Seminar Report

Cloud Infrastructure with Go

Valerius Mattfeld

MatrNr: 11580056

Supervisor: Jonathan Decker, M.Sc.

Georg-August-Universität Göttingen
Institute of Computer Science

April 6, 2024

Abstract

In cloud computing, even the tiniest performance bottleneck can have costly consequences. This report explores a possible bottleneck when receiving computed functions from an HTTP server.

This bottleneck is suspicious of causing extra latency, even for small function invocations in serverless function environments. The goal is to determine if such a bottleneck exists and if switching out parts of an implementation helps to resolve this problem.

Invoking API endpoints, or in that case, serverless functions, are limited to their respective web server handling the incoming invocation requests. Therefore, this part is crucial when optimizing performance for latency.

Switching out web frameworks for known implementations of serverless functions in the same programming language - which is Go in this report - might help to find the root cause of the problem.

Complete HTTP roundtrips from a benchmarking server over a load balancer to a function container were invoked on mass, with several endpoints challenging different parts of the hardware and infrastructure. Different web frameworks behind the endpoints were switched out and evaluated. A significant part of the request roundtrip time resides in creating the response for the requester; this applies to almost all web frameworks tested and all endpoints.

Declaration on the use of ChatGPT and comparable tools in the context of examinations

In this work I have used ChatGPT or another AI as follows:

- ☐ Not at all
- ☐ During brainstorming
- ☐ When creating the outline
- ☐ To write individual passages, altogether to the extent of 0% of the entire text
- ☐ For the development of software source texts
- ☐ For optimizing or restructuring software source texts
- ☐ For proofreading or optimizing
- ☑ Further, namely: GitHub Copilot for assistance during using and learning the `pandas` python package

I hereby declare that I have stated all uses completely.
Missing or incorrect information will be considered as an attempt to cheat.

Contents

List of Tables

List of Figures

List of Listings

List of Abbreviations

HPC	High-Performance Computing
FaaS	Functions-as-a-Service
RPC	Remote Procedure Call
VM	Virtual Machine
HTTP	Hypertext Transfer Protocol

1 Introduction

1.1 Motivation

Serverless functions, also known as Functions-as-a-Service (FaaS), are services that free the deployer of several obligations.

Those obligations include but are not limited to, the maintenance of an application server infrastructure, the ability to scale well with user demand, and cost-effectiveness because the underlying hardware resources are more efficiently used.[Clo24][Inc24]

Proprietary services, like AWS Lambda[1], Cloudflare Workers[2], and to some extent, Vercel Functions[3], make the deployment of application code an accessible task. Furthermore, open source FaaS solutions, like nuclio[23d], give a bigger insight into how such an architecture might look by making heavy use of Kubernetis[23c] and its benefits.

Finding performance bottlenecks in such an architecture could benefit a non-trivial amount of users, and FaaS providers, as well as help mitigate hardware allocation inefficiency.

1.2 Goals and Contributions

This report aims to investigate a known problem, which arises in unusually high latencies when short function executions are being sent back to the function caller; the inspiration for this report is found in [DKK22].

The experimental infrastructure recreates a minimal functioning setup on Remote Procedure Call (RPC) handling between three Virtual Machine (VM)s, simulating a function call in a FaaS environment, utilizing core packages and alternatives where the core logic of bigger FaaS frameworks resides.

By substituting various prominent Go web frameworks alongside their corresponding Hypertext Transfer Protocol (HTTP) implementations, which are commonly utilized within major open FaaS frameworks, we endeavor to ascertain any discernible enhancements or regressions within the latency domain.

1.3 Structure

After a brief introduction of the relevance of Go in server infrastructure code and currently relevant Open Source FaaS frameworks in section 2 and **??**, this report elaborates on the setup of the experiment, choice of web frameworks and identifying the room for optimization in section 3.

The report continues with the benchmarking of the experimental setup and data evaluation, section 5. The results and applicability for cluster environments are discussed in section 6.

[1]`https://aws.amazon.com/lambda/`, April 6, 2024
[2]`https://workers.cloudflare.com/`, April 6, 2024
[3]`https://vercel.com/docs/functions`, April 6, 2024

2 Background

2.1 Serverless Functions

Serverless functions represent a model in which the cloud provider dynamically manages the allocation of available hardware resources. These functions are triggered by events or requests and execute user-provided code.

The characteristics of serverless functions are split into several key traits.

Firstly, we have the event-driven execution, which ensures the execution of a function only when necessary, therefore, helping to ensure efficient resource allocation, [Sch+22]

Then, automatically scaling the available resources to a specific deployment allows dynamic scaling without manual intervention, [RPT22].

Serverless functions are commonly encapsulated within container images, providing a portable and isolated environment for execution, [Bro+23]. This also allows the functions to be deployed in various programming languages (The right tool for the right job), [JC18].

2.2 Current Open Source FaaS frameworks

Since various Open Source FaaS frameworks utilize the advantages of Kubernetes, some frameworks stand out the most. Those include, but are not limited to:

- knative.dev (supporting languages like Go, Elixir, Java, etc.), [23b]
- nuclio.io - With a data science focus and completely written in Go, [23d]
- openfaas.com - Also using Go, [23e]
- fission.io - Built with Go, [23a]
- openwhisk.apache.org - Implemented in Scala, [23f]

The dominance of Go in those frameworks could be explained by Kubernetes is also written in Go,[23c] and Go's ability to have an excellent developing experience regarding concurrency, e.g., `goroutines`, and `channels`.

3 Methodology

3.1 Setup of the experiment

The experimental setup consists of three VMs, each inhabiting a specific role. One simulates the function caller, the second will be the load-balancer, web server, or proxy for the actual function deployment, and the last will simulate a node responsible for the user-provided function execution.

3.1.1 Machine One - Function Caller

To simulate a serverless function call from a user or client, the first VM is created with 2 VCPUs, 8 GB of RAM, and 40 GB of storage, running Ubuntu Linux 22.04.

The load testing software and data accumulation software reside inside this machine.

3.1.2 Machine Two - HTTP Server

Requests from machine one will be sent to this machine. Running with 4 VCPUs, 4 GB of RAM, and 40 GB of storage on Ubuntu Linux 22.04, it will serve four Docker containers over the host network to all machines inside the internal network.

Each Docker container hosts a different implementation of a web server written on Go on a different port, which ranges from `5555` to `5559`.

3.1.3 Machine Three - RPC Server

When a function from machine one gets called on machine two, machine two invokes an RPC call to this machine. It is running with the same specifications as machine two, but with the difference that only one Docker container is running here.

The implementation of the RPC server, doing the actual processing of the function request.

Upon the respective functions' completion, the result is sent back to machine two, which completes the invocation for the function caller and sends the response with the function result.

3.2 Choice of web frameworks

Traditionally, HTTP is used with some form of REST API to interact with the deployed functions on the server cluster. In the nuclio FaaS application, an HTTP endpoint is called a *Trigger*.[Sch+22]

The underlying HTTP handler of the respective implementation is the crucial part where the bottleneck can occur.

The experimental setup implements a choice of web frameworks to test a variety of HTTP handling implementations.

3.2.1 `net/http`

The Go-native `net/http`[4] package is a built-in solution for developing HTTP clients and servers with Go, eliminating the need for a third-party package or solution. The Go developers maintain it and use Go-native features like goroutines and channels to ensure concurrency in the software developers' implementation.[5] It should serve as a standard for evaluating the performance of other web frameworks.

3.2.2 Iris

Iris[6] is built on the net-http package, providing the developer an Express.js-like API. It claims to be the fastest HTTP/2 framework compared to others mentioned in this report.[7]

However, since the setup is built on HTTP/1.1, we are not interested in this feature.

Iris comprises the responses by default, making the response time possibly slower.[8]

[4]`https://pkg.go.dev/net/http`, April 6, 2024
[5]`https://go.dev/solutions/cloud`, April 6, 2024
[6]`https://www.iris-go.com/#features`, April 6, 2024
[7]`https://github.com/kataras/server-benchmarks#server-benchmarks`, April 6, 2024
[8]`https://www.iris-go.com/docs/#/?id=api-examples`, April 6, 2024

3.2.3 Gin

Gin[9] is a Go webserver framework that uses a custom implementation of the `httprouter`[10] module, claiming to implement a zero-allocation router.

Their benchmarks indicate that Gin is not the fastest framework in every regard but an allrounder for various use cases and the go-to solution for memory-efficient applications.[11]

3.2.4 Echo

Echo[12] is a Go web framework that implements a framework-native HTTP router module implementation. Like other web frameworks, it claims to be suitable for high-performance, highly scalable applications.[13]

The maintainers advertise extendability due to several community-maintained framework extensions and - like Gin - no dynamic memory allocation.[14] Their benchmarks contend that the Echo is more efficient than Gin.[15]

3.2.5 Fiber

Perhaps the most promising framework in this listing is Fiber.[16] It uses the same underlying HTTP engine as nuclio, namely fasthttp, making it the best candidate to compare other frameworks to in the reports' use case.[17]

Their benchmarks indicate an unmatched performance for a production-ready library. It comes in third place on the TechEmpower ranking[18], after `gnet` - written explicitly for HTTP benchmarking[19] and currently unsuitable for real-world server applications and a fasthttp fork. Moreover, Fiber claims to have the lowest request latencies among other frameworks,[20] which is a focal point in this report.

3.2.6 Gorilla

Another popular framework is Gorilla[21], notably the Gorilla/mux package. It was maintained until 2021, and the framework found new core maintainers as of August 2023. Since this framework's maturity and recent updates are open questions, this report does not benchmark it.[22]

[9]https://gin-gonic.com/
[10]https://github.com/julienschmidt/httprouter
[11]https://github.com/gin-gonic/gin/blob/master/BENCHMARKS.md
[12]https://echo.labstack.com/
[13]https://echo.labstack.com/
[14]https://github.com/labstack/echo?tab=readme-ov-file#feature-overview
[15]https://github.com/labstack/echo?tab=readme-ov-file#benchmarks
[16]https://gofiber.io/
[17]https://docs.gofiber.io/api/app#server
[18]https://docs.gofiber.io/extra/benchmarks/#plaintext
[19]https://github.com/panjf2000/gnet?tab=readme-ov-file#benchmarks-on-techempower
[20]https://gofiber.io
[21]https://github.com/gorilla/mux
[22]https://gorilla.github.io/blog/2023-07-17-project-status-update/

3.3 Network architecture

The machines reside in an internal network inside the GWDG Cloud environment[23] and communicate with each other via their respective internal IP addresses, ensuring not only physical proximity but also staying true to Kubernetes clusters, usually staying inside one location.[24]

The roundtrip of the function invocation, as shown in Figure 1,

1. The client, which in that case is the Benchmarking Server, or VM 1, requests a function invocation on an endpoint of the HTTP server with their respective parameters.
2. Then, the HTTP server handles the request by extracting optional function parameters and invoking an RPC call to the RPC server.
3. The RPC server executes the function with the given parameters.
4. The result of the invocation is sent back to the HTTP server.
5. The HTTP server builds the response object, including a status code, and optionally parses the result in the desired format. Finally, it sends it back to the requester, the Benchmarking Server.

Since this is a minimal and simplified version of a production-ready architecture, aspects like security measures with firewall rules and scalability outside this schema are not considered.

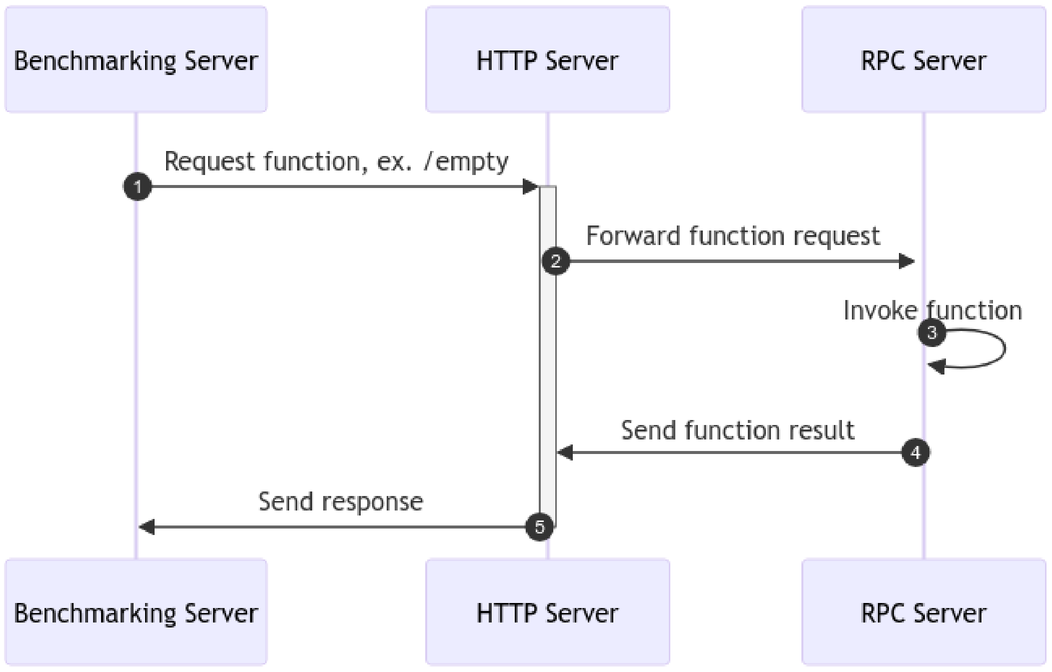

Figure 1: Request roundtrip of a function invocation

[23]https://cloud.gwdg.de
[24]https://kubernetes.io/docs/setup/best-practices/multiple-zones/

3.4 Criteria for measuring latency and identifying the bottleneck

3.4.1 Metrics

Measuring function execution time within the infrastructure is vital to evaluating the potential bottleneck's performance. Therefore, several factors are assessed and logged with critical precision.

Log Timestamps Logging each function call and vital instance of program execution gives precise insight into how and when a function execution has occurred, making them essential markers in identifying the duration spent at each stage.

Network Latency Time taken between the Benchmarking server and the HTTP server will give insight into how long the roundtrip lasted.

HTTP server processing time The start of the function call, excluding constant factors like ID generation, is measured with a start and end time, including the trip to the RPC server and back. It is the time between step 2 and the receiving step 4, as shown in Figure 1.

RPC server processing time The processing of the actual "user" provided function is also measured to build suitable latency deltas.

CPU and RAM threshold Hardware allocations, which are also monitored, could be critical to the performance impact. They directly influence the computational resources available and can slow down function invocations' processing times when excessive parallel requests occur.

3.4.2 Test Endpoints

Various functions at the RPC server are implemented to simulate a real-life workload while focusing on different aspects of a specific workload.

`/empty` An empty function that does nothing. It serves as a reference point for other endpoints.

`/sleep` A function that waits one second and terminates.

`/math` The math endpoint takes a parameter $n, 1 \leq n \leq 2^{64} - 1$ (n is an unsigned 64-bit integer) and executes a Monte Carlo algorithm to approximate π to a sample size of n. This endpoint serves the purpose of stimulating CPU resource exhaustion.

`/image` The function image handles file I/O, applies transformations, and streams the resulting binary data back to the requester.

3.5 Identifying room for optimization

As the previous papers suggest, a high-latency chokepoint between steps 4 and 5 in the process shown in Figure 1 is possible.

The report investigates whether using different frameworks helps circumvent the problem.

The result could indicate an architectural issue if the bottleneck is confirmed and using different web frameworks does not mitigate the problem.

4 Implementation

The experiment aims to mimic a simple serverless function invocation in its basic setup while staying true to a real-life implementation.

In an optimal case, the infrastructure of the different applications is deployed inside one data center in a single location. Moreover, container virtualization and communication over a private LAN are essential.

The source code is available in the following repository: `https://github.com/valerius21/scap-2024`

4.1 Experiment setup vs. Open Source FaaS frameworks

4.1.1 Experiment setup

As shown in Figure 1, the HTTP and RPC processing servers reside in separate VMs provided by the data center. Those are the focus of the performance benchmark.

Each VM runs the Docker[25] virtualization software with at least one application container that communicates with either HTTP or TCP Websockets with each other or the client. The Prometheus Node Exporter software[26] is installed on all systems in conjunction with Grafana[27] for monitoring purposes.

Additionally, the setup includes a benchmarking server, which simulates function invocations from a user with minimal latency. It hosts multiple tools for benchmarking, log accumulation, and monitoring each server.

4.1.2 Open Source FaaS frameworks

This section focuses on nuclio, a data science-focused serverless functions framework that can run on Kubernetes or Docker Swarms.

Kubernetes clusters can be deployed on physical servers or virtual machines in the data center. The cluster comprises a control plane (API server, scheduler, controller manager) and worker nodes that run the containerized applications. It can run on networking solutions that enable communication between pods and services within the cluster.

Also, one core trait of Kubernetes is its ability to auto-scale cluster-wide, horizontally or vertically, for certain pods. A load balancer spreads incoming traffic between multiple pods to efficiently utilize available resources.

[25]`https://www.docker.com/`
[26]`https://prometheus.io/docs/guides/node-exporter/`
[27]`https://grafana.com/`

Nuclio is a scalable serverless function solution that aims to handle a significant amount of throughput and leverages Kubernetes' benefits.

4.1.3 Comparission

The experimental setup misses several core features, like factual load-balancing between multiple machines and container scaling of Kubernetes/nuclio, and mimics the infrastructure of a single request flow. The structure helps to isolate the problem to a limited number of points of failure or performance issues. The core networking and VM configuration of receiving a function request on an HTTP server, which mimics a load-balancer in a theoretical cluster, and the RPC server, which does the actual processing, is sufficient to investigate the problem.

4.2 Initial approach for creating an artificial round-test

The initial versions of the experimental implementation used a Message Query service with a PubSub Pattern between the HTTP and RPC servers to communicate function invocations over Redis channels. The configuration is displayed in Figure 2.

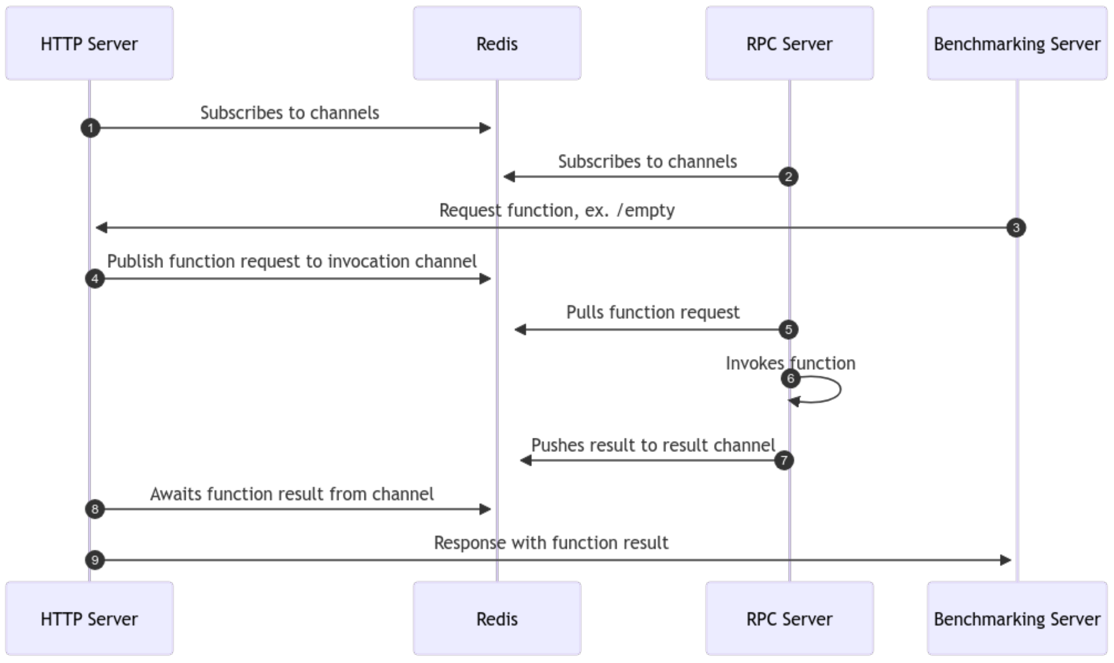

Figure 2: Request roundtrip of a function invocation with a Redis PubSub instance

In contrast, to direct RPC invocation over TCP and the Go-native `net-rpc` library, the Redis-dependent solution offers little to no benefit regarding maintainability, simplicity, and bottleneck discovery.

Therefore, this configuration was discarded in favor of the direct approach, as shown in Figure 1.

4.3 Architecture of the experiment code

Listing 1 shows the core architecture of the experiment software.

The entry point is found in `cmd/main.go`, which allows the server to run in either load-balancer or RPC server mode. This approach has the advantage that one container, or running executable, can be deployed on both configuration-critical machines, HTTP and RPC, as shown in Figure 1.

The repository defines a `pkg` directory, which houses the `fns` and `webserver` packages.

fns include the pure implementations of the user-simulated-deployed business logic in their respective files. Each filename corresponds to the endpoints mentioned in section 3.4.2.

The webserver packages have implementations for each tested webserver framework. The command-line flag used when executing the entry point determines which server is run.

Each server (re-)uses an RPC client implementation corresponding to the RPC servers' endpoints. Both, the RPC server and RPC client code are located in `pkg/fns/srpc`.

4.4 Reproducing the experimental setup

Reproducing the setup involves using Ansible[28]. Three Ubuntu VMs on separate IPs and their discoverability to each other are required. Details on how and when which Ansible-Playbook needs to run are found in the `ansible/README.md` file, see listing 3.

The experiment involved more software installed on the machines than shown in the repository[29]. This software includes logging and monitoring services, which are left out of the repository code on purpose because the repository code aims to provide a barebones starter and the source for the webserver code used.

The data accumulation process and configuration are elaborated in section 5.

5 Benchmarking

This section starts with the data accumulation process and follows with a description of the load-testing process. We briefly explore the optimal number of concurrent connections and continue to the log-processing section. After that, we explore the heart of this report - the benchmarking results. We examine each part of the testing infrastructure in terms of their durations and look at the resulting time deltas, where the possible bottleneck could occur.

5.1 Data accumulation and infrastructure

In the early stages of developing the testing software, benchmarks caused the docker containers to fully occupy the host machine's disk space with logs.

This problem was circumvented using a Docker logging driver, Loki[30], which is compatible with the Grafana dashboard. Loki's integration allowed a deep and detailed exploration and querying of the logs in specific timeframes with optional filters.

The Loki server and Grafana are running on the benchmarking server. This reduces the performing VMs, HTTP, and RPC to one purpose only—running the testing software.

[28] `https://www.ansible.com/`

[29] `https://github.com/valerius21/scap-2024`

[30] `https://grafana.com/docs/loki/latest/`

The logs were extracted mainly through the `logcli` tool, developed by the Loki and Grafana maintainers.[31]

5.2 Load testing

Upon discovering that this tool, like its alternatives, e.g., `bombardier`[32], cannot save detailed timestamps, a custom solution, a simple benchmarking tool testing the test infrastructure, was written and used. It is available under the following repository: `https://github.com/valerius21/yabt`.

The produced logs are part of the dataset in section 5.4.

5.2.1 Finding the optimal number of connections

One fundamental question is how many concurrent connections the test infrastructure can handle. Too many connections could stall the response time. Therefore, this number is of moderate importance.

The benchmark tool was tested on each framework's empty endpoint, and Figure 3 shows the average of the requests per second corresponding to the number of connections.

The resulting plot, shown in figure 3, indicates no significant benefit to using more than 150 concurrent connections, the number used in benchmarking.

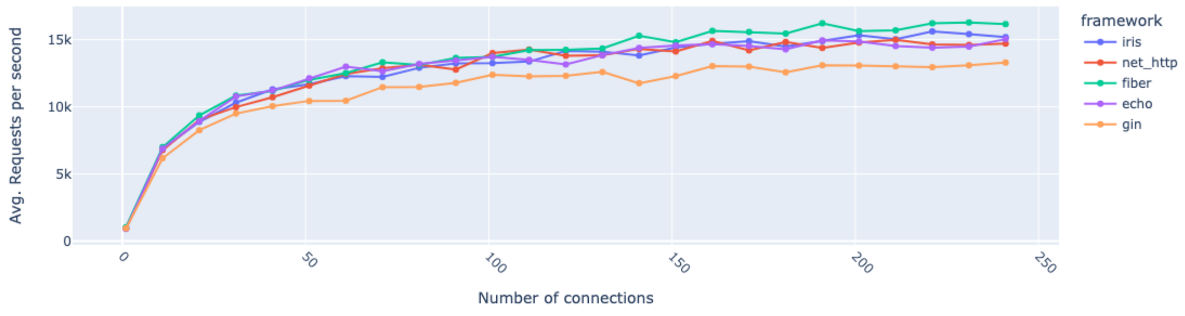

Figure 3: Average Requests per Second for a number concurrent connections for each frameworks `/empty` endpoint.

5.3 Log parsing and evaluation

With the focus on performance and avoiding skewing the performance measurements, a parameter parsing from search query parameters was omitted.

Therefore, only the relation to a specific framework and the considered endpoint were measured without relating every request to a specific roundtrip. That would require the client to generate the request or roundtrip ID and the respective receiving HTTP server to implement the ID-parsing from the search query parameters, as mentioned above.

[31]`https://github.com/grafana/loki`
[32]`https://github.com/codesenberg/bombardier`

5.4 Results

This section will cover the results of section 4's benchmarks mentioned above.

Firstly, we will look at the whole roundtrip for each endpoint for every framework, which was