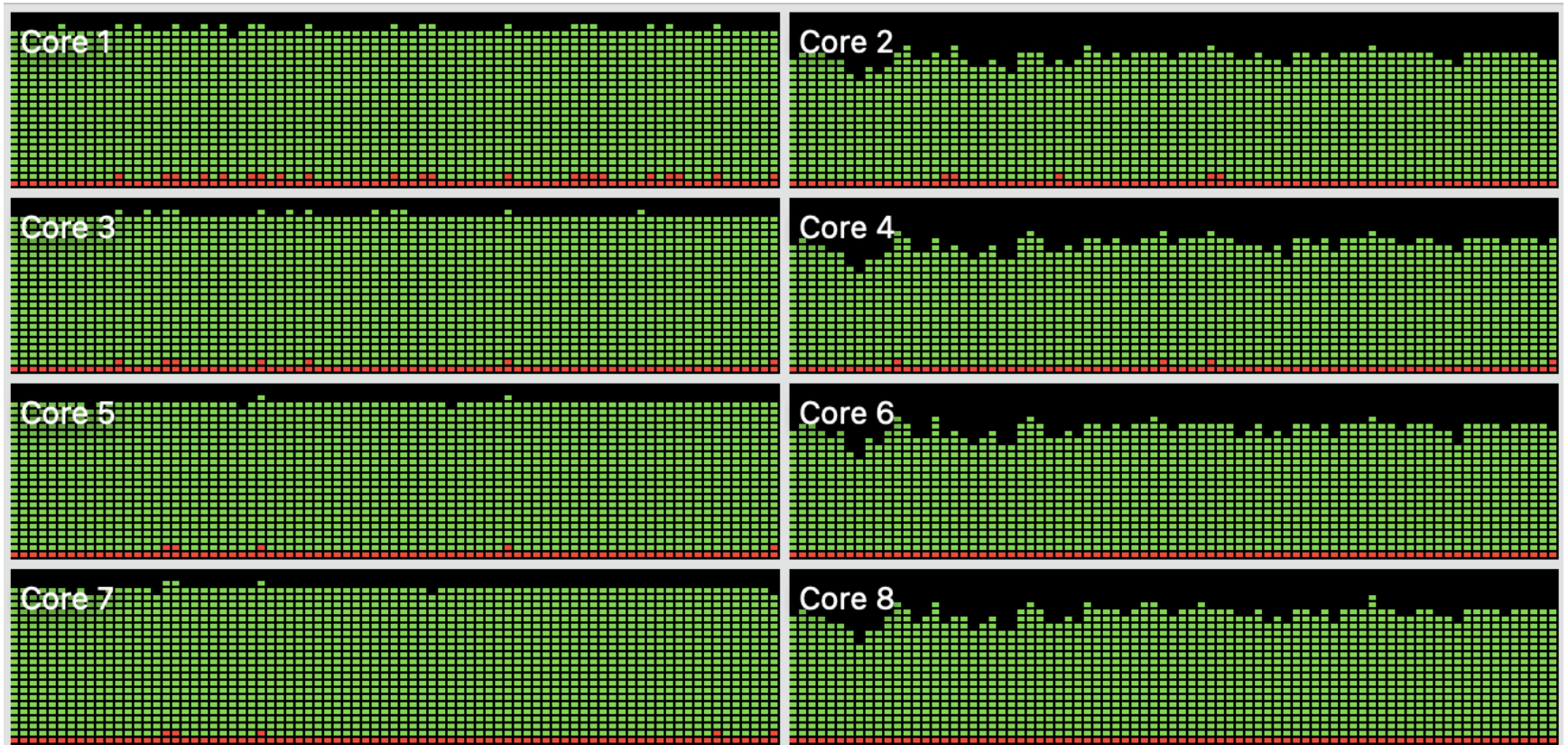
Issues 27,200
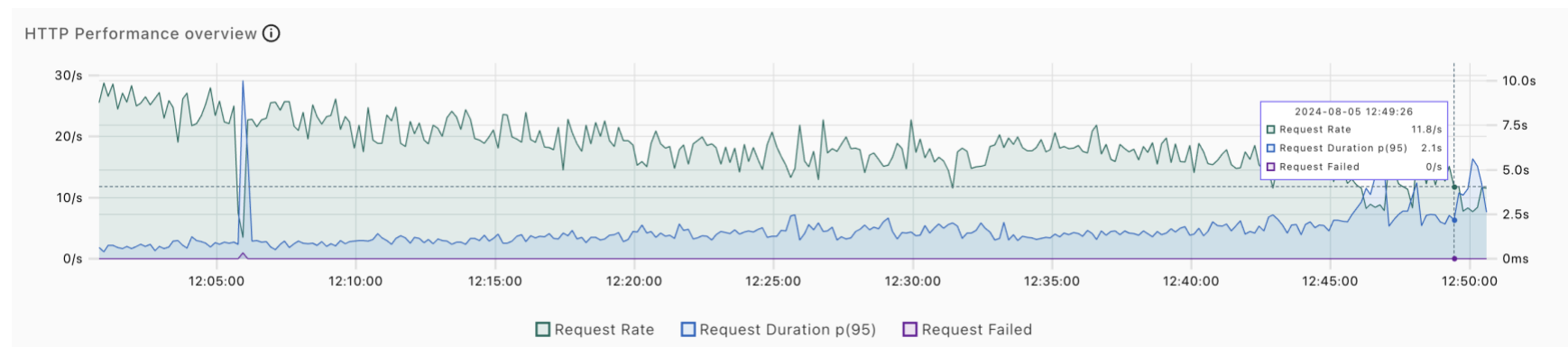
My initial assumption of **system degradation depending on data