

Phase 9 - Testing and evaluation

Group 2:

Diogo Lança – fc53495

Meuri Canhanga – fc42559

Miguel Silva – fc58974

Pedro Santos – fc46417

Renato Ribeiro – fc57433

Operational support

To confirm that the microservices were acting in accordance with the system requirements defined, we use the pytest module to develop unit tests for each component of the "games", "users", "reviews" and "steam" microservices. We try to cover not only the positive cases, but also the cases in which invalid inputs were passed to the components. In total were defined 24 unit tests, and the result was:

```
neur@DESKTOP-R305TGJ:/mnt/c/CN22_Group2$ pytest --cov=. --cov-report xml:api/microservices/coverage.xml
===== test session starts =====
platform linux -- Python 3.8.10, pytest-7.1.2, pluggy-0.13.1
rootdir: /mnt/c/CN22_Group2
plugins: cov-3.0.0
collected 24 items

api/microservices/games/tests/test_gameMicroservice.py ..... [ 37%]
api/microservices/reviews/test_scripts/test_reviewsMicroservice.py ..... [ 75%]
api/microservices/steam/test_scripts/test_steamMicroservice.py .. [ 83%]
api/microservices/users/test_scripts/test_usersMicroservice.py .... [100%]

----- coverage: platform linux, python 3.8.10-final-0 -----
Coverage XML written to file api/microservices/coverage.xml

===== 24 passed in 15.51s =====
```

Based on test results, we note that the unit tests defined for the games Microservice aren't enough, since the code coverage obtained was less than 70% (which we considered an acceptable level).

We also created a GitHub action “Python application” described in the file `.github/workflows/python-app.yml` :

```
name: Python application

on:
  push:
    branches: [ devops ]
  pull_request:
    branches: [ main ]

permissions:
  contents: read

jobs:
  build:

    runs-on: ubuntu-latest

    steps:
      - uses: actions/checkout@v3
      - name: Set up Python 3.10
        uses: actions/setup-python@v3
        with:
          python-version: "3.10"
      - name: Install dependencies
        run: |
          python -m pip install --upgrade pip
          pip install flake8 pytest
          pip install -r api/microservices/games/requirements.txt
      - name: Start micorservices
        run: |
          python3 api/microservices/games/games.py &
          python3 api/microservices/users/users.py &
          python3 api/microservices/reviews/reviews.py &
          python3 api/microservices/steam/steam.py &
      - name: Test with pytest
        run: |
          pytest
```

This action creates a workflow for running a job that builds the application, starts the microservices and then runs the unit tests, which the purpose is to run that workflow when a pull request is created for the main branch.



Authentication

For the authentication on our API, we used the Auth0 external service. With Auth0 we configured an authentication API and created an App so that the integration with our Steam Reviews API was possible.

On this Auth0 authentication API we defined two different roles/scopes, the normal users, and the admins. With this we have three types of users that can use the Steam Reviews API, the non-authenticated users, the authenticated users with normal permissions, and the admins. The difference between normal users and admins is essentially that admins can add or remove games and accounts.

List of Permissions (Scopes)

These are all the permissions (scopes) that this API uses.

Permission	Description	
admin	Only for app Admins	
user	Only for authenticated users	

The flow used for this authentication App was the implicit flow.

So, we blocked some operations so that they could only be accessible by authenticated users, some for general users, some restricted to admins. This can be tested on swagger openAPI, for that, the user can click in “Authorize” and choose the permission he wants that session to have, either normal user or admin. After this he will be redirected to an auth0 login/register screen, that can be personalized to our liking. The user authenticate himself and then is redirected back to swagger, but the token of the authentication is stored by swagger and included on the request’s headers, and this is the way that when a request is made the token is verified and if it is valid the request goes through, if is not valid the unauthorized response is given.

Available authorizations

x

Scopes are used to grant an application different levels of access to data on behalf of the end user. Each API may declare one or more scopes.
API requires the following scopes. Select which ones you want to grant to Swagger UI.

oAuthCheck (OAuth2, implicit)

Authorization URL: <https://dev-p3dnwrxe.us.auth0.com/authorize?audience=https://cn22group2/>
Flow: implicit
client_id:

[Redacted client_id]

Scopes: [select all](#) [select none](#)

