Declan Dornstauder
11284744 / DVU072
CMPT 436
Project

Exploring Concurrency in Elixir: Analyzing the Interaction Between Phoenix REST APIs and Concurrent Client Requests

I. Introduction

In the ever-evolving domain of web technologies, the significance of leveraging concurrent processing to achieve optimal performance cannot be overstated. The emergence of Elixir, a robust functional programming language, and Phoenix, a high-performance web framework, presents an opportunity to explore the interplay between client-initiated concurrent requests and server-side handling within a RESTful architecture.

The primary objective of this project is to conduct a high-level analysis of concurrency from both the client and server perspectives. Specifically, the analysis will focus on setting up a Phoenix REST API and implementing an Elixir client script to generate multiple simultaneous requests to this API. Through this setup, the project aims to scrutinize and evaluate how the API manages and responds to concurrent requests initiated by the client.

II. Setup

To explore the objective of this project, the following technologies are used:

1. Phoenix and PostgreSQL to implement the REST API and database.
2. Elixir to implement the client script.
3. Prometheus to monitor and visualize performance metrics.
4. Docker for containerization

1. REST API and database

Both a Phoenix REST API and PostgreSQL database are deployed as containers within a Docker cluster. The Postgres database is connected to the API, allowing client requests to modify its data via the following endpoint:

localhost:5000/api/items

The API is initialized using a command *(appendix A)* flagged to generate a Phoenix project without the additional functionalities required in the development of a Phoenix web

application. In the case of a REST API, particularly in the context of this analysis, it is important to ensure the API's foundation is as lightweight as possible.

To ensure connectivity to the API in the Docker cluster, manual configurations must be made to the generated Phoenix project (*appendix B*). Beyond the manual configurations and the Phoenix project setup boilerplate, the PostgreSQL database and remainder of the API are configured in a Docker compose file. This includes the endpoints for client requests and the handler functions to modify the database based on these requests *(appendix C)*.

2. Client script

An Elixir client script, executed locally outside of the docker cluster, generates concurrent requests to the Phoenix REST API *(appendix D)*. The script spawns a variable number of processes to simulate multiple clients making sequential HTTP requests to the API. These requests follow CRUD operations (Create, Read, Update, Delete) by creating an *item*, fetching a list of *items*, retrieving a specific *item*, updating it, and then deleting it. This sequence of requests iterates a variable number of times for each process.

The volume of requests sent to the API during a single execution of the script can be determined by $\mathrm{T} = \mathrm{NM}5$, where:

- $\mathrm{T} =$ the total number of requests
- $\mathrm{N} =$ the number of concurrent processes
- $5 =$ the constant number of requests within a single request sequence.
- $\mathrm{M} =$ the number of request sequences executed.

To assess the performance of the API and database under varying loads of client requests, the script can be executed using a range of values N and M.

3. Performance monitoring and visualization

In addition to the API and database, a Prometheus container is deployed within the Docker cluster. This is used to monitor the API's behavior under varying request loads and provide insights into its performance, scalability, and responsiveness during concurrent interactions.

Following an examination of the base metrics exposed by the API through ProQL queries *(appendix E)*, the following metrics have been identified and are considered for further analysis:

- Number of requests received
- Average number of requests received / 2s
- Average response time (ms) of requests received / 2s
- Number of additional server processes created
- Number of ready-queued server processes

Each metric tracks data throughout the duration of the client script's execution time, contributing as another key metric under consideration for the analysis.

To finely visualize these metrics, Prometheus is configured to scrape data from the API every second (*appendix F*). Due to limitations in the Prometheus platform, It should be noted that ProQL queries calculating rate must have an interval greater than the configured scrape rate.

4. Docker containers

Docker compose is used to create a multi-container cluster consisting of the Phoenix API, PostgreSQL database, and Prometheus monitoring tool *(appendix G)*. This setup streamlines the deployment of these components in a standardized environment. Docker compose also enables the generation and integration of the *Items* table in the PostgreSQL database and API, respectively. It also generates the API handler functions to modify and retrieve *items* in the database upon client requests (*appendix C*).

Using a standardized testing environment, such as the Docker cluster described, is beneficial when stress-testing an API with HTTP requests. This approach ensures a controlled and consistent environment for running tests and experiments. It also provides isolation between different services, replicating real-world scenarios more accurately while maintaining stability and reproducibility in testing conditions.

III. Experiment Definitions and Procedures

To evaluate concurrent performance both on the client and server side, the client script is executed with different values of N and M, as defined in the previous section. The test cases involve executing the client script once for each scenario, denoted by identifiers A to E in the following table:

Test case identifier	Number of concurrent processes (N)	Number of request iterations (M)	Calculated number of requests sent (T)
A	10	20	1,000
B	100	20	10,000
C	1,000	20	100,000
D	20	100	10,000
E	20	1,000	100,000

To prevent any data accumulation in the PostgreSQL database affecting performance, the database contents are entirely cleared between each test case.