volume** was right. As we can see, service time increases linearly as data grows in the system (see issues throughput report). That means each issue creation operation becomes more and more CPU intensive:
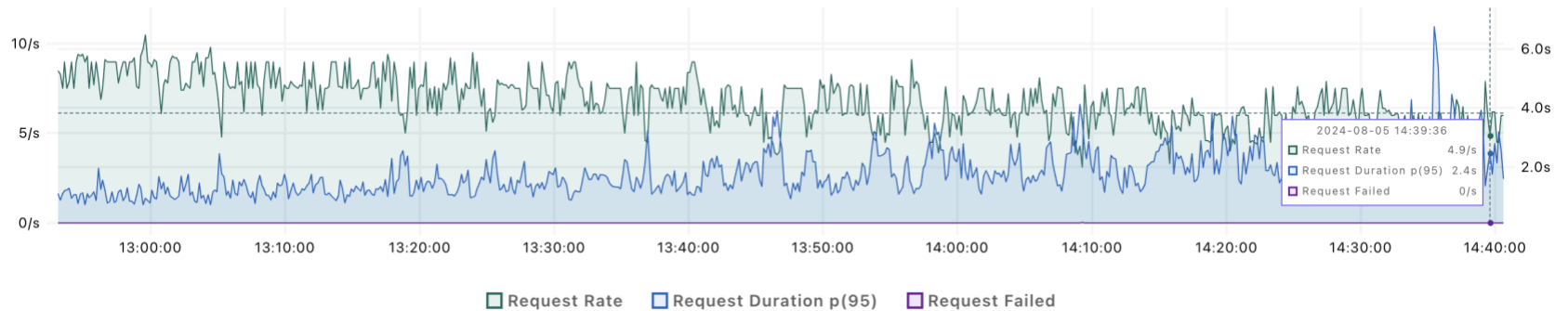
- How it started:
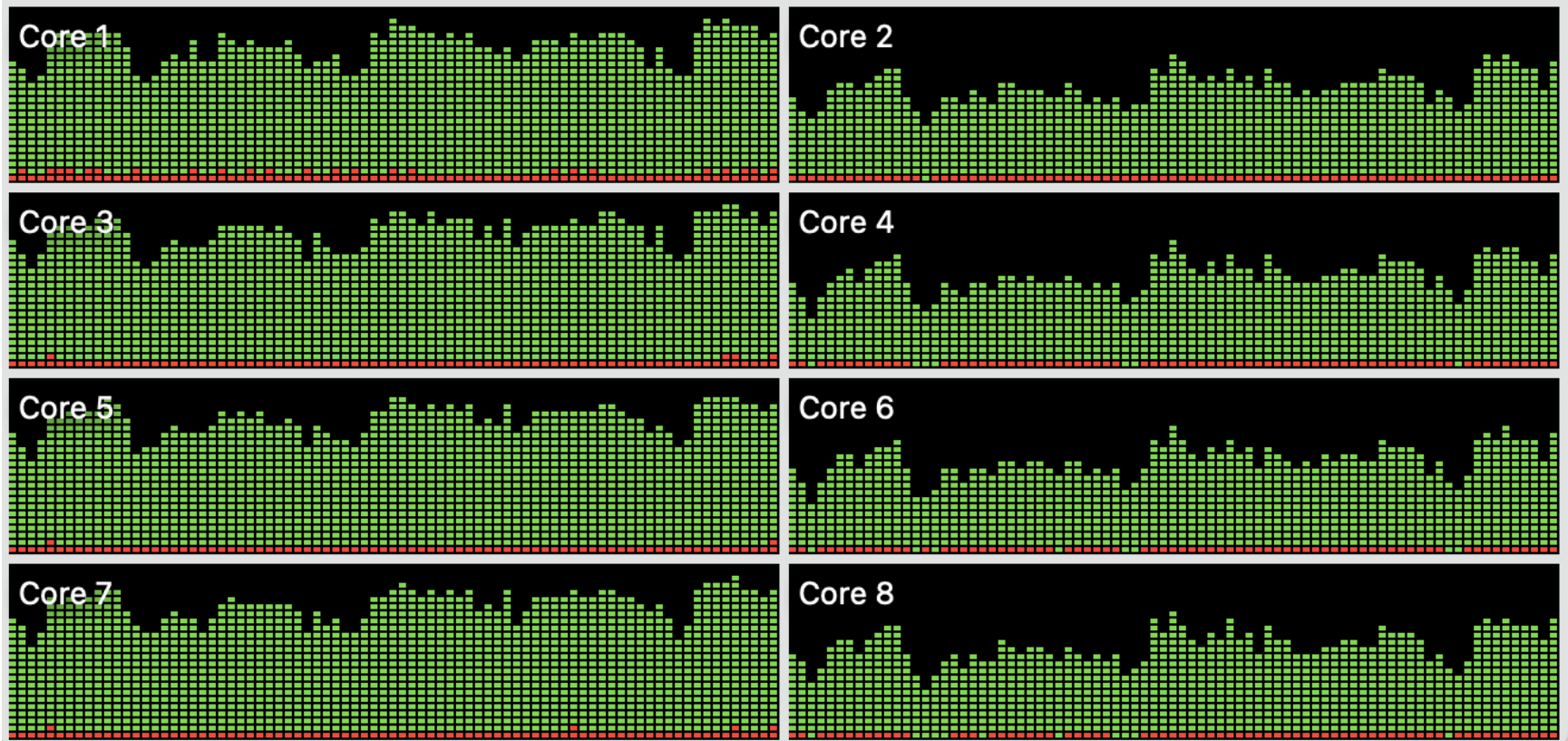
- How it ended:

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):

HTTP Performance overview

Request Rate | Request Duration p(95) | Request Failed

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.

Issues 127,230

```
running (0h50m00.4s), 00/35 VUs, 55303 complete and 35 interrupted iterations
create_100k_issues ✗ [==================>----------------] 35 VUs  0h50m00.4s/3h0m0s  055303/100000 shared iters
ERRO[3001] test run was aborted because k6 received a 'interrupt' signal
```

```
running (1h50m44.2s), 00/15 VUs, 44700 complete and 0 interrupted iterations
create_100k_issues ✓ [===================================] 15 VUs  1h50m44.2s/3h0m0s  44700/44700 shared iters
```

Q&A

Q: <u>How much it would take to 'upload' 100k issues</u>?
A: It took **160m (2h40m)** due to **decreasing insert performance as data volume grows**.

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). Besides of that, it's easier to emulate expected load rates directly rather than adjusting think times. <u>For a test task purpose I'm going to stick to throughput-based model</u>.

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 (where x1 equals to 100 users) to find out system's 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 <u>many other calls made from UI</u> 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 <u>task solution purpose, I'm not going to include them</u>, **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*

HTTP Performance overview ⓘ
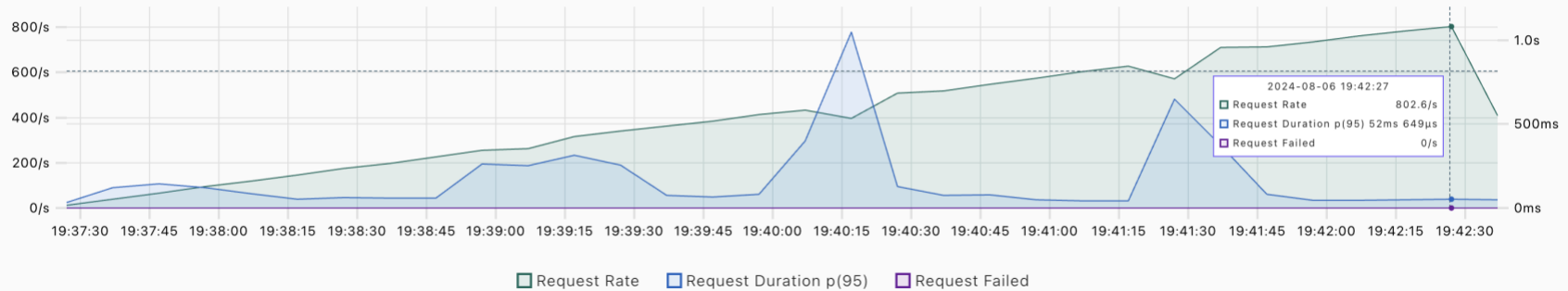
System can **easily handle >800rps** with such 'minor' calls. Given that 100 users are viewing issues with a rate of ~1rps, there should be hundreds of these calls per single view to make any impact, which **allows us to ignore them**.
Interestingly, an impact of presumably a background task running every ~1m I mentioned earlier is visible even on such test by impacted response times.

Resulting scenarios

- View all issues: *GET /sortedIssues, POST /issuesGetter/count, POST /issuesGetter*;
- View particular issue: *GET /issues*;
- Create issue: *POST /users/me/drafts, POST /users/me/drafts/<draft_id>, POST /issues*;
- Update issue: *POST /commands/assist, POST /commands*;
- Search issues (both attribute & text): *POST /search/assist, GET /sortedIssues?query=, POST /issuesGetter*;

See https://github.com/yurysup/jetbrains-youtrack-test/blob/main/youtrack.js

In conclusion, I'd like to mention that I wasn't aiming to providing the most realistic/accurate load model, but rather tried to come up with a general framework of assessing endpoints/operations coverage. Simplistic resulting set is relevant for a test task purpose only.

Test execution

Capacity

First, let's figure out the maximum capacity, e.g. find saturation point.

Although task explicitly declares 1000ms/500ms for POST/GET requests respectively, saturation point can't be defined by SLA only. Generally, saturation Is characterized by rapid increase of response times, inability to increase throughput and/or increase of error rate:

- Response time – R
- Throughput – X
- Resource utilisation - U
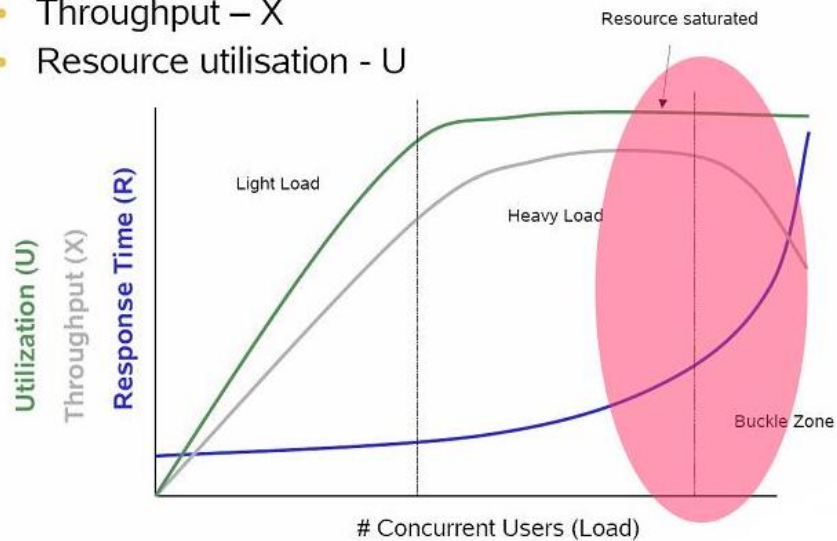
For capacity test I'm going to ramp the load up to X_LOAD=3 (equals to 300 concurrent users) over 30m and track response times/throughout/error rate.

```
scenarios: (100.00%) 4 scenarios, 160 max VUs, 30m30s max duration (incl. graceful stop):
         * create_issue: Up to 0.10 iterations/s for 30m0s over 3 stages (maxVUs: 20, exec: testCreateIssue, gracefulStop: 30s)
         * search_issues: Up to 0.85 iterations/s for 30m0s over 3 stages (maxVUs: 20, exec: testSearchIssues, gracefulStop: 30s)
         * update_issue: Up to 0.75 iterations/s for 30m0s over 3 stages (maxVUs: 20, exec: testUpdateIssue, gracefulStop: 30s)
         * view_issue: Up to 2.50 iterations/s for 30m0s over 3 stages (maxVUs: 100, exec: testViewIssue, gracefulStop: 30s)
```

After ~20m of the test run, I was forced to stop it as responses grew up to ~10s causing timeouts (set in k6 script):

HTTP Performance overview
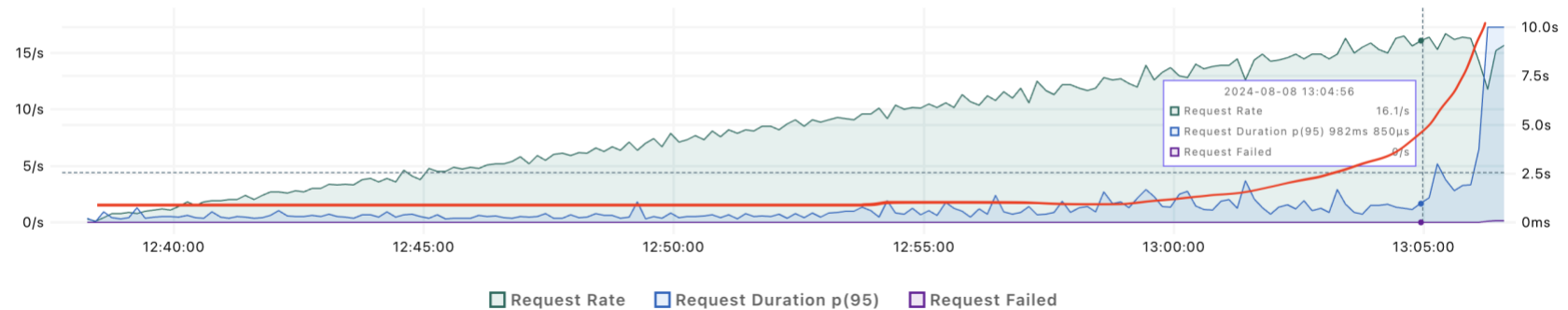
Saturation point seems to be reached ~ at 13:05 (**~28m** after test start or ~at **280 concurrent users**) as responses start to grow rapidly due to CPU resource exhaustion (last 7m of the test):
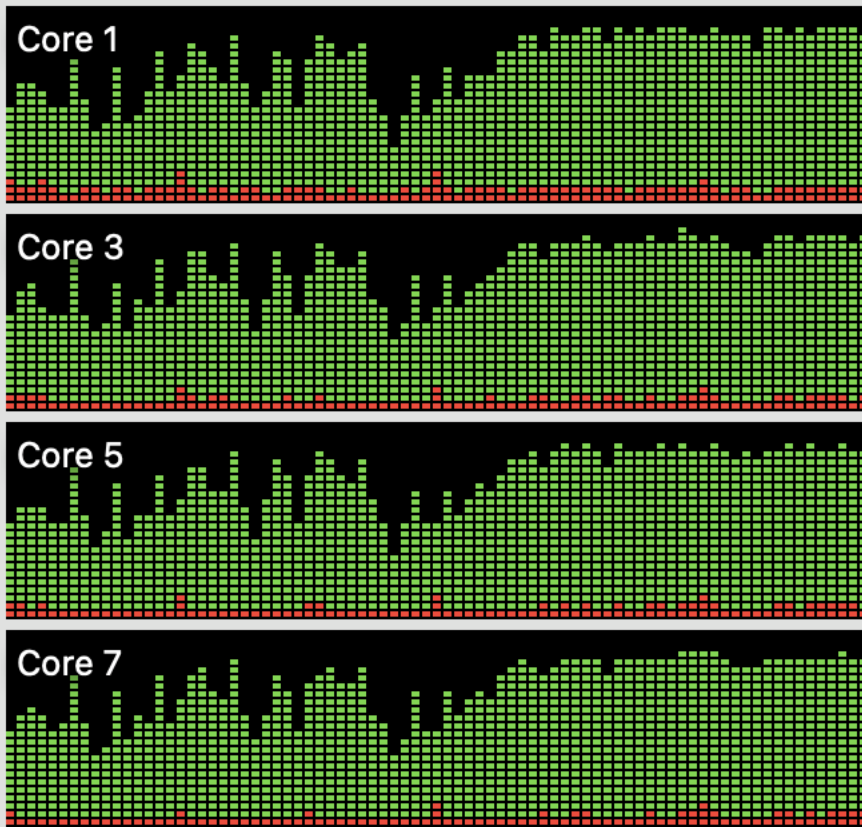
It gives us ~100% (x2) of safety buffer in terms of load system can handle (see capacity report). Response times started to visibly degrade at **~200 users**.

Meanwhile, load generator machine was hardly loaded:

Which double-confirms that the **bottleneck is on application side**.
Capacity limitation due to CPU resource can be investigated using Java profilers or DB profilers (depending on which component consumes most of the CPU time).
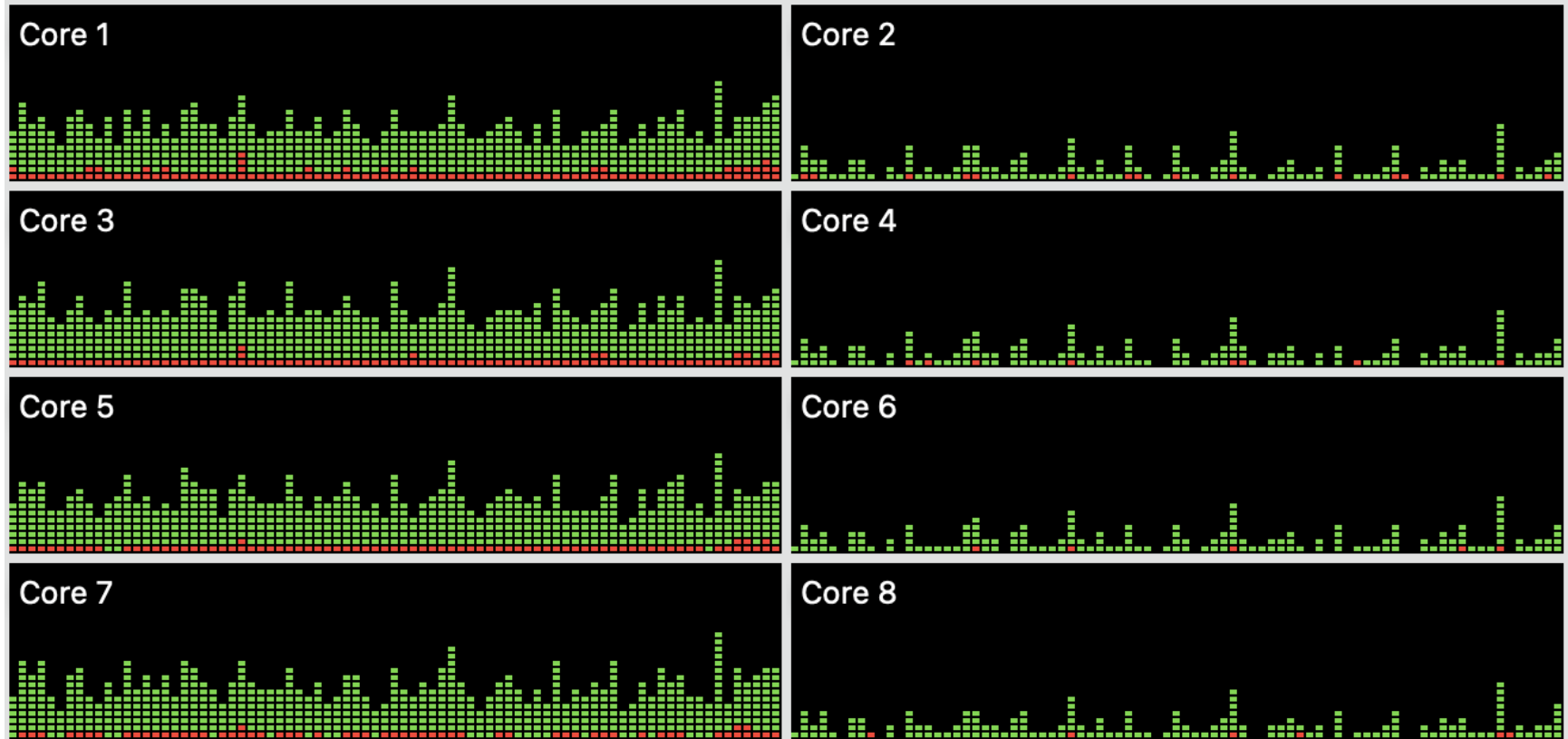
100 users fixed-load test

