Final Test Report

1. Introduction

1.1. Objective

This report summarizes the test execution results for the CoinGecko API testing. It covers functionality, performance, error handling, and security aspects.

1.2. Scope

The test was conducted on the following API endpoints:

- Simple Price: `GET /simple/price`
- Coin Markets: `GET /coins/markets`
- Coin List: `GET /coins/list`

2. Test Plan and Strategy

2.1. Testing approach

This plan follows a structured approach that includes:

1. **Manual Testing**: Using Postman to verify API functionality before automation.
2. **Automated Testing**: Implementing API test scripts in Postman and running them via Newman. After that this will be integrated in a CI/CD pipeline.
3. **Performance Testing**: Simulating multiple users to assess API response times and stability. Non-functional tests to to run Load testing and Stress testing.
4. **Security Testing**: Running basic security tests to identify vulnerabilities, like SQL injection, access control and data sensitivity protection.

2.2. Tools and technologies

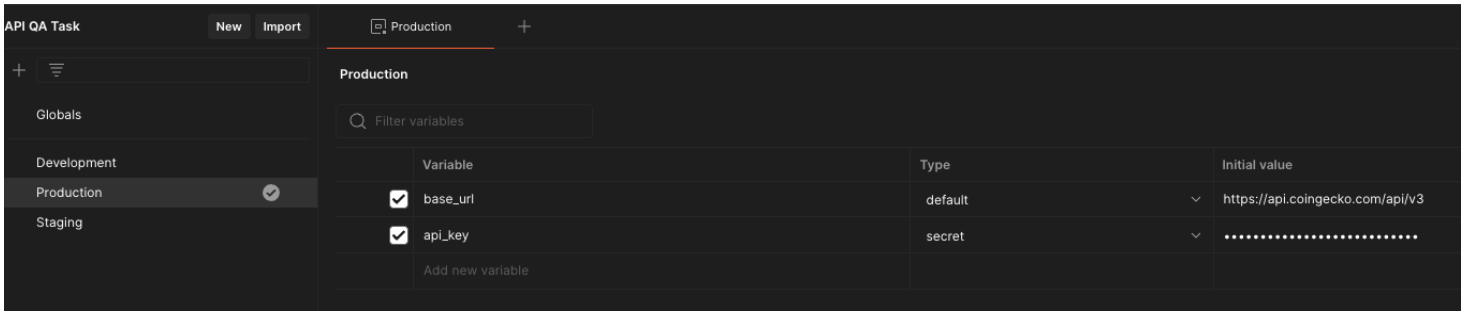
Type	Tool used
Manual testing	Postman
API automation	Postman / Newman
Performance testing	K6
Security testing	OWASP ZAP
CI/CD pipeline integration	GitHub Actions

2.3. Test environments

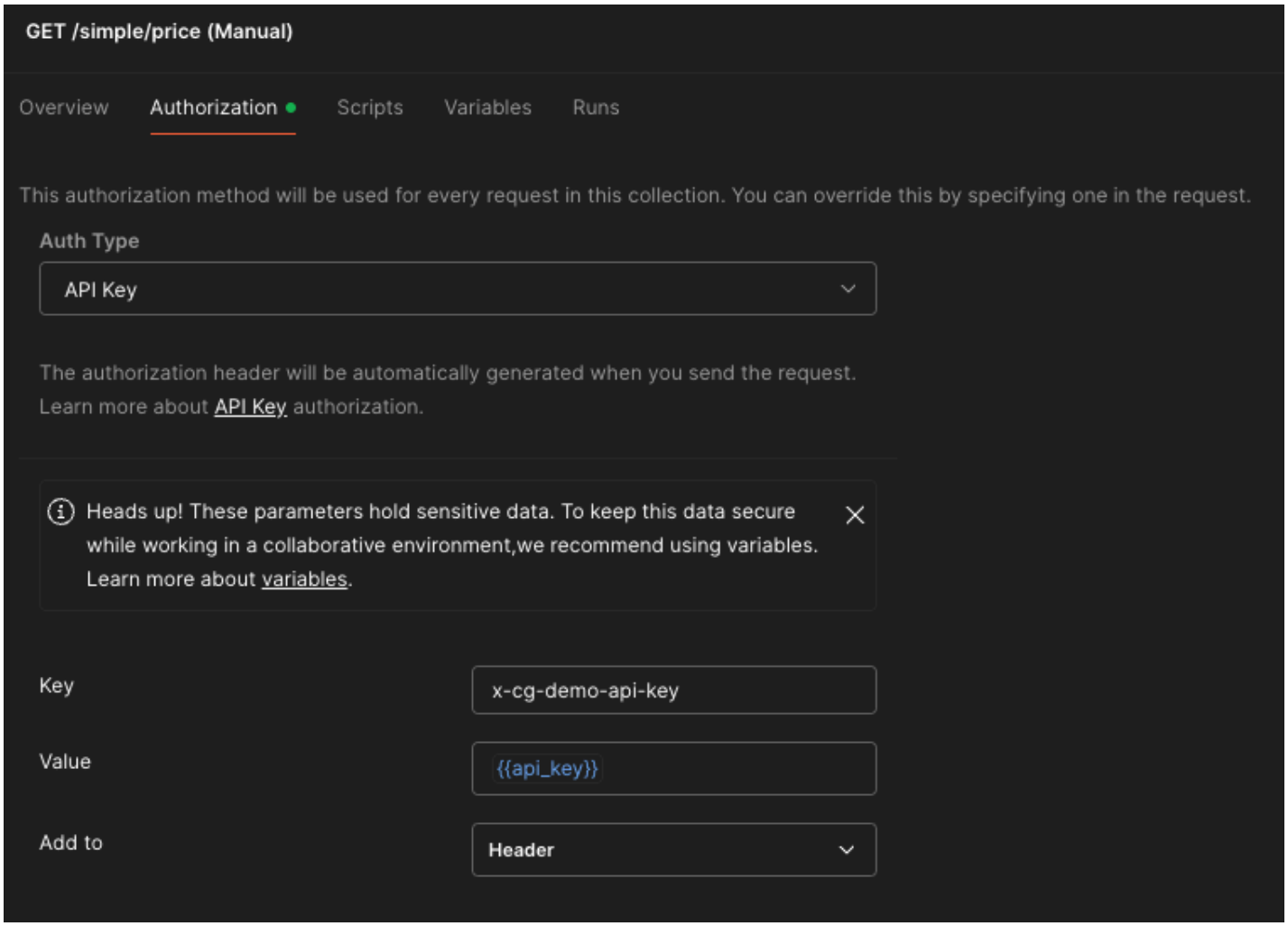
Since the CoinGecko public API (the demo API for free users) does not offer different URLs for different environments, we'll use the same here, but Postman was configured to have 3 environments with their own variables values (`base_url` and `api_key`) so the tests can reusable and configurable for all environments.

Environment	URL
Development	https://api.coingecko.com/api/v3
Staging	https://api.coingecko.com/api/v3
Production	https://api.coingecko.com/api/v3

The default environment is **Production**.



The `api_key` is used in the Authorization tab in each Collection in Postman, as recommended here in the CoinGecko API (https://docs.coingecko.com/v3.0.1/reference/authentication) to be used in the header.