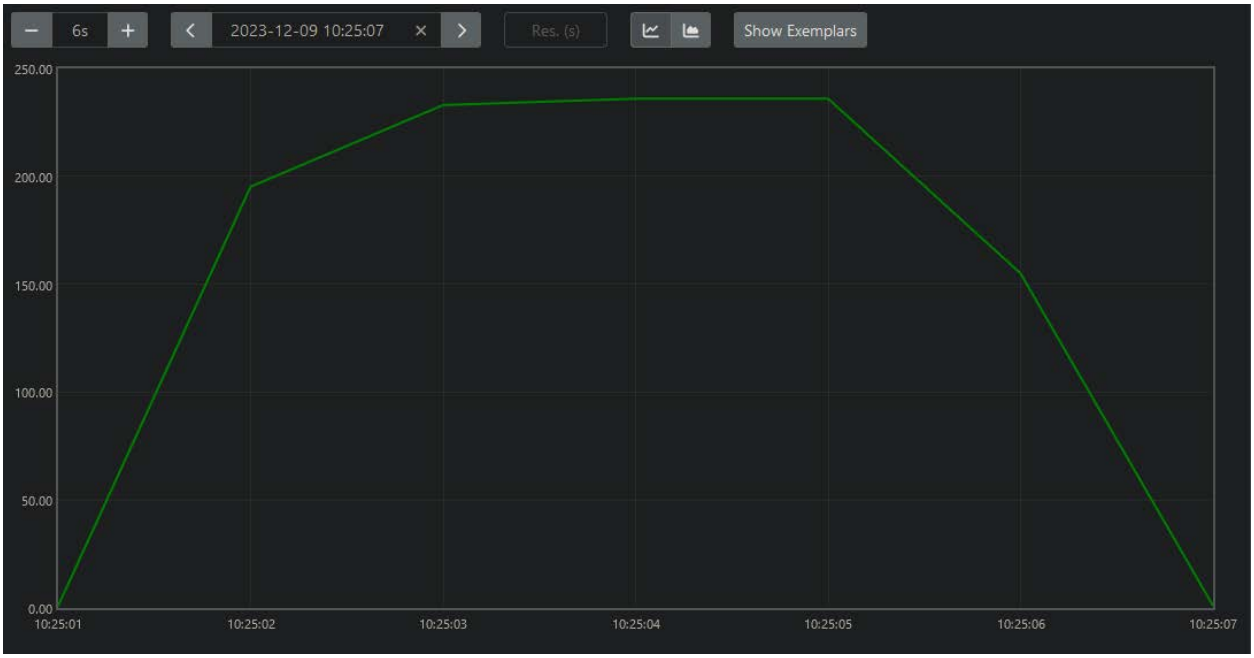
VI. Experiment Results and Data Collection

Test case A.

Client script run-time: 4.53s

Number of requests received: 1,266

Average number of requests received / 2s



Average response time (ms) of requests received / 2s

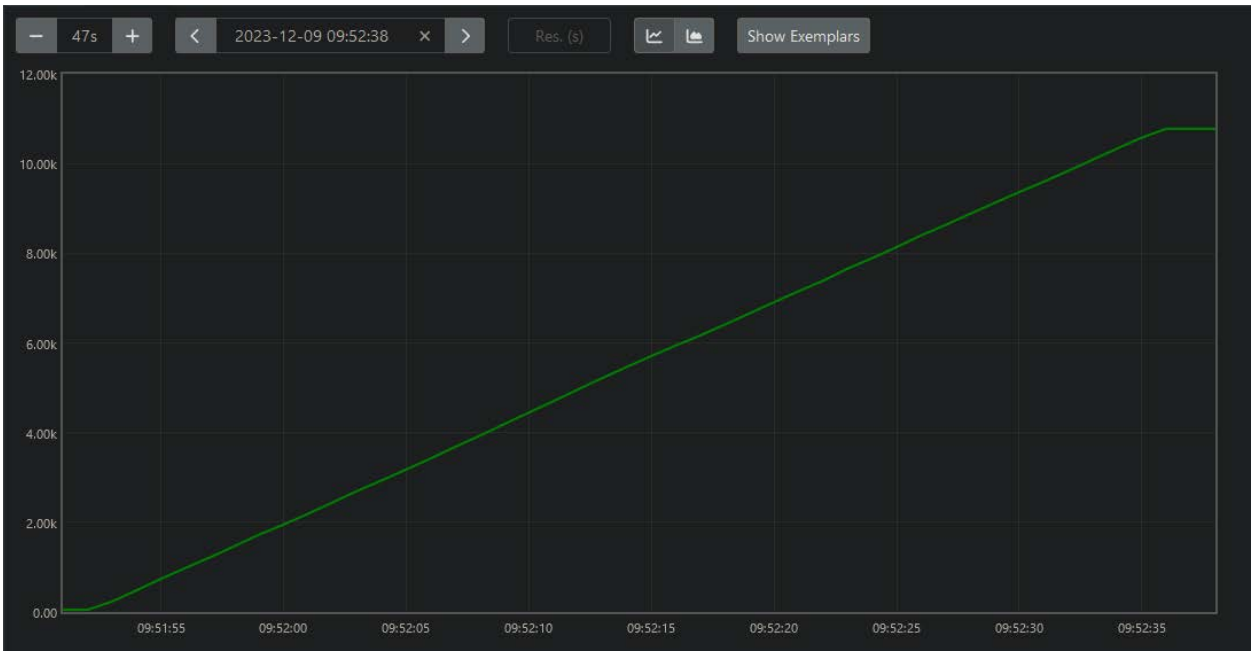
Number of additional server processes created

Number of ready-queued server processes

Test case B.

Client script run-time: 43.66s

Number of requests received: 10,775



Average number of requests received / 2s

Average response time (ms) of requests received / 2s

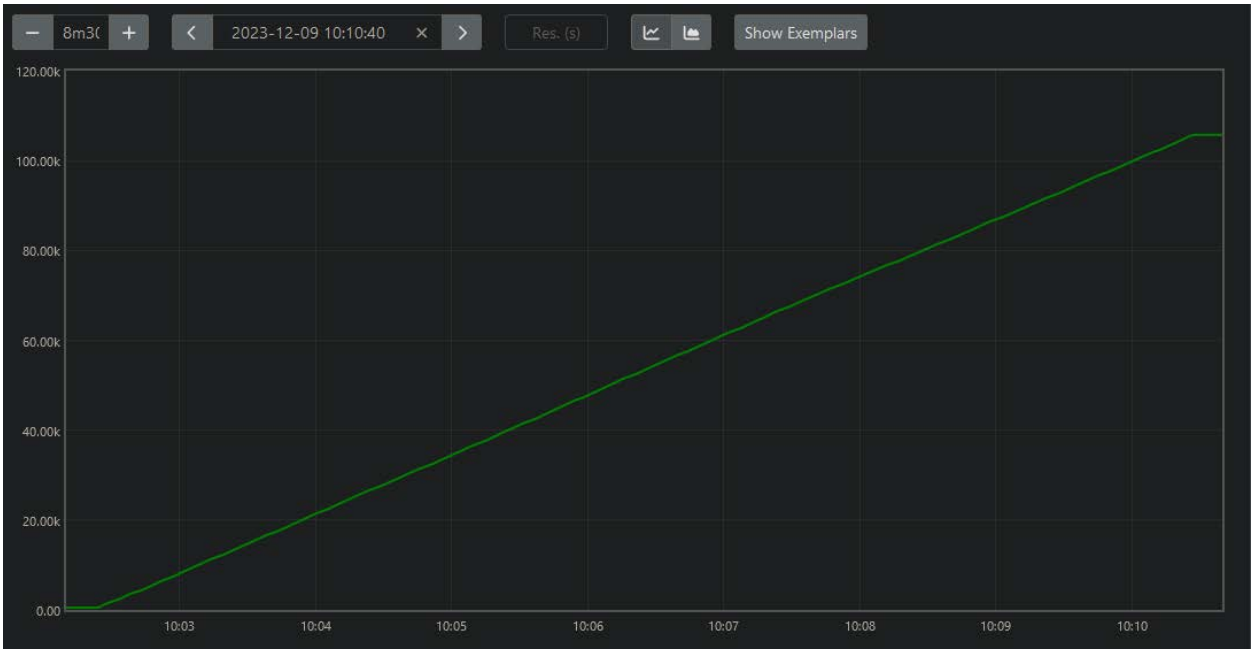
Number of additional server processes created

Number of ready-queued server processes

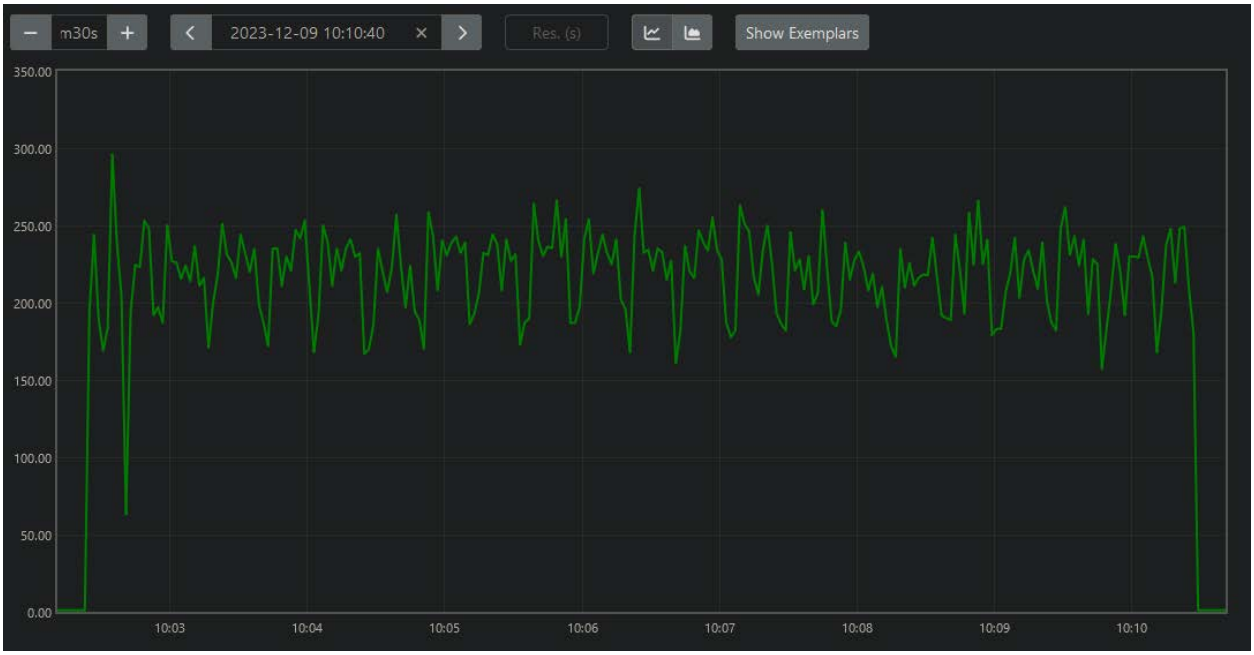
Test case C.