Now let's figure out if system can stay stable at expected peak load of **100 concurrent users** over longer time – **1h** as suggested by the task (see fixed-load report).
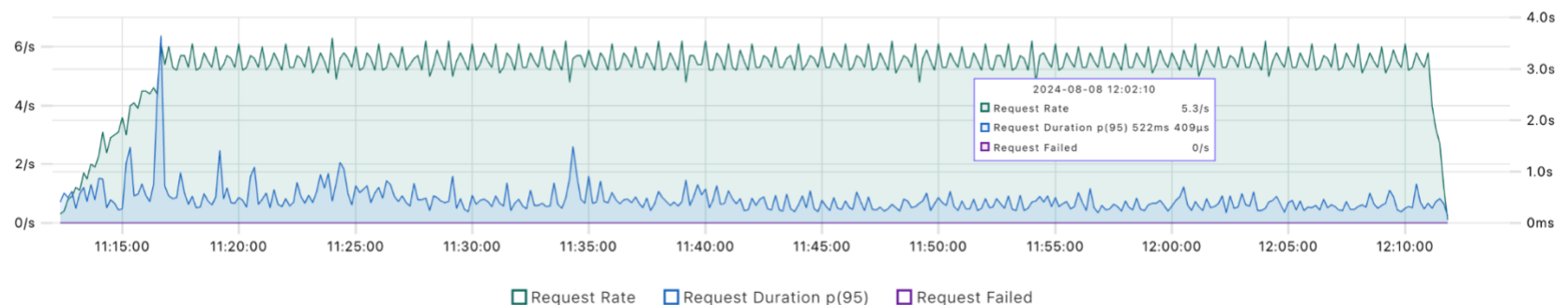
Response times were stable throughout the test apart from a few peaks. CPU utilization:

Let's assess **summary statistics**:

```
  http_req_duration..................: avg=130.13ms min=4.48ms   med=52.1ms   max=5.03s   p(90)=317.5ms  p(95)=461.77ms
    { expected_response:true }.......: avg=130.13ms min=4.48ms   med=52.1ms   max=5.03s   p(90)=317.5ms  p(95)=461.77ms
  ✓ { name:GET /issues }.............: avg=95.68ms  min=13.26ms  med=54.56ms  max=2.42s   p(90)=211.53ms p(95)=281.95ms
  ✓ { name:GET /search/assist }......: avg=92.58ms  min=11.76ms  med=52.87ms  max=753.37ms p(90)=219.16ms p(95)=292.65ms
  ✓ { name:GET /sortedIssues }.......: avg=46.51ms  min=5.65ms   med=22.96ms  max=2.58s   p(90)=88.68ms  p(95)=178.07ms
  ✓ { name:POST /commands }..........: avg=85.52ms  min=26.45ms  med=60.07ms  max=1.15s   p(90)=129.42ms p(95)=307.52ms
  ✓ { name:POST /commands/assist } ..: avg=72.32ms  min=5.86ms   med=42.5ms   max=879.6ms p(90)=174.7ms  p(95)=244.04ms
  ✗ { name:POST /issues }............: avg=1.09s    min=558.14ms med=1.03s    max=2.03s   p(90)=1.45s    p(95)=1.56s
  ✓ { name:POST /issuesGetter }......: avg=346.4ms  min=5.4ms    med=265.84ms max=5.03s   p(90)=603.14ms p(95)=803.77ms
  ✓ { name:POST /users/me/drafts } ..: avg=115.15ms min=28.18ms  med=92.67ms  max=427.23ms p(90)=206.14ms p(95)=264.7ms
✓ http_req_failed...................: 0.00%   ✓ 0          ✗ 18920
```

Thresholds were set to p90:1000ms for all POST requests and p90:500ms for all GET requests.
As we can see, **POST /issues (create an issue) requests have significantly crossed the expected SLA** - **1.45s**.

It may seem like we've chosen saturation point too high and gave wrong capacity evaluation, but in fact it's **caused by general slowness of 'create issue' operation due to high data volume**. It's visible even on idle system with single (1) user interaction, which means system will exceed SLAs in that case as well:

It means that system can handle more than 100 users without degradation, but each 'create issue' operation will still be slower than expected and should be addressed separately.

Generally, such **high response times need to be investigated/profiled to make API calls faster** (e.g. check traces if available, investigate underlying SQLs if any, etc). It may also decrease pressure on the system, and lead to capacity increase (although it's not guaranteed and depends on type of the bottleneck).

Q&A

Q: Can YouTrack tracker handle 100 concurrent users?
A: Yes, it **can handle more than 100 concurrent users** without significant degradation;

Q: What endpoints are crucial for load testing and what are not?
A: See **Endpoint selection** and **Resulting scenarios sections**;

Q: How much of safe-buffer system has in terms of load (capacity without significant degradation)?
A: System demonstrates **capacity of ~280 concurrent users with max workload of ~200 users without any visible degradation** in current setup;

Q: Can saturation point be detected in current setup or something is preventing us from doing it?
A: **Saturation point was successfully identified**. However, SLA threshold for specific endpoint was crossed from the very beginning of the test, which means system doesn't demonstrate desired performance in terms of how fast it operates from user perspective.

Additional Questions

Q&A

Q: In case of load generator and service sharing same server, how can performance impact be mitigated? What system resources are they sharing?
A: For **Docker installation**, **specific resources can be provisioned per container** - https://docs.docker.com/config/containers/resource_constraints/ (*docker run –cpus <cpu_count> --memory <memory_amount>*), while for **ZIP installation native OS mechanisms should be used**, e.g. *ulimits* & *cgroups* (which I assume are implicitly used by docker as well);

Q: In case of load generator and service being set on separate servers, how performance degradation of the service can be justified?
A: By **providing telemetry both for service and load generator** (see Test execution section);

Q: In case of Cloud installation, can in-built telemetry from admin console be used for performance degradation justification?
A: I haven't used Cloud installation, but I used **admin telemetry endpoint** for ZIP installation (https://www.jetbrains.com/help/youtrack/devportal/resource-api-admin-telemetry.html) and it **doesn't seem to provide much of useful data for bottleneck localization**

```
{
  "textIndexSize": "130.5 MB",
  "blobStringsCacheHitRate": "78.64%",
  "usedMemory": "980.1 MB",
  "databaseBackgroundThreads": 1,
  "pendingAsyncJobs": 0,
  "databaseQueriesCacheHitRate": "25.04%",
  "requestsPerSecond": "0.13",
  "$type": "Telemetry"
}
```