☐ admin
Only for app Admins

☐ user
Only for authenticated users

Authorize

Close

Reliability

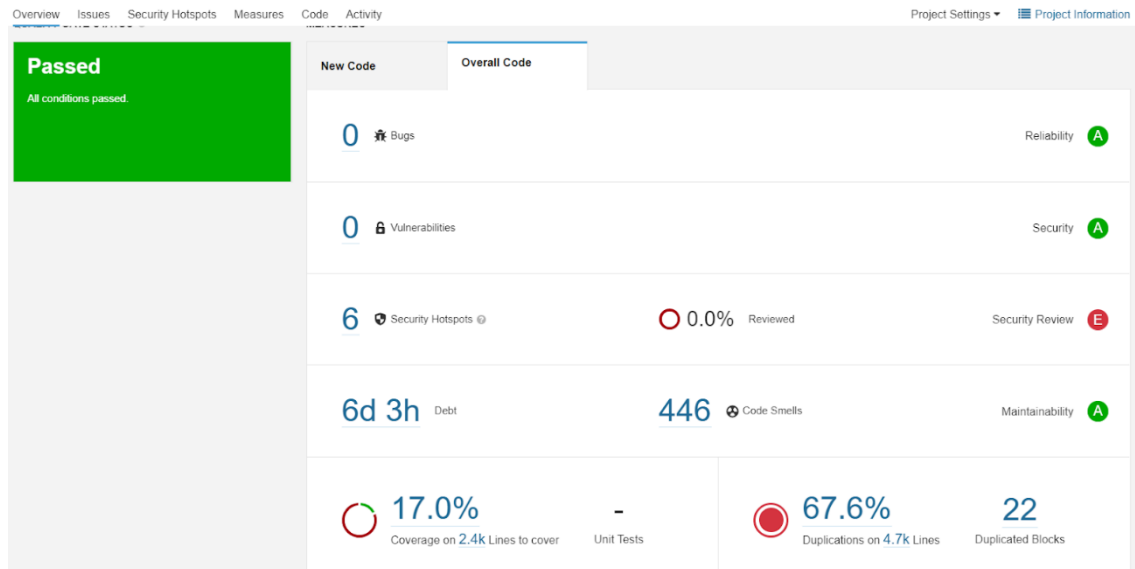
To measure the reliability of our implementation we used **SonarQube**. The scanning was locally, where we called the sonar-scanner to our project folder (cloned-repo where all pulled files go to from our repository) and the scanner checked all folders and files, measuring all the metrics specified within SonarQube.

On the SonarQube the primary indication of reliability is the number of bug issues. The difficulty of individual issues, their number, statuses, types, and severities are used to determine reliability rating and reliability remediation effort.

The **reliability remediation effort** is the effort to fix all bug issues. The measure is stored in minutes in the DB. An 8-hour day is assumed when values are shown in days.

A	0 Bugs
B	at least 1 Minor Bug
C	at least 1 Major Bug
D	at least 1 Critical Bug
E	at least 1 Blocker Bug

SonarQube score rating



Project final score

Cost Evaluation

Services	Cost
GKE Standard Node Pool (Kubernetes Engine)	25.30€
GKE Cluster Management Fee	0.00€
Anthos	15.16€
Cloud Build	0.00€
Total	40.47€

Table 1: Expected costs after 1 month

These are the expected costs after one month of running our application in the cloud. To obtain the cost of each service, we used Google Cloud Pricing Calculator, obtaining a total value of 25.30€, which is possible to cover in its entirety by the coupons given by the professor.

The Kubernetes engine turned out to be the most expensive where it was defined that our node was running 24 hours a day having a total of 730 hours per month.

GKE Cluster is a zonal cluster that has a total of 730 hours per month where one zonal cluster is free per billing account.

We used Anthos for Prometheus and Grafana, where we need 2 vCPUs running 24 hours a day.

Although Prometheus and Grafana are not yet implemented for this phase, it is something we intend to have implemented in the final delivery and so we decided to include its value in the costs.


Finally, the Cloud Build has a cost of 0.00€, due to the fact that the first 120 minutes of building per day are free.

Observability

In order to be able to observe and evaluate the performance of our cluster, we had to implement a Prometheus module that is supposed to evaluate and send data to a Grafana interface which would in turn allow us to see the data represented with more accuracy. We found several difficulties doing this, the first one being that the initial cluster we implemented didn't support the use of Prometheus. We had to roll back and reimplement our project in a cluster running a previous version of the GKE. The GCP already deploys a metrics server by default, so we didn't need to deploy our own. We used the GCP Marketplace to deploy a Prometheus and Grafana framework, and used a bash script to get its pods alongside our own as we can see here:

NAME	READY	STATUS	RESTARTS	AGE
api-gateway-7fccdf984b-d1l6m	1/1	Running	0	3h23m
api-gateway-7fccdf984b-g491f	1/1	Running	0	3h23m
games-78fbb84864-kv8ts	1/1	Running	0	3h23m
games-78fbb84864-w9vqx	1/1	Running	0	3h23m
prometheus-1-alertmanager-0	1/1	Running	0	4h12m
prometheus-1-alertmanager-1	1/1	Running	0	4h10m
prometheus-1-deployer-vdv9d	0/1	Completed	0	4h13m
prometheus-1-grafana-0	1/1	Running	0	4h12m
prometheus-1-kube-state-metrics-5fc94cff66-lz2xw	2/2	Running	0	4h12m
prometheus-1-node-exporter-c8pw4	1/1	Running	0	4h12m
prometheus-1-node-exporter-k826g	1/1	Running	0	4h12m
prometheus-1-node-exporter-rx9vr	1/1	Running	0	4h12m
prometheus-1-prometheus-0	1/1	Running	0	4h12m
prometheus-1-prometheus-1	1/1	Running	0	4h12m
reviews-75965d867b-bt7qf	1/1	Running	0	3h23m
reviews-75965d867b-fms2d	1/1	Running	0	3h23m
steam-5998896c97-4ggd2	1/1	Running	0	3h23m
steam-5998896c97-cg27t	1/1	Running	0	3h23m
users-546b77658-25z4l	1/1	Running	0	3h23m
users-546b77658-bxkwn	1/1	Running	0	3h23m

After we got Prometheus up and running as well as sending data to Grafana, we were able to access the Grafana UI by connecting to the IP specified in this service, using the specified port and logging in with the credentials we generated for Grafana:

<input type="checkbox"/>	prometheus-1-grafana	 OK	External load balancer	34.71.20.81:80	1/1	default	cluster-2
--------------------------	----------------------	--	------------------------	----------------	-----	---------	-----------

In the end we were able to see data about our clusters in the Grafana interface:

