# GatherChain Performance Evaluation

João Almeida (42490) — 2021 — FCT-NOVA, DI


**Figure 1:** *Gatherchain logo*

## Abstract

In this paper, it is proposed a blockchain-based solution for version control of model-driven engineering artefacts. The goal is to facilitate collaboration in a multi-user area, like the education field, and track changes in a trusted and secure manner. This solution is based on using the **Hyperledger Fabric Network** to govern and regulate file version control functions among students and teachers. A thorough state-of-the-art is done in project management tools, modelling tools and authenticity technology. Research highlights many related work who try to solve similar issues. The risks and disadvantages of existing collaboration solutions are considered, as well as possible ways of their elimination.

**Keywords:** blockchain, version-control, performance testing, azure, REST API, bash

## 1 Introduction

This document serves as a guide to the performance evaluation done to the GatherChain solution. The questions we try to answer with the performance tests are:

1. Is the solution responsive enough for users not to be thrown off?
2. Can it handle peak loads of numerous students trying to communicate with the solution?
3. Is it costly for the faculty?

As it can be derived from reading the questions, questions number *2* and *3* are intrinsically related. The more load can the solution handle, the more it will cost.

## 2 Tests

The goal of this performance evaluation was to gather data about the duration of requests, resource utilization of the virtual machine hosting the blockchain server, the web app and the redis cache, and their peak load. Due to the goal in mind, the git and GitHub part of the solution was not tested in this way. In order to evaluate the solution's resources, the testing was made using the API server in the Web App hosted in the cloud. 6 different calls were made to the server:

- **Initialize network**: uses the api */init*. Responsible for the blockchain network initialization. It receives the password set during the solution's deployment;
- **Clear network**: uses the api */clear*. Responsible for clearing the cache's data and bringing down the blockchain network. It receives the password set during the solution's deployment;
- **User registration**: uses the api */registernumber*. Responsible for the user registration in the cache, and therefore the system. For testing purposes, it only takes the student number, and the group name, which in the beginning is 0;
- **Group creation**: uses the api */creategroup*. This call is responsible for the student's group creation. It receives the student number of the creator, the group's numbers, and 0000 as the first commit; After the group is created it uses the */registernumber* api call to change the group's field of the different members of the group;
- **Push commit**: uses the api */push*. It stores the transaction in the blockchain network in the specific group (channel). Receives the student number of the change's author, the group's id, and the hash of the commit (randomly generated);
- **Get history**: uses the api */history*. It receives the group's id. It outputs the transaction history of the group stored in the blockchain network.

To automate the testing process, 3 bash shell scripts were built:

- **test1.sh**: consists in 3 users registrations, 1 group creation and 4 commit pushes, in a sequential order. Gets the transaction history of the group;
- **test2.sh**: consists in 6 users registrations, 2 group creation and 12 commit pushes, in a sequential order. Gets the transaction history of both groups;
- **test3.sh**: consists in 6 users registrations, 2 group creation and 17 commit pushes, in a parallel order. Gets the transaction history of both groups;

These scripts can be found in the **GatherChain-Testing** repository, hosted on GitHub.

## 3 Results

### 3.1 Is the solution responsive enough for users not to be thrown off?

For question number *1* the results are divided in 6 different tests, 1 for each type of call. The data presented next consist in averages. Each test was executed 20 times. The graphics present the minimum, the average and the maximum times recorded for each test.

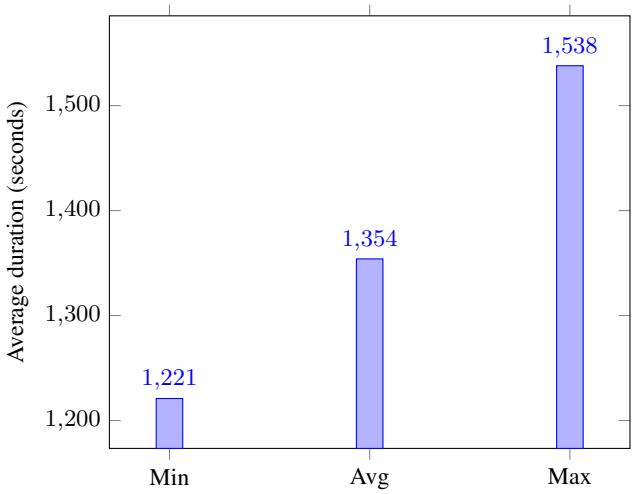
For developing and testing reasons, the solution was deployed with the most basic and less performant resources in Azure.

#### 3.1.1 ARM Deployment

This test consists in the duration it takes for the resources to be deployed in Azure Cloud.

**Table 1:** *Cloud resources used for testing purposes (question 1)*

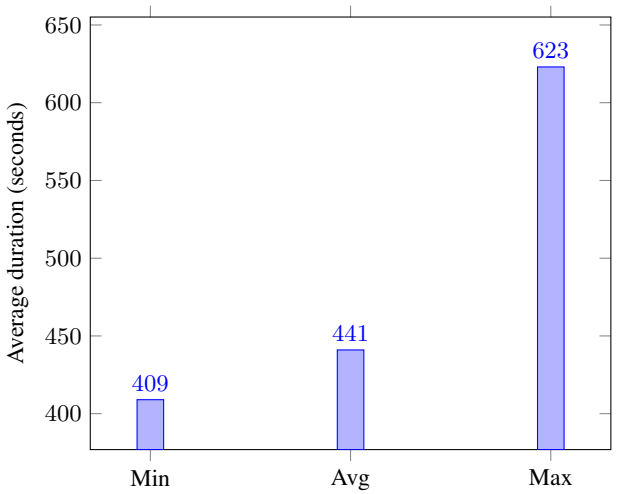
| Resource | Type | Details |
|---|---|---|
| VM Disk | SSD LRS | 500IOP/s, 60Mb/s bandwith |
| VM SKU | DS1 v2 | 1vCPU, 3.5Gb RAM |
| Web SKU | F1 | Shared vCPU, 1GB RAM |
| Cache SKU | C1 | 1vCPU, 500Mb/s bandwith |




The duration took anywhere between 20 and 25 minutes, with average falling on 22 minutes and 34 seconds. The resource which takes almost 90% of the deployment's duration is the Redis cache.

#### 3.1.2 Network Initialization

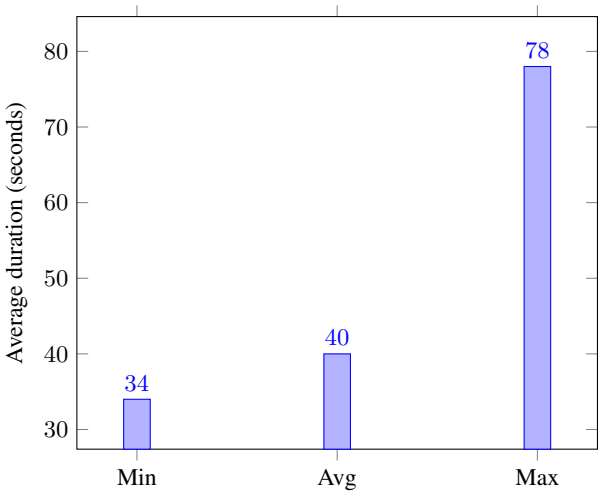
This test consists in the duration it takes for the blockchain network to be initialized in the virtual machine.




The network took between 6 and 10 minutes to be initialized, with the average being 7 minutes and 21 seconds.

#### 3.1.3 Shutting Down

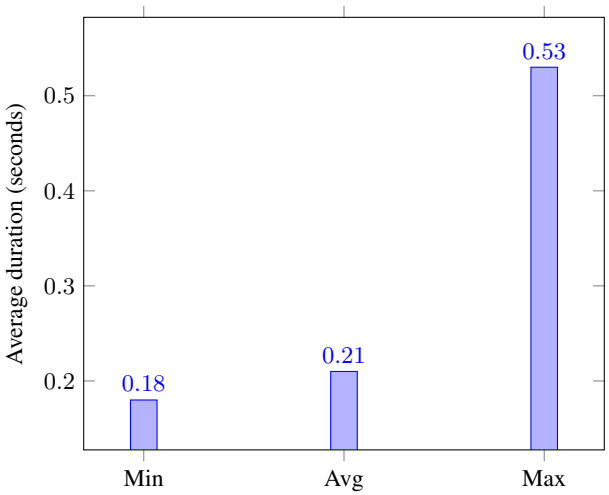
This test consists in the duration it takes for the blockchain network shut down and the redis cache to be cleared.




The network took less than 1 minute to be brought down and the cache to be cleared, with the average being 40 seconds.

#### 3.1.4 User Registration

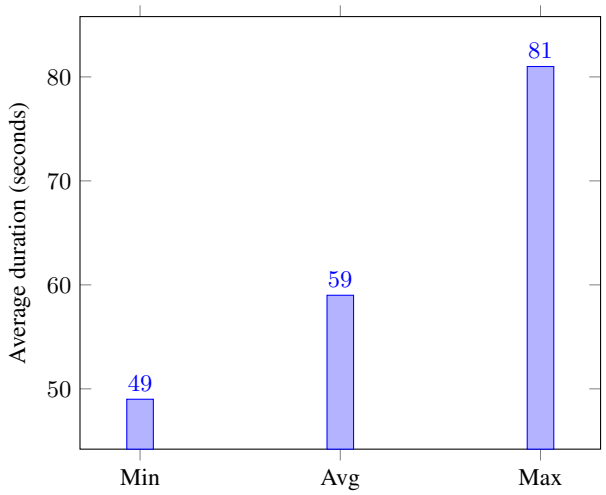
This test consists in the duration it takes for the user to be registered in the solution's cache.




This call doesn't communicate with the blochain network, and that's visible in the times gathered in the tests. Every user registration took less than a second, with the average being less than half a second, 0.21 seconds.

#### 3.1.5 Group Creation

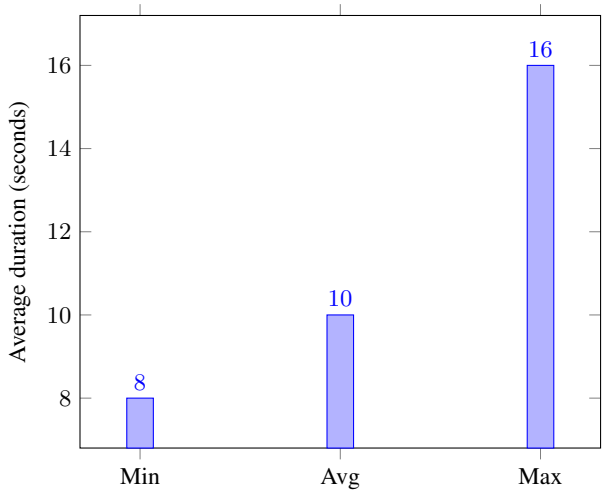
This test consists in the duration it takes for a group to be created in the solution.




The network took close to 1 minute to register a group, with the average being 59 seconds.

#### 3.1.6 Push Duration

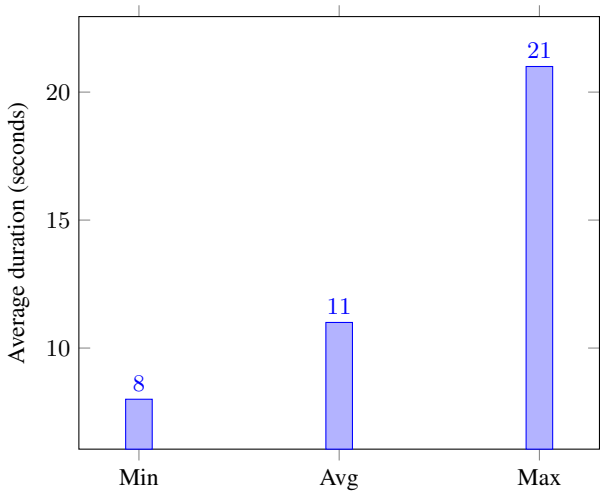
This test consists in the duration it takes for a commit hash to be commited to the blockchain network.




The network took in average 10 seconds to store a transaction.

#### 3.1.7 Get History

This test consists in the duration it takes for the solution to output the transaction history of a specific group (channel).




The network took in average 11 seconds to output the transaction history with the maximum recorded being 21 seconds.

As expected the commands that took more time are the administration commands of deploying the solution in the cloud and the blockchain network initialization. Luckily these commands should only be performed once in the beginning stages of the solution. The command that is performed more by the users is the *push command* which takes approximately 10 seconds to finish. Comparing to a git commit and push which takes approximately 3 seconds, it can be seen that there's a slight difference. Above all, in the prototype's solution, the *push command* also does a git commit and push, which adding to the recorded times it gives, 13 seconds. The same happens in the *group creation* command. In the prototype, this command also creates a local repository, a GitHub repository, adds, commits and pushes the initial environment and invites the collaborators. Adding everything up the *group creation* command takes in average 91 seconds. This last one has the benefit of also being only used one by group.

In conclusion, some users can be thrown by the duration of by both the *push* and *group creation*, but by providing enough feedback in the app's interface it can be partially reduced.

### 3.2 Can it handle peak loads of numerous students trying to communicate with the solution? Is it costly for the faculty?

The last 2 questions (*Can it handle peak loads of numerous students trying to communicate with the solution? Is it costly for the faculty?*) can be summarized into one section as both are intrinsically aligned.

Because it is hosted in the cloud, the solution can be automatically scaled to respond to flexible loads to the servers. For that reason, the faculty pays only for the cloud resources needed. The auto scaling feature allows to scale resources intelligently to ensure better service availability, performance and reduce Cloud costs.

Unfortunately, the last two tests, *test3.sh* and *test4.sh* failed considerably. The prototype can't handle concurrent calls to the blockchain network because of the web server's implementation. This is occurring because the web server uses the minifabric binary to communicate with the blockchain network, and this binary can't be used by two or more processes at the same time. We can bypass this by using **Hyperledger Fabric Client SDK for Go**.