3. Test execution plan

3.1. Manual tests

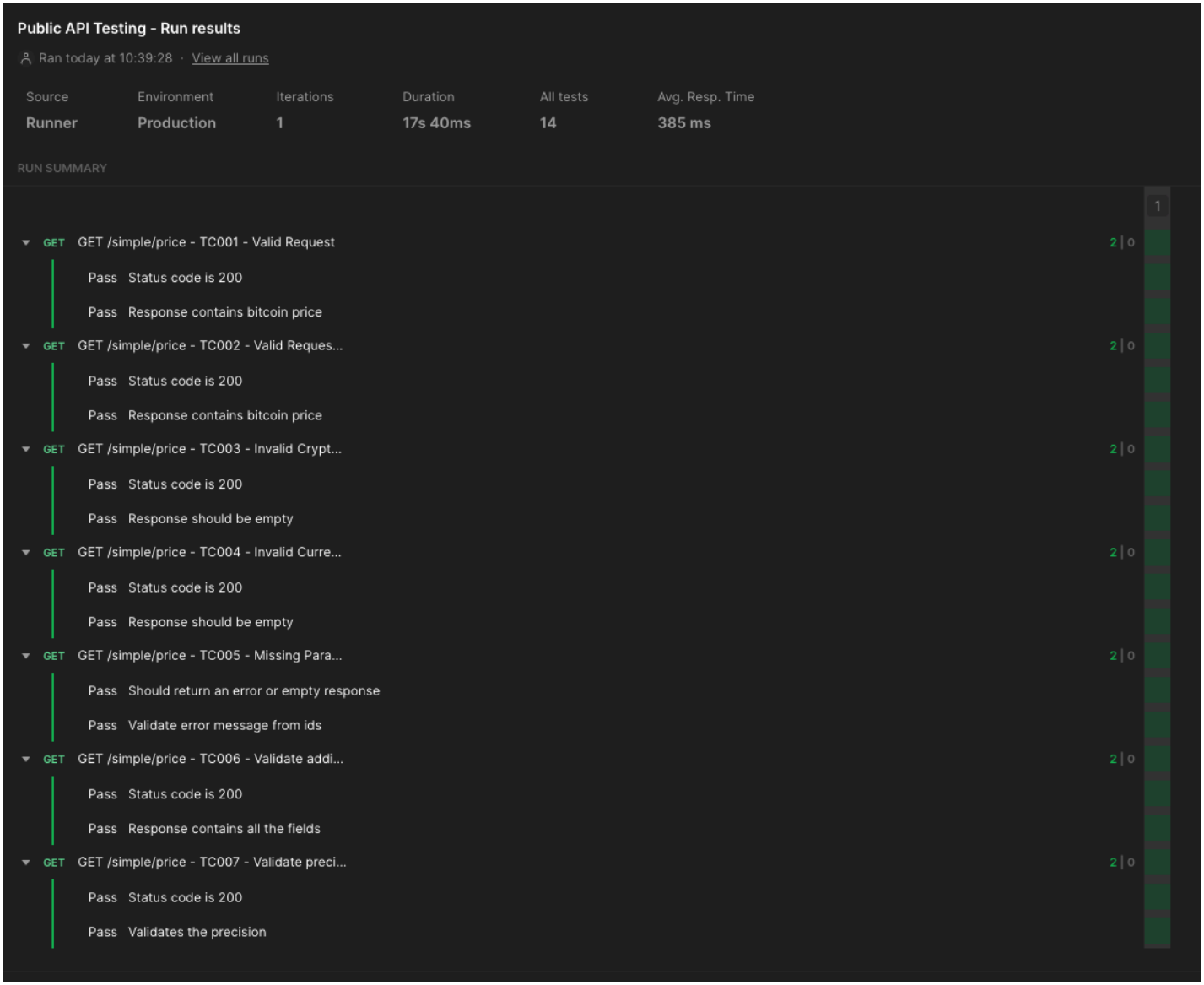
For the manual tests the chosen endpoint was the Simple Price: `GET /simple/price`
This Postman collection is available in the following path:
`scripts/Postman/GET-simple-price(Manual).postman_collection.json`

Test cases scenarios, steps and expected results:

Test Case ID	Test Scenario	Steps	Expected Result	Status (Pass/Fail)
TC001	Verify successful response with valid parameters	1. Send a GET request to `/simple/price` with `ids=bitcoin&vs_currencies=usd`. 2. Check the response status code. 3. Validate the response contains correct price data.	Status code = **200 OK** Response contains `{ "bitcoin": { "usd": 50000 } }` (example).	Passed
TC002	Verify response with multiple cryptocurrencies and currencies	1. Send a GET request with `ids=bitcoin,ethereum&vs_currencies=usd,eur`. 2. Validate response contains data for both cryptocurrencies in both currencies.	Status code = **200 OK** Response includes `{ "bitcoin": { "usd": 50000, "eur": 46000 }, "ethereum": { "usd": 3500, "eur": 3200 } }` (example).	Passed
TC003	Verify response when requesting an invalid cryptocurrency	1. Send a GET request with `ids=invalidcoin&vs_currencies=usd`. 2. Observe the API response.	Status code = **200 OK** API returns `{}` (empty json object).	Passed
TC004	Verify response when requesting an invalid currency	1. Send a GET request with `ids=bitcoin&vs_currencies=invalidcurrency`. 2. Check the response format. 3. (variations with invalid ids could be done)	Status code = **200 OK** API returns `{}` <u>if the id is unsupported</u>. If supported, the result is `{"bitcoin": {}}`, for example.	Passed
TC005	Verify response when missing required parameters	1. Send a GET request without `ids` and `vs_currencies`. 2. Observe the response.	Status code = **422 Unprocessable Entity** API returns an error message	Passed
TC006	Verify additional boolean parameters	1. Send a GET request with `include_market_cap=true&incl	Status code = **200 OK**	Passed

	(`include_market_cap`, `include_24hr_vol`, `include_24hr_change`, `include_last_updated_at`)	ude_24hr_vol=true&include_24hr_change=true&include_last_updated_at=true`. 2. Validate the response includes additional fields.	Response includes `market_cap`, `24h_vol`, `24h_change`, and `last_updated_at`.	
TC007	Verify response precision parameter	1. Send a GET request with `precision=2`. 2. Validate the response precision points.	Status code = **200 OK** Response numbers must have precision point '2'.	**Passed**

Summary of the results executed on Postman (delay between each request call was 3500 ms):



Besides this summary I also have the details for each Test case showing the parameters used for the calls, the test that were written for them and the results, like this example here:

TC007

Parameters:

GET {{base_url}} /simple/price?ids=bitcoin&vs_currencies=usd&precision=2

Params • Authorization Headers (8) Body Scripts • Settings

Query Params

Key	Value
ids	bitcoin
vs_currencies	usd
precision	2
Key	Value

Script Tests:

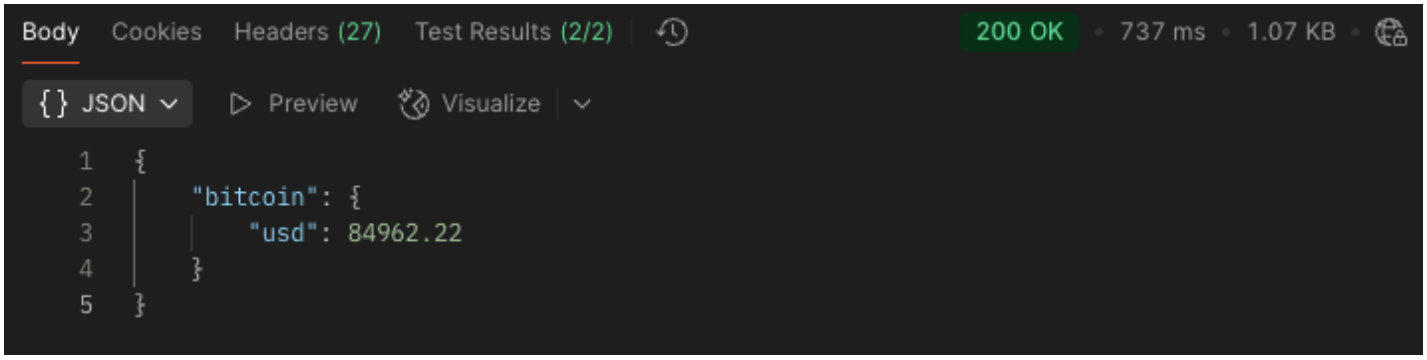
Params • Authorization Headers (8) Body Scripts • Settings

Pre-request

Post-response •

```
pm.test("Status code is 200", function () {
    pm.response.to.have.status(200);
});

pm.test("Response contains bitcoin price", function () {
    var jsonData = pm.response.json();
    pm.expect(jsonData).to.have.property('bitcoin');
    // usd
    pm.expect(jsonData.bitcoin).to.have.property('usd');
    pm.expect(jsonData.bitcoin.usd).to.be.a('number');
    //validate precision point
    pm.expect(String(jsonData.bitcoin.usd).split(".")[1]?.length == 2).to.be.true;
});
```

Results:



All the Test cases details are available in the the following path: reports/2-test_cases-simple_price.pdf

3.2. Automated tests

3.3. Test cases, Postman and Newman

For the manual tests the chosen endpoint was the Simple Price: `GET /coins/markets`
This Postman collection is available in the following path: `scripts/Postman/GET-coins-markets(Automation).postman_collection.json`

Test cases scenarios, steps and expected results:

Test Case ID	Test Scenario	Steps	Expected Result	Status (Pass/Fail)
TC101	Verify successful response with valid parameters	1. Send a GET request to `/coins/markets` with `vs_currency=usd`. 2. Check the response status code. 3. Validate that the response contains cryptocurrency market data.	Status code = **200 OK** Response contains a list of market data, including `id`, `symbol`, `name`, `current_price`, `market_cap`, etc.	Passed
TC102	Verify response with invalid currency	1. Send a GET request with `vs_currency=invalid_currency`. 2. Observe the response.	Status code = **400 Bad Request** API should return an **error message**	Passed
TC103	Verify response with specific coins	1. Send a GET request with `ids=bitcoin,ethereum`.	Status code = **200 OK**	Passed

		2. Validate that only Bitcoin and Ethereum market data are returned.	Response only contains objects for `bitcoin` and `ethereum`.	
TC104	Verify response when requesting an invalid coin	1. Send a GET request with `ids=invalidcoin`. 2. Observe the response.	Status code = **200 OK** API should return an **empty array `[ ]`**.	**Passed**
TC105	Verify response when omitting required parameters	1. Send a GET request **without `vs_currency`**. 2. Observe the response.	Status code = **422 Unprocessable Entity** API returns an error message indicating the missing parameter.	**Passed**
TC106	Verify optional parameters (`order`, `per_page`, `page`, `sparkline`)	1. Send a GET request with various optional parameters such as `order=market_cap_desc`, `per_page=10`, `page=2`, and `sparkline=true`. 2. Validate the response format and pagination.	Status code = **200 OK** Response should be **sorted** by market cap, limited to 10 results, on page 2, with sparkline data included.	**Passed**
TC107	Verify response pagination	1. Send a GET request with `per_page=50` and `page=2`. 2. Validate that page 2 results are different from page 1.	Status code = **200 OK** API returns a **different set of coins** for page 2.	**Passed**
TC108	Verify response when requesting an excessively high page number	1. Send a GET request with `per_page=50` and `page=99999`. 2. Observe the response.	Status code = **200 OK** API should return an **empty array `[ ]`** if no results exist for that page.	**Passed**

All the Test Cases have 2 tests: 1 validating the status code and the other the expected result.

Since the idea here is to automate the test runs, first I installed `newman` in my local environment via `npm` and ran this collection choosing the Production environment:

```
newman run GET-coins-markets\(Automation\).postman_collection.json -e
Production.postman_environment.json --reporters cli,junit
```

and the results were:

```
→ GET /coins/markets - TC101 - Verify successful response with valid parameters
  GET https://api.coingecko.com/api/v3/coins/markets?vs_currency=usd [200 OK, 81.06kB, 340ms]
  ✓  Status code is 200
  ✓  Response contains valid market data

→ GET /coins/markets - TC102 - Verify response with invalid currency
  GET https://api.coingecko.com/api/v3/coins/markets?vs_currency=invalid_currency [400 Bad Request, 996B, 203ms]
  ✓  Status code is 200
  ✓  Response should be empty or return an error

→ GET /coins/markets - TC103 - Verify response with specific coins
  GET https://api.coingecko.com/api/v3/coins/markets?ids=bitcoin,ethereum&vs_currency=usd [200 OK, 2.72kB, 186ms]
  ✓  Status code is 200
  ✓  Response contains valid market data

→ GET /coins/markets - TC104 - Verify response when requesting an invalid coin
  GET https://api.coingecko.com/api/v3/coins/markets?ids=invalidcoin&vs_currency=usd [200 OK, 1.08kB, 221ms]
  ✓  Status code is 200
  ✓  Response should be empty

→ GET /coins/markets - TC105 - Verify response when omitting required parameters
  GET https://api.coingecko.com/api/v3/coins/markets [422 Unprocessable Entity, 739B, 206ms]
  ✓  Should return an error or empty response
  ✓  Validate error message from ids

→ GET /coins/markets - TC106 - Verify optional parameters
  GET https://api.coingecko.com/api/v3/coins/markets?vs_currency=usd&order=market_cap_desc&per_page=10&page=2&sparkline=true [200 OK, 41.53kB, 265ms]
  ✓  Status code is 200
  ✓  Response contains valid market data

→ GET /coins/markets - TC107 - Verify response pagination
  GET https://api.coingecko.com/api/v3/coins/markets?vs_currency=usd&per_page=50&page=2 [200 OK, 41.42kB, 263ms]
  ✓  Status code is 200
  ✓  Response contains valid market data

→ GET /coins/markets - TC108 - Verify response when requesting an excessively high page number
  GET https://api.coingecko.com/api/v3/coins/markets?vs_currency=usd&per_page=50&page=99999 [200 OK, 1.08kB, 241ms]
  ✓  Status code is 200
  ✓  Response should be empty
```

	executed	failed
iterations	1	0
requests	8	0

erge branch 'main' of github.com:felipejgribeiro/onchain-qa-api-test

test-scripts	16	0
prerequest-scripts	8	0
assertions	16	0

total run duration: 2.1s

total data received: 162.51kB (approx)

average response time: 240ms [min: 186ms, max: 340ms, s.d.: 46ms]

I also generated a HTML report that can be found at reports/3-newman-run-report-example.html

3.4. CI/CD Pipeline

To integrate all of this in a CI/CD pipeline tool I chose the Github actions by editing the default yml file github provide by adding the items we need:

- Create a test job that runs latest ubuntu image
- Create the following steps:
  - Check out the repository
  - Install Node.js 22 (same one I'm using in my local environment)
  - Install newman
  - Change dir to where the postman collections are and run the tests with newman

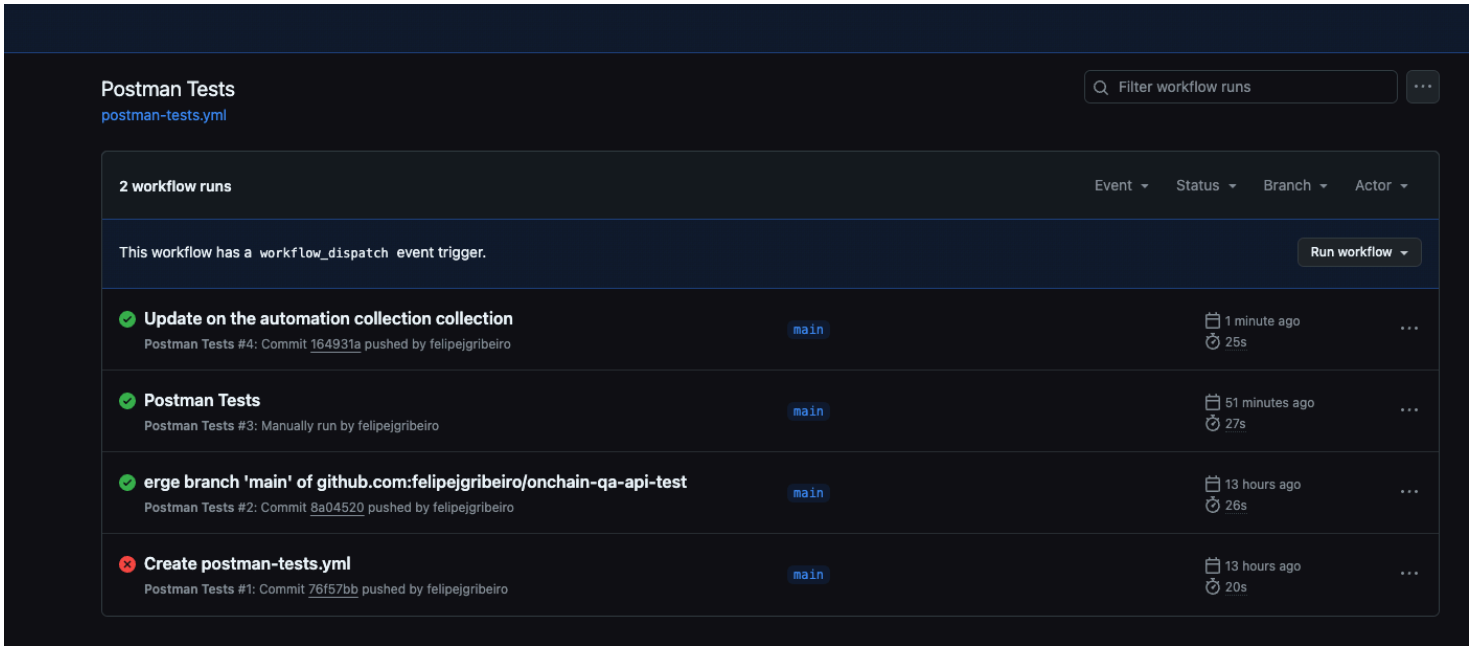
The triggers will be done on `pushes` or `pull_requests` on the main branch.

The script (also provided on .github/workflows/postman-tests.yml):

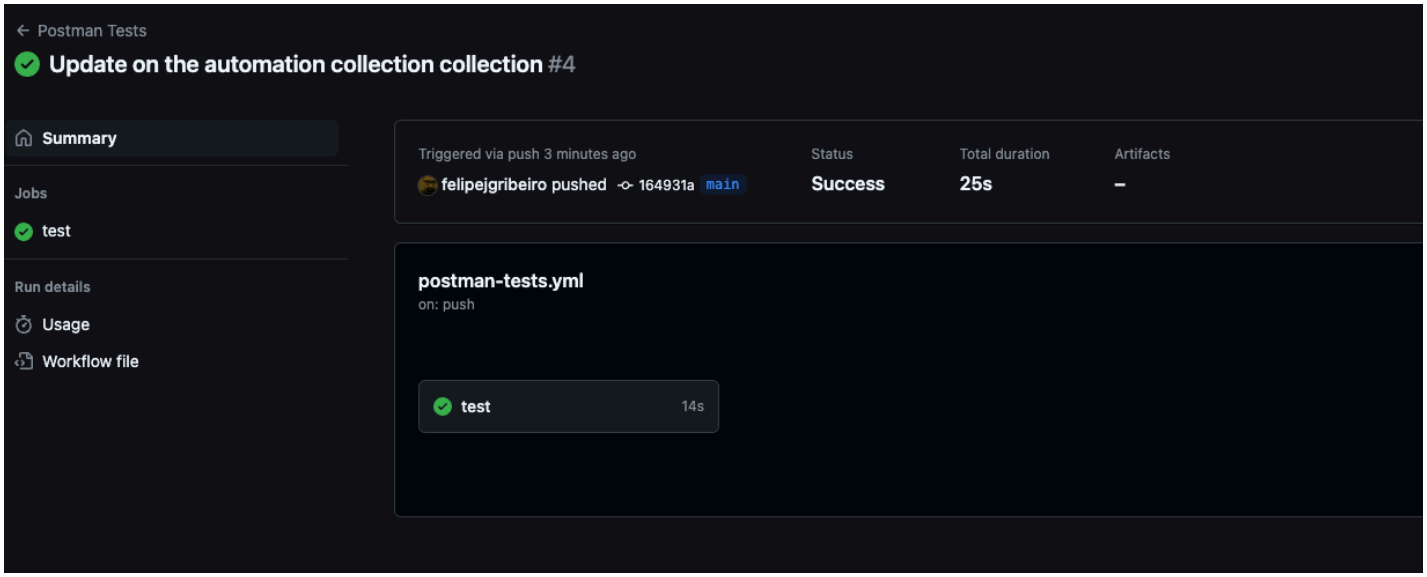
```yaml
name: Postman Tests

# Controls when the workflow will run
on:
  # Triggers the workflow on push or pull request events but only for the "main" branch
  push:
    branches: [ "main" ]
  pull_request:
    branches: [ "main" ]

  # Allows you to run this workflow manually from the Actions tab
  workflow_dispatch:

jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout Repository
        uses: actions/checkout@v2

      - name: Install Node.js
        uses: actions/setup-node@v2
        with:
          node-version: '22'

      - name: Install Newman
        run: npm install -g newman

      - name: Run Postman Collection
        run: cd scripts/Postman && newman run GET-coins-markets\(Automation\).postman_collection.json -e Production.postman_environment.json --reporters cli,junit
```

The URL: https://github.com/felipejgribeiro/onchain-qa-api-test/actions/workflows/postman-tests.yml

The runs on the workflow panel:



The latest run:



```
→ GET /coins/markets - TC107 - Verify response pagination
  GET https://api.coingecko.com/api/v3/coins/markets?vs_currency=usd&per_page=50&page=2 [200 OK, 41.2kB, 148ms]
  ✓  Status code is 200
  ✓  Response contains valid market data

→ GET /coins/markets - TC108 - Verify response when requesting an excessively high page number
  GET https://api.coingecko.com/api/v3/coins/markets?vs_currency=usd&per_page=50&page=99999 [200 OK, 1.08kB, 139ms]
  ✓  Status code is 200
  ✓  Response should be empty

┌─────────────────────────┬────────────────────┬───────────────────┐
│                         │           executed │            failed │
├─────────────────────────┼────────────────────┼───────────────────┤
│              iterations │                  1 │                 0 │
├─────────────────────────┼────────────────────┼───────────────────┤
│                requests │                  8 │                 0 │
├─────────────────────────┼────────────────────┼───────────────────┤
│            test-scripts │                 16 │                 0 │
├─────────────────────────┼────────────────────┼───────────────────┤
│      prerequest-scripts │                  8 │                 0 │
├─────────────────────────┼────────────────────┼───────────────────┤
│              assertions │                 16 │                 0 │
├─────────────────────────┴────────────────────┴───────────────────┤
│ total run duration: 1297ms                                        │
├───────────────────────────────────────────────────────────────────┤
│ total data received: 161.59kB (approx)                            │
├───────────────────────────────────────────────────────────────────┤
│ average response time: 122ms [min: 86ms, max: 176ms, s.d.: 31ms] │
└───────────────────────────────────────────────────────────────────┘
```

4. Performance analysis

All the K6 scripts are in the scripts/K6 folder and there is no need to paste their entire code here.

For the performance tests the chosen endpoint was the Coins List: `GET /coins/list`

4.1. Load tests

For these 2 executions I used the same data as input:

- 50 Users (Virtual Users os VUs).
- 1 minute of test running.
- All the response status code should be 200.
- All the response time should be < 500 ms.

Since this is a CoinGecko demo's environment that is not robust and has rate limits for free users, these parameters were very OK for the tests to run and I was also able to get insights for improvements like a real environment analysis.

4.1.1. First execution - Native report

```
     scenarios: (100.00%) 1 scenario, 50 max VUs, 1m30s max duration (incl. graceful stop):
              * default: 50 looping VUs for 1m0s (gracefulStop: 30s)


     ✓ Status code is 200
     ✗ Response time is < 500ms
      ↳  84% — ✓ 1901 / ✗ 348

     checks.........................: 92.26% 4150 out of 4498
     data_received..................: 2.3 GB 37 MB/s
     data_sent......................: 6.6 MB 108 kB/s
     http_req_blocked...............: avg=6ms      min=0s      med=0s       max=299.59ms p(90)=1µs      p(95)=1µs
     http_req_connecting............: avg=566.8µs  min=0s      med=0s       max=49.25ms  p(90)=0s       p(95)=0s
     http_req_duration..............: avg=341.91ms min=62.17ms med=270.65ms max=3s       p(90)=638.2ms  p(95)=888.5ms
       { expected_response:true }...: avg=341.91ms min=62.17ms med=270.65ms max=3s       p(90)=638.2ms  p(95)=888.5ms
     http_req_failed................: 0.00%  0 out of 2249
     http_req_receiving.............: avg=255.74ms min=7.65ms  med=184.18ms max=2.95s    p(90)=551.36ms p(95)=759.19ms
     http_req_sending...............: avg=22.53µs  min=8µs     med=13µs     max=11.23ms  p(90)=23µs     p(95)=30µs
     http_req_tls_handshaking.......: avg=1.23ms   min=0s      med=0s       max=78.64ms  p(90)=0s       p(95)=0s
     http_req_waiting...............: avg=86.14ms  min=27.66ms med=80.54ms  max=375.63ms p(90)=132.03ms p(95)=154.48ms
     http_reqs......................: 2249   36.694638/s
     iteration_duration.............: avg=1.34s    min=1.06s   med=1.27s    max=4s       p(90)=1.64s    p(95)=1.89s
     iterations.....................: 2249   36.694638/s
     vus............................: 10     min=10           max=50
     vus_max........................: 50     min=50           max=50


running (1m01.3s), 00/50 VUs, 2249 complete and 0 interrupted iterations
default ✓ [======================================] 50 VUs  1m0s
```

4.1.2. Second execution - InfluxDB + Grafana report

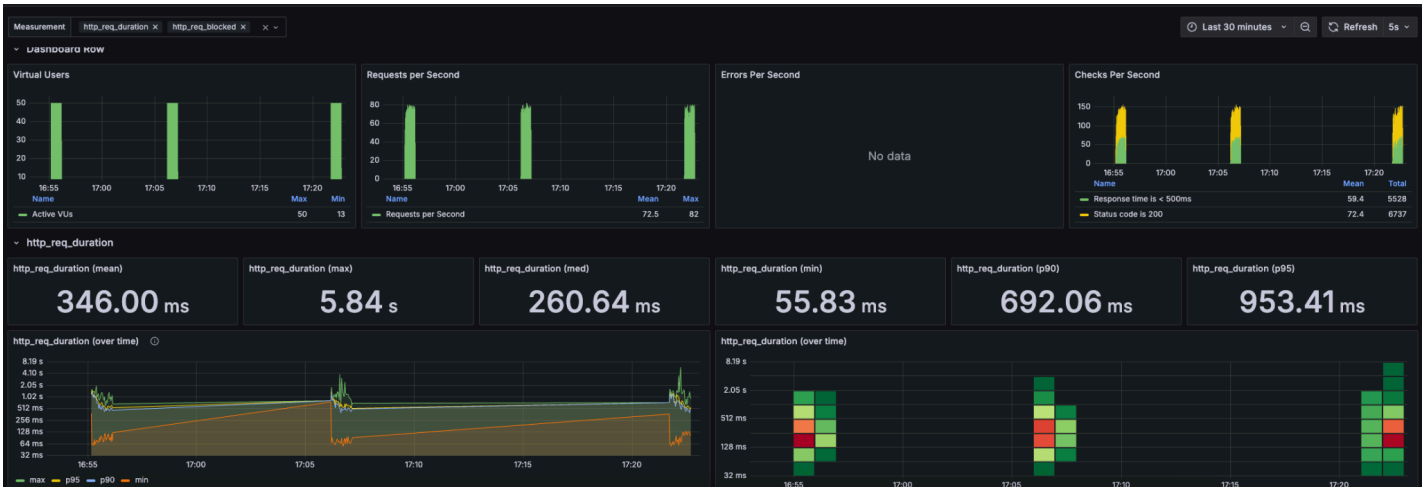
Having a better way to see the data is always good to improve productivity. So as a taste of what we can do with the data I created a docker compose file the install InfluxDB and Grafana so we can visualize the data better and we can have many other benefits like:

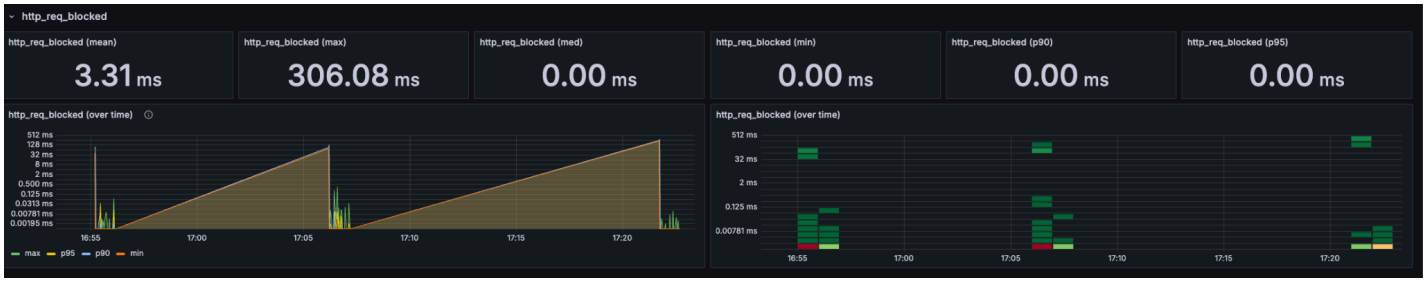
- Historical data
- Real time data
- Personalized dashboards
- Graphical data visualization
- Configure alerts

The docker-compose file is located at `scripts/k6/influxdb_grafana/docker-compose.yml` and to start this stack just go to this folder and run `docker-compose up -d` or `docker-compose down` to stop and remove the containers.

After having the containers running, just run the k6 script as: `k6 run load_testing.js --out influxdb=http://admin:admin123@localhost:8086/k6`

The results with this approach were the screenshots below. Note I ran 3 times with the same parameters just for the sake of showing the historical data, real time and all the benefits.

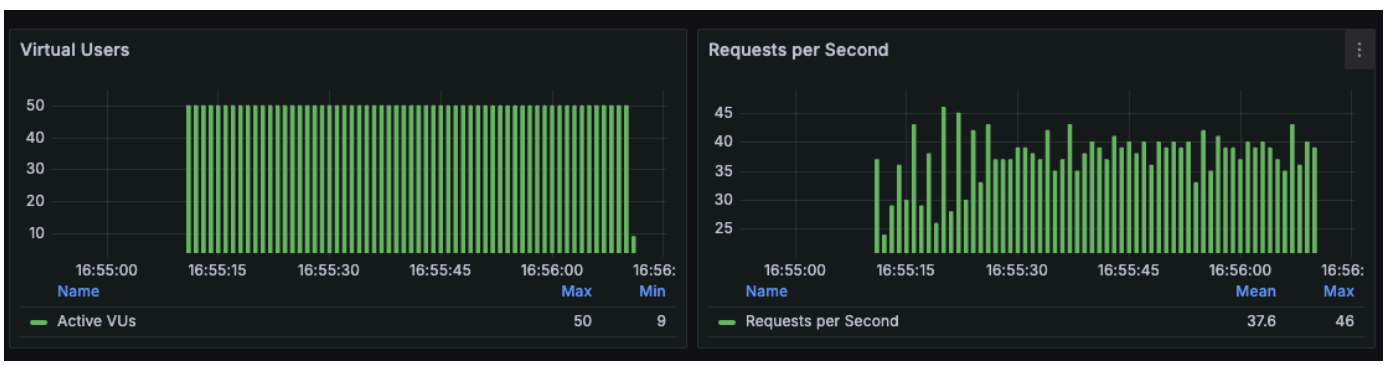




4.1.3. Results and analysis

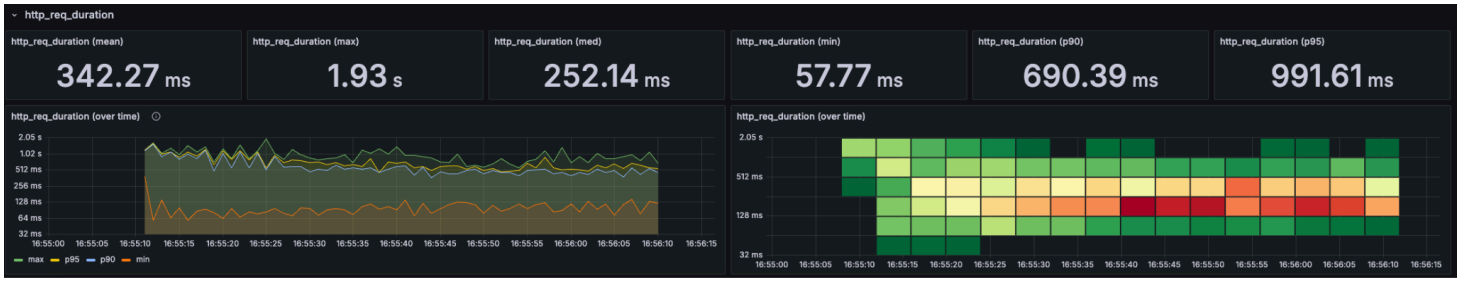
<u>Positive outcomes:</u>

1. **`http_req_failed: 0.00%`**
2. The mean for all runs was around 2200 requests. **All successful, no errors. HTTP 200 OK.**
3. High Throughput (Requests Per Second). Isolating just one execution of the tests we had 37.6 requests per second and a mx of 46 having a constant number of 50VUs:



<u>Warnings:</u>

1. High response time.
    a. Average Response Time (http_req_duration): 342.27ms
    b. 95th percentile (p(95)): 991.61ms
    c. Max Response Time: 1.93 seconds
    d. <u>Problem: 18% of requests exceeded 500ms</u> (the threshold configured in the script).



2. Spikes in connection times
    a. Max Connection Time: 40.96ms
    b. Some requests took up to 299.59ms to be blocked
    c. Resolution: for me this is just a matter of using this demo API that has rate limits and limitations like that.
3. Long waiting time
    a. Avg. Wait Time (http_req_waiting): 73.90ms
    b. P95 Waiting Time: 127.67ms
    c. <u>Max Wait Time (http_req_waiting): 430.92ms</u>
    d. <u>Resolution: Think about caching data, improving backend processing or database queries.</u>

4.2. Stress tests

The stress test will have this configuration to ramp up and ramp down the users:

```
export let options = {
  stages: [
    { duration: '30s', target: 25 },  // Start with 25 users
    { duration: '30s', target: 50 },  // Increase to 50 users
    { duration: '1m', target: 200 },  // Increase to 200 users
    { duration: '30s', target: 500 }, // Stress test at 500 users
    { duration: '30s', target: 0 },   // Ramp down to 0 users
  ],
};
```

Which means the test will last 3 minutes starting with 25 users, peak with 500 users and ramp down to 0 in the last 30 seconds.

4.2.1. First execution - Native report

```
     scenarios: (100.00%) 1 scenario, 500 max VUs, 3m30s max duration (incl. graceful stop):
              * default: Up to 500 looping VUs for 3m0s over 5 stages (gracefulRampDown: 30s, gracefulStop: 30s)

WARN[0150] Request Failed                                error="Get \"https://api.coingecko.com/api/v3/coins/list\": unexpected EOF"
WARN[0157] Request Failed                                error="Get \"https://api.coingecko.com/api/v3/coins/list\": read tcp 192.168.15.203:562
33->104.22.78.164:443: read: connection reset by peer"
WARN[0159] Request Failed                                error="Get \"https://api.coingecko.com/api/v3/coins/list\": http2: client conn could no
t be established"
WARN[0168] Request Failed                                error="Get \"https://api.coingecko.com/api/v3/coins/list\": read tcp 192.168.15.203:562
09->104.22.78.164:443: read: connection reset by peer"
WARN[0177] Request Failed                                error="request timeout"

     ✗ Status code is 200
      ↳  99% — ✓ 5831 / ✗ 5
     ✗ Response time is < 1000ms
      ↳  35% — ✓ 2062 / ✗ 3774

     checks.........................: 67.62% 7893 out of 11672
     data_received..................: 5.9 GB 30 MB/s
     data_sent......................: 15 MB  78 kB/s
     http_req_blocked...............: avg=100.66ms min=0s      med=1µs      max=17.48s  p(90)=2µs      p(95)=334.08ms
     http_req_connecting............: avg=42.68ms  min=0s      med=0s       max=6.21s   p(90)=0s       p(95)=159.7ms
     http_req_duration..............: avg=3.77s    min=0s      med=1.95s    max=1m0s    p(90)=8.95s    p(95)=13.29s
       { expected_response:true }...: avg=3.76s    min=52.34ms med=1.95s    max=54.92s  p(90)=8.94s    p(95)=13.29s
     http_req_failed................: 0.08%  5 out of 5836
     http_req_receiving.............: avg=3.58s    min=0s      med=1.73s    max=59.63s  p(90)=8.66s    p(95)=13.03s
     http_req_sending...............: avg=58.18µs  min=0s      med=29µs     max=30.45ms p(90)=103µs    p(95)=140µs
     http_req_tls_handshaking.......: avg=58.15ms  min=0s      med=0s       max=15.48s  p(90)=0s       p(95)=167.84ms
     http_req_waiting...............: avg=187.37ms min=0s      med=174.73ms max=3.5s    p(90)=366.32ms p(95)=496.67ms
     http_reqs......................: 5836   30.108822/s
     iteration_duration.............: avg=4.88s    min=1.05s   med=2.99s    max=1m1s    p(90)=10.18s   p(95)=14.39s
     iterations.....................: 5836   30.108822/s
     vus............................: 1      min=1            max=500
     vus_max........................: 500    min=500          max=500


running (3m13.8s), 000/500 VUs, 5836 complete and 1 interrupted iterations
default ✓ [======================================] 000/500 VUs  3m0s
```

4.2.2. Second execution - InfluxDB + Grafana report

4.2.3. Results and analysis

<u>Positive outcomes:</u>

1. High traffic handled by the API
    a. 5749 successful requests
    b. Data received: 5.8GB (~29MB/s)

2. Low failure rate
    a. 0.07% of failures (not HTTP 200 status)
    http_req_failed: 0.07% (4 out of 5749 requests failed). Despite 500 users making continuous requests, 99.93% of API requests succeeded.

Warnings:

1. High response times
    a. **Avg Response Time (`http_req_duration`): 3.85s**
    b. **P90 Response Time: 9.09s**
    c. **P95 Response Time: 14.08s**
    d. **Max Response Time: 1 min**
    e. Solution: Think about caching data, improving backend processing or database queries. Maybe Redis, Nginx cache, Cloudflare.

2. Connection Errors & API Instability
    a. **Errors observed:**
        i. `"connection reset by peer"`
        ii. `"unexpected EOF"`
        iii. `"request timeout"`

3. High waiting times
    a. Avg Wait Time (http_req_waiting): 190.04ms
    b. P90 Wait Time: 364.03ms
    c. P95 Wait Time: 507.22ms

Summary of the warnings:

Metric	Observed	Ideal target	Action Needed ?
API Success Rate	✅ 99.93% success (4 failures)	99%+	✅ API is stable
Requests per Second	✅ 30 req/sec	25+ req/sec	✅ Good throughput
Avg Response Time	❌ **3.77s**	<500ms	⚠️ Needs optimization
P90 Response Time	❌ **8.95s**	<1s	⚠️ High latency
P95 Response Time	❌ **13.29s**	<1.5s	❗ Urgent
Max Response Time	❌ **54.92s**	<2s	❗ Critical issue
Connection Errors	❌ **Multiple timeouts**	None	❗ Urgent

5. Security analysis

All tests will be done using OWASP ZAP, which is one of the best open-source tools for automated and manual API security testing. It is actively maintained by the OWASP community and is widely used by penetration testers, DevSecOps teams, and security engineers.
For the security tests the chosen endpoint was the Coins List: `GET /coins/list`

Those are all the calls made by the tool using the **Active Scan**:

Sent Messages | Filtered Messages

ID	Req. Timestamp	Resp. Timestamp	Method	URL	Code	Reason
20	2/27/25, 9:54:42 PM	2/27/25, 9:54:42 PM	GET	https://api.coingecko.com/api/v3/coins/1043411681070065783	404	Not Found
22	2/27/25, 9:54:42 PM	2/27/25, 9:54:42 PM	GET	https://api.coingecko.com/latest/meta-data/	421	Misdirected Request
23	2/27/25, 9:54:42 PM	2/27/25, 9:54:42 PM	GET	https://api.coingecko.com/latest/meta-data/	421	Misdirected Request
24	2/27/25, 9:54:42 PM	2/27/25, 9:54:42 PM	GET	https://api.coingecko.com/computeMetadata/v1/	421	Misdirected Request
25	2/27/25, 9:54:42 PM	2/27/25, 9:54:42 PM	GET	https://api.coingecko.com/computeMetadata/v1/	421	Misdirected Request
26	2/27/25, 9:54:42 PM	2/27/25, 9:54:42 PM	GET	https://api.coingecko.com/opc/v1/instance/	421	Misdirected Request
27	2/27/25, 9:54:42 PM	2/27/25, 9:54:42 PM	GET	https://api.coingecko.com/opc/v1/instance/	421	Misdirected Request
28	2/27/25, 9:54:42 PM	2/27/25, 9:54:42 PM	GET	https://api.coingecko.com/latest/meta-data/	421	Misdirected Request
29	2/27/25, 9:54:42 PM	2/27/25, 9:54:42 PM	GET	https://api.coingecko.com/latest/meta-data/	421	Misdirected Request
30	2/27/25, 9:54:42 PM	2/27/25, 9:54:42 PM	GET	https://api.coingecko.com/metadata/instance	421	Misdirected Request
31	2/27/25, 9:54:43 PM	2/27/25, 9:54:43 PM	GET	https://api.coingecko.com/api/v3/coins/list/	200	OK
32	2/27/25, 9:54:43 PM	2/27/25, 9:54:44 PM	GET	https://api.coingecko.com/api/v3/coins/actuator/health	404	Not Found
33	2/27/25, 9:54:44 PM	2/27/25, 9:54:44 PM	GET	https://api.coingecko.com/api/v3/coins/actuator/health	404	Not Found

And this is the report provided by those calls:

https://api.co..i/v3/coins/list Scan Progress

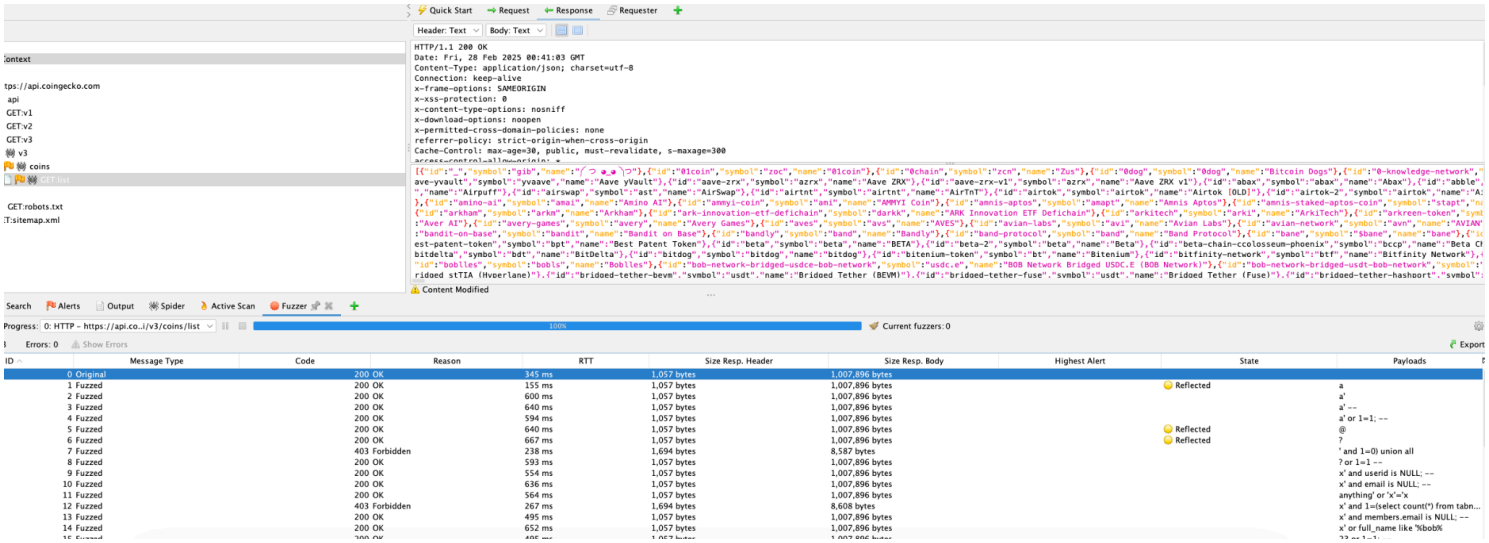
Progress | Response Chart

Host: https://api.coingecko.com

	Strength	Progress	Elapsed	Reqs	Alerts	Status
Analyser			00:00.360	1		
Plugin						
Remote File Inclusion	Medium		00:00.013	0	0	✔
External Redirect	Medium		00:00.007	0	0	✔
Server Side Include	Medium		00:00.011	0	0	✔
SQL Injection	Medium		00:00.004	0	0	✔
Server Side Code Injection	Medium		00:00.008	0	0	✔
Remote OS Command Injection	Medium		00:00.005	0	0	✔
XPath Injection	Medium		00:00.004	0	0	✔
XML External Entity Attack	Medium		00:00.008	0	0	✔
Cloud Metadata Potentially Exposed	Medium		00:00.372	9	0	✔
Server Side Template Injection	Medium		00:00.369	0	0	✔
Server Side Template Injection (Blind)	Medium		00:00.007	0	0	✔
Directory Browsing	Medium		00:00.844	1	0	✔
Buffer Overflow	Medium		00:00.004	0	0	✔
Format String Error	Medium		00:00.004	0	0	✔
CRLF Injection	Medium		00:00.005	0	0	✔
Parameter Tampering	Medium		00:00.003	0	0	✔
Spring Actuator Information Leak	Medium		00:00.528	2	0	✔
XSLT Injection	Medium		00:00.525	0	0	✔
Script Active Scan Rules	Medium		00:00.006	0	0	⊘
SOAP Action Spoofing	Medium		00:00.002	0	0	✔
SOAP XML Injection	Medium		00:00.007	0	0	✔
Totals			00:02.221	13	0	

The **Script Active Scan Rules** marked as a problem in the status in fact it was skipped because the result was: no scripts enabled.

Now, below are all the calls and results made by the tool using the **Fuzzer Scan**. The Fuzzer first makes a call with an original payload and then creates a lot of different payloads to simulate **SQL Injection, XSS injection, Header without authentication**, etc.



A .csv file is available in the reports folder with all of the payloads here. The tool also generates a HTML report, also available in the report folder (`2025-02-27-ZAP-Report-.html`). This is one of the summaries:

Summaries

Alert counts by risk and confidence

This table shows the number of alerts for each level of risk and confidence included in the report.

(The percentages in brackets represent the count as a percentage of the total number of alerts included in the report, rounded to one decimal place.)

		Confidence				
		User Confirmed	High	Medium	Low	Total
Risk	High	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)
	Medium	0 (0.0%)	0 (0.0%)	1 (33.3%)	0 (0.0%)	1 (33.3%)
	Low	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)
	Informational	0 (0.0%)	0 (0.0%)	1 (33.3%)	1 (33.3%)	2 (66.7%)
	Total	0 (0.0%)	0 (0.0%)	2 (66.7%)	1 (33.3%)	3 (100%)

The most severe problem found was a Cross-Domain Misconfiguration:

Alerts

Risk=Medium, Confidence=Medium (1)

https://api.coingecko.com (1)

Cross-Domain Misconfiguration (1)

▶ GET https://api.coingecko.com/api/v3/coins/list

The request, response, evidences and possibles solutions are also described there in the report:

Request

▼ Request line and header section (253 bytes)

```
GET https://api.coingecko.com/api/v3/coins/list
HTTP/1.1
host: api.coingecko.com
user-agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/131.0.0.0 Safari/537.36
pragma: no-cache
cache-control: no-cache
```

▼ Request body (0 bytes)

Response

▶ Status line and header section (1057 bytes)

▶ Response body (1007896 bytes)

Evidence

`access-control-allow-origin: *`

Solution

Ensure that sensitive data is not available in an unauthenticated manner (using IP address white-listing, for instance).

Configure the "Access-Control-Allow-Origin" HTTP header to a more restrictive set of domains, or remove all CORS headers entirely, to allow the web browser to enforce the Same Origin Policy (SOP) in a more restrictive manner.