Client script run-time: 482.64s

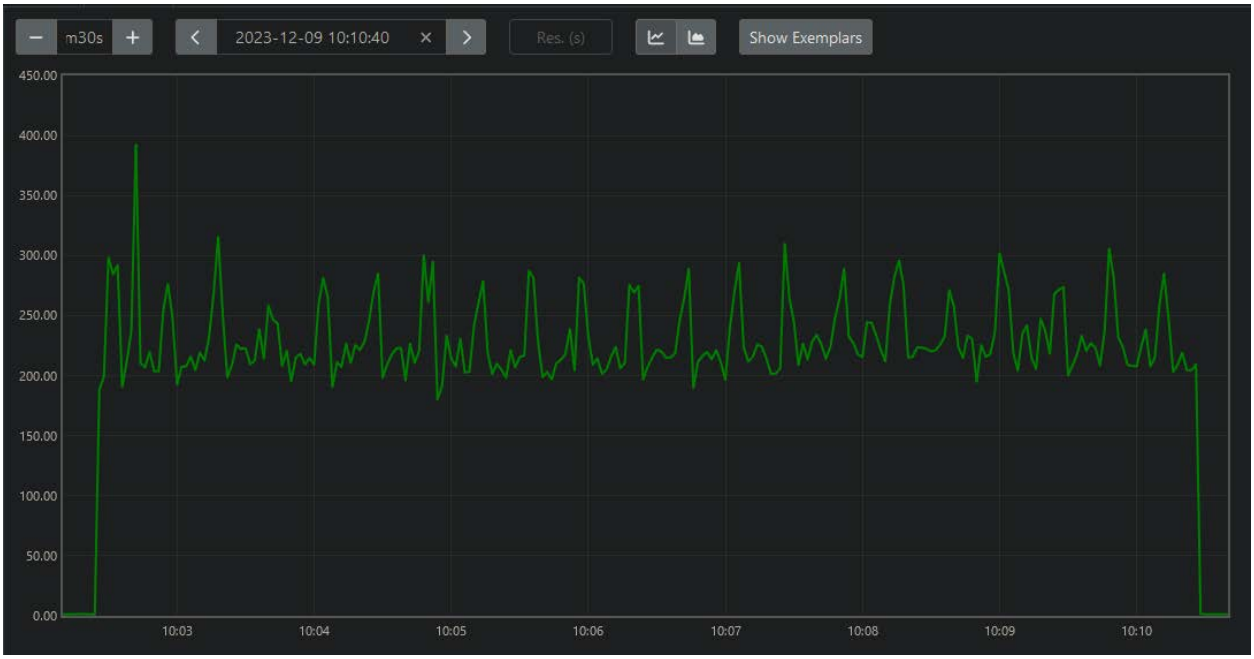
Number of requests received: 105,717



Average number of requests received / 2s

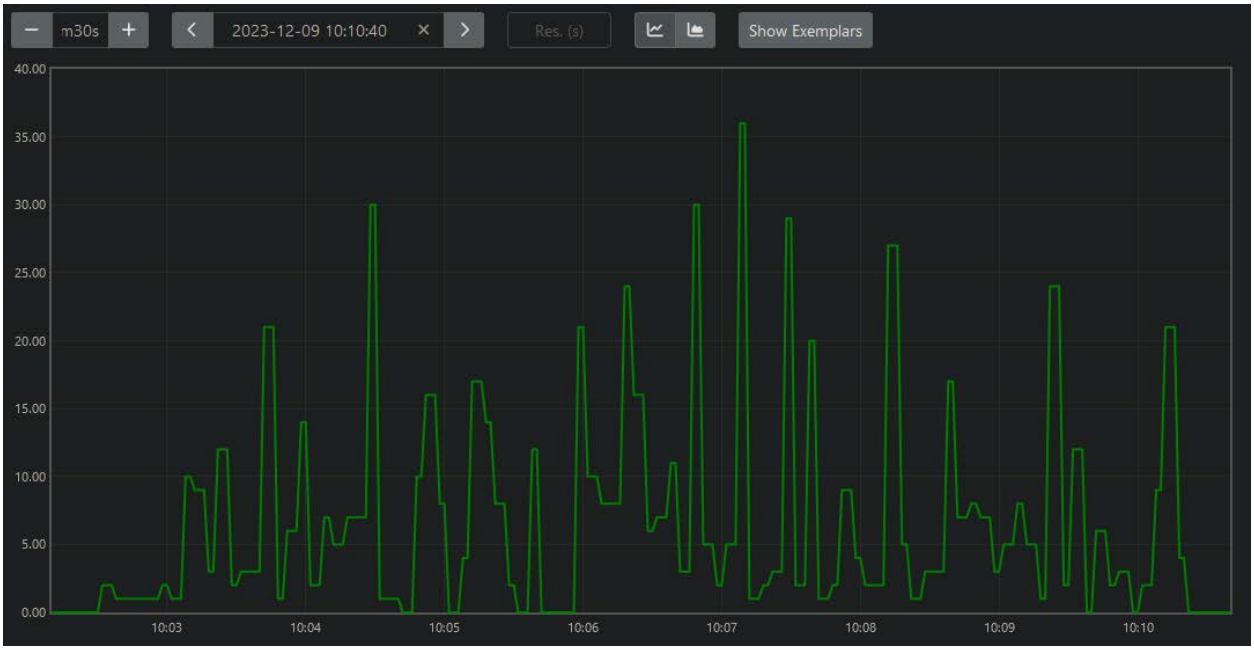


Average response time (ms) of requests received / 2s



Number of additional server processes created

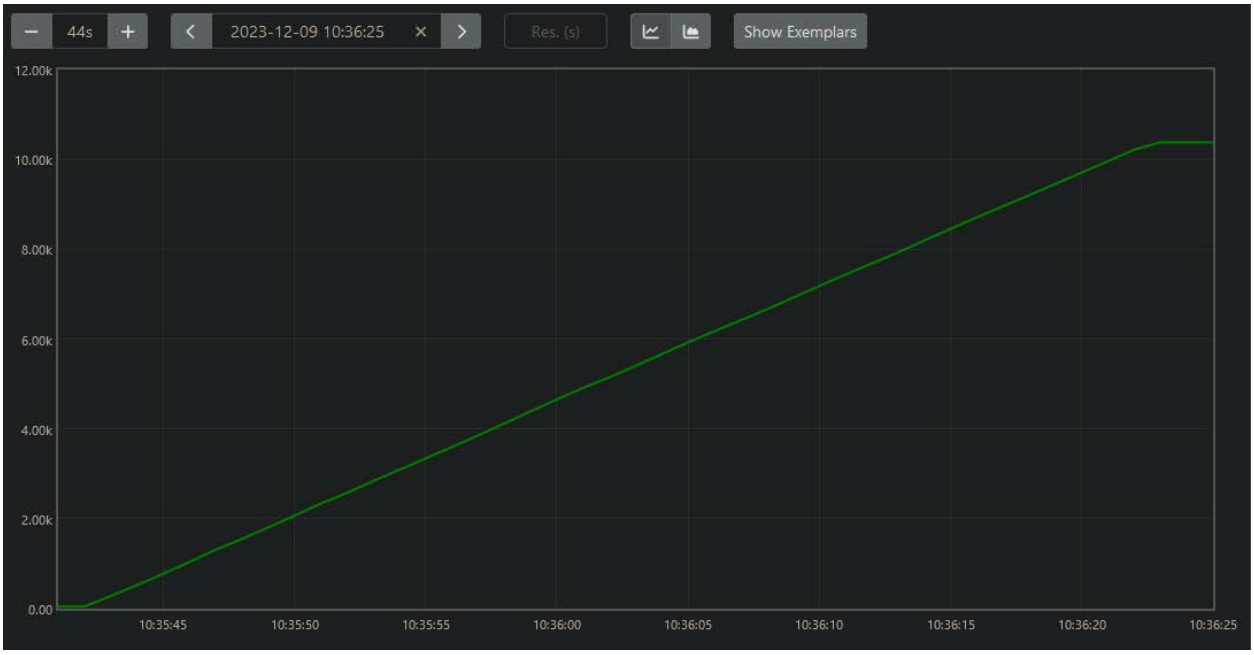
Number of ready-queued server processes



Test case. D.

Client script run-time: 40.65s

Number of requests received: 10,378

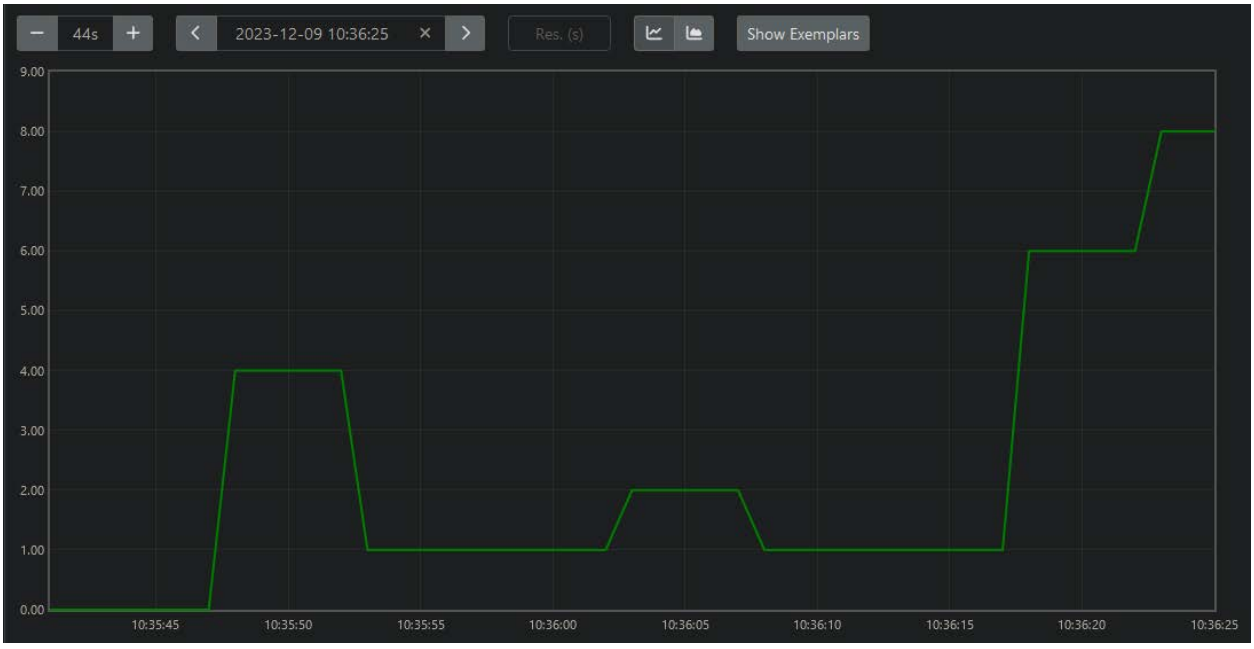


Average number of requests received / 2s

Average response time (ms) of requests received / 2s

Number of additional server processes created

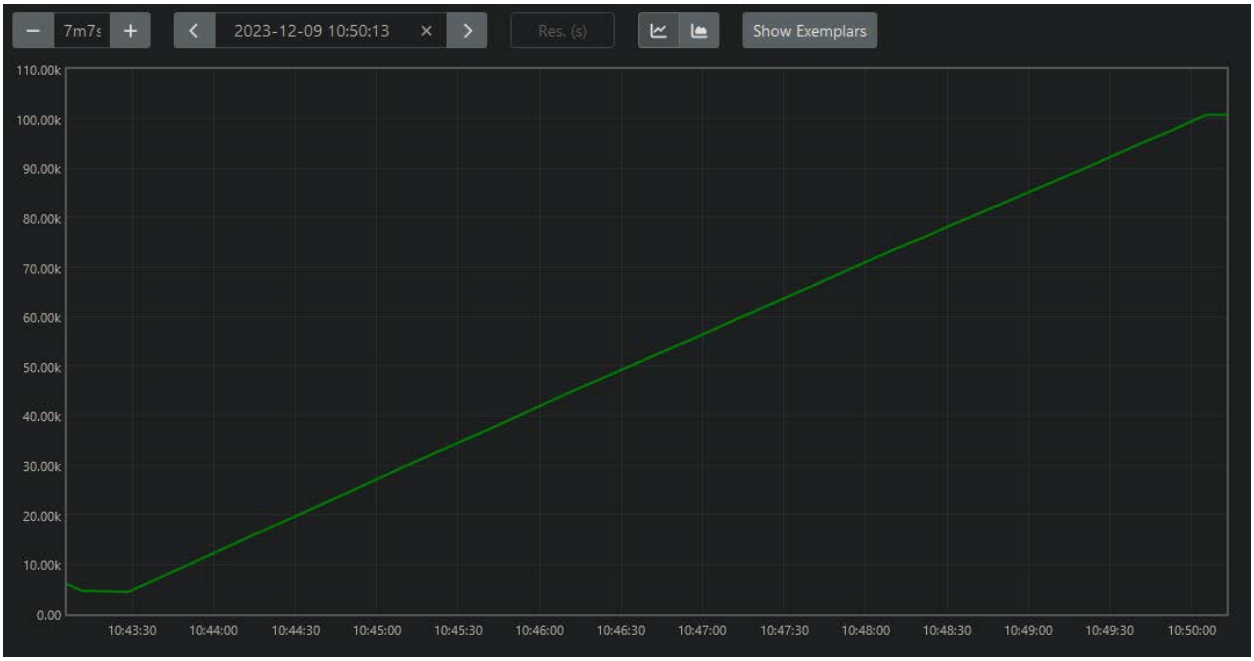
Number of ready-queued server processes



Test case E.

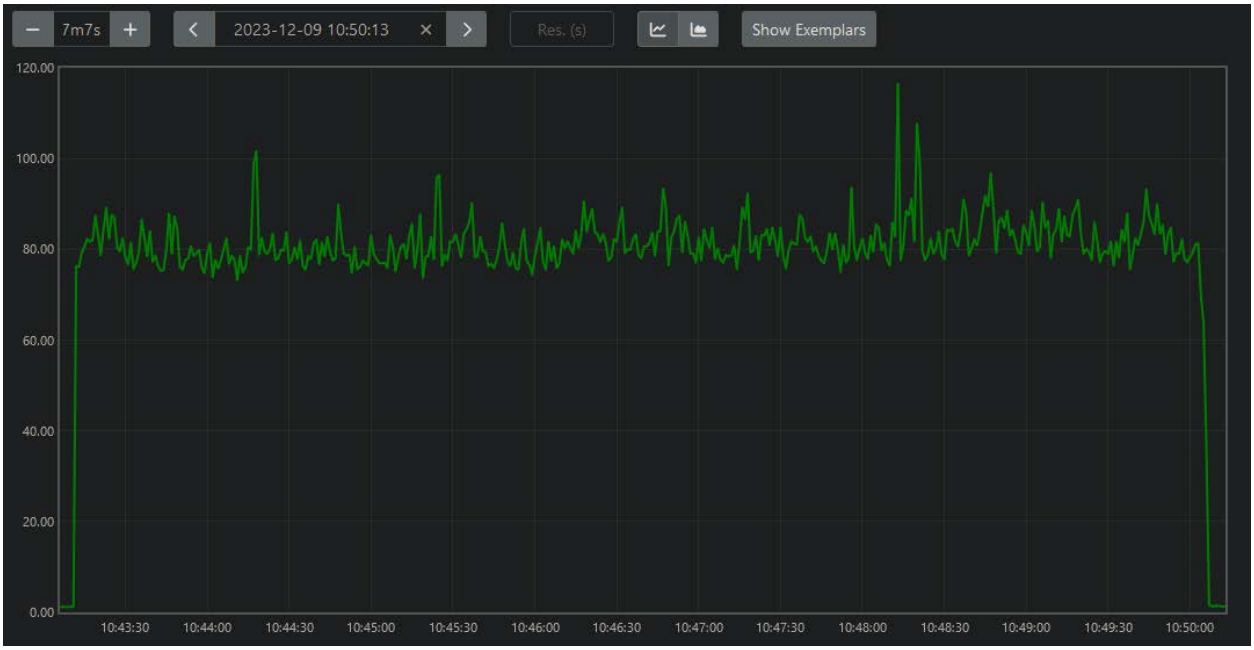
Client script run-time: 413.94s

Number of requests received: 100,763



Average number of requests received / 2s

Average response time (ms) of requests received / 2s



Number of additional server processes created

Number of ready-queued server processes

VII. Analysis and Conclusion

Analysis of the results indicates notable performance trends and insights into the API's behavior under varying levels of concurrency.

Test case	N	M	T	Requests received	Client script execution time (s)	Average requests received (approx.)	Average response time (ms) (approx.)	Peak additional server processes (approx.)	Peak ready-queued server processes
A	10	20	1,000	1,266	4.53	240	42	47	9
B	100	20	10,000	10,775	43.66	240	210	2,800	20
C	1,000	20	100,000	105,717	482.64	220	250	32,000	36
D	20	100	10,000	10,378	40.65	250	80	2,200	8
E	20	1000	100,000	100,763	413.94	230	90	20,000	18

The received requests, observed to exceed the sent requests in each test case, are likely the result of the continuous requests sent by Prometheus to scape information from the API. Negating this discrepancy, the observed requests received align will the calculated volume of request, T.

The client script execution time increased proportionally with the total number of requests sent. The average number of requests remained mostly consistent in each test case, aligning with the proportional increase in run-time.

The most notable differences can be seen in the average response times, number of additional server processes created, and number of ready-queued server processes. The results from each test case trend upwards with T, but do not scale proportionally like the execution time and requests received. Despite this, an upward trend seems to indicate significantly higher work capacities of the API, followed by degradations in performance and under heavy load conditions.

Analyzing the values across the pairs of test cases with inverse values of N and M reveals particularly interesting results. Though the total number of sent requests is equal in these pairs, the workload distribution among client processes seems to significantly impact execution times, server resource utilization, and the creation/queuing of server processes.

Test scenarios with higher concurrency (more client processes but fewer request iterations per process) result in longer execution time, higher additional server processes created, and higher queued server processes. Conversely, scenarios with lower concurrency (fewer client processes but higher request iterations per process) result in slightly shorter execution times and decreased server resource overhead. The most notable difference can be seen in the comparison of average response times, where the results of lower concurrency test cases are significantly shorter than their high concurrency counterparts.

For example, test case B, a higher concurrency scenario, exhibits longer execution time and average response time, followed by a notably higher peak number of additional server processes created. The opposite can be said for these results regarding test case D, a lower concurrency scenario. This pattern is evident in the comparison of test cases C (higher concurrency), and E (lower concurrency), but to an even greater degree.

Regarding these discrepancies, the longer execution times and higher response times in higher concurrency scenarios may stem from increased parallelism, requiring the server to manage numerous shorter client request sequences. This could lead to higher server resource contention and queuing, contributing to the observed performance lag. Conversely, the difference in lower concurrency scenarios could be attributed to more efficient handling of requests with decreased contention, resulting in significantly shorter response times. Given these speculations, the disparities highlight the trade-offs between parallelism and resource utilization.

Conclusions drawn from the results suggest that while the Phoenix REST API demonstrated the ability to manage concurrent requests across various test scenarios, there were indications of performance degradation under exceptionally high concurrency. As concurrent processes increased, average response times and server resource usage increased, signifying potential limitations in handling excessive concurrent loads. These findings emphasize the importance of optimizing API performance, particularly under extreme concurrent conditions, to maintain acceptable response times and throughput. Further exploration into server resource utilization and error handling mechanisms could enhance understanding and inform strategies for optimizing API performance under varying levels of concurrency.

VIII. Appendices

A. Phoenix project generation command

```
mix phx.new project --no-install --no-assets --no-html --no-dashboard --no-live --no-mailer
```

B. Phoenix project manual configuration

dev.ex:

1. Change the root end-point port from 127.0.0.1 (local host) to 0.0.0.0 for Docker container connection.
2. Change database username, password, hostname to match the information in the docker compose file.

router.ex:

1. Modify the API scope block as follows:

```
scope "/api", ProjectWeb do
    pipe_through :api
    resources "/items", ItemController
  end
```

C. API generated handler functions

Get items:

```
# GET localhost:5000/api/items
def index(conn, _params) do
  items = Items.list_items()
  render(conn, :index, items: items)
end
```

Create item:

```
# POST localhost:5000/api/items
# Body:
# {
#   "item": {
#     "value": <Boolean, default false>
#   }
```

```
# }
def create(conn, %{"item" => item_params}) do
  with {:ok, %Item{} = item} <- Items.create_item(item_params) do
    conn
    |> put_status(:created)
    |> put_resp_header("location", ~p"/api/items/#{item}")
    |> render(:show, item: item)
  end
end
```

Get item by ID:

```
# GET localhost:5000/api/items/:id
def show(conn, %{"id" => id}) do
  item = Items.get_item!(id)
  render(conn, :show, item: item)
end
```

Update item by ID:

```
# PUT localhost:5000/api/items/:id
# or
# PATCH localhost:5000/api/items/:id
# Body:
# {
#   "item": {
#     "value": <Boolean, default false>
#   }
# }
def update(conn, %{"id" => id, "item" => item_params}) do
  item = Items.get_item!(id)
  with {:ok, %Item{} = item} <- Items.update_item(item, item_params) do
    render(conn, :show, item: item)
  end
end
```

Delete item by ID:

```
# DELETE localhost:5000/api/items/:id
def delete(conn, %{"id" => id}) do
  item = Items.get_item!(id)

  with {:ok, %Item{}} <- Items.delete_item(item) do
    send_resp(conn, :no_content, "")
  end
```

```
end
```

D. Elixir client script

client.ex:

```
defmodule Client do
  use HTTPoison.Base

  @url "http://localhost:5000/api/items"

  def main(args) do
    n = Enum.at(args, 0) |> String.to_integer()
    m = Enum.at(args, 1) |> String.to_integer()

    startTime = System.monotonic_time(:millisecond)

    processes = Enum.map(1..n, fn _ ->
      Task.async(fn ->
        HTTPoison.start()
        sendRequests(0, m)
      end)
    end)

    Task.await_many(processes, :infinity)
    IO.puts("Done (#{Float.round((System.monotonic_time(:millisecond) - startTime) /
1000, 2)}s)")
    System.halt(0)
  end

  def sendRequests(loopCount, loopMax) do
    case loopCount <= loopMax do
      true ->
        id = createItem()
        getItems()
        getItem(id)
        updateItem(id)
        deleteItem(id)
        sendRequests(loopCount + 1, loopMax)
      false -> :ok
    end
  end
```

```
  def createItem do
    case HTTPoison.post(@url, "{\"item\": {\"value\": true}}", [{"Content-Type",
"application/json"}]) do
      {:ok, %HTTPoison.Response{status_code: 201, body: body}} ->
        case Jason.decode(body) do
          {:ok, parsed_body} ->
            case Map.get(parsed_body, "data") do
              %{"id" => id} -> id
            end
          {:error, _} -> 0
        end
      {:error, _} -> 0
    end
  end

  def getItems do
    HTTPoison.get(@url)
  end

  def getItem(id) do
    HTTPoison.get("#{@url}/#{id}")
  end

  def updateItem(id) do
    HTTPoison.put("#{@url}/#{id}", "{\"item\": {\"value\": false}}", [{"Content-
Type", "application/json"}])
  end

  def deleteItem(id) do
    HTTPoison.delete("#{@url}/#{id}")
  end
end

Client.main(System.argv())
```

E. ProQL queries

Number of requests received

```
increase(cowboy_request_duration_milliseconds_count[<time>])
```

Average number of requests received / 2s

```
rate(cowboy_request_duration_milliseconds_count[2s])
```

Average response time (ms) of requests received / 2s

```
rate(cowboy_request_duration_milliseconds_sum[2s]) /
rate(cowboy_request_duration_milliseconds_count[2s])
```

Number of additional server processes created

```
increase(erlang_vm_system_counts_process_count[<time>])
```

Number of ready-queued server processes

```
erlang_vm_run_queue_total
```

F. Prometheus configuration

Note: The running Prometheus container is mapped by default to port 9090 of the local host

prometheus.yml:

```
global:
  scrape_interval: 1s

scrape_configs:
  - job_name: 'server'
    static_configs:
      - targets: ['server:4050']
```

G. Docker configuration

dockerfile:

```
FROM elixir:latest
WORKDIR /app
RUN mix local.hex --force && \
    mix local.rebar --force
RUN mix archive.install hex phx_new --force
COPY . .
RUN mix deps.get
RUN mix phx.gen.json Items Item items value:boolean
RUN mix compile
CMD ["mix", "phx.server"]
```

docker-compose.yml:

```yaml
version: '3.9'
services:
  server:
    container_name: server
    build: .
    ports:
      - "5000:4000"
    depends_on:
      - database
    environment:
      PGDATABASE: database
      PGUSER: username
      PGPASSWORD: password
      PGHOST: database
    command:

  database:
    image: postgres:latest
    container_name: database
    ports:
      - "5432:5432"
    environment:
      POSTGRES_DB: database
      POSTGRES_USER: username
      POSTGRES_PASSWORD: password

  prometheus:
    image: prom/prometheus:latest
    container_name: prometheus
    depends_on:
      - server
    ports:
      - "9090:9090"
    volumes:
      - prometheus_data:/etc/prometheus
    command:
      - '--config.file=/etc/prometheus/prometheus.yml'

volumes:
  prometheus_data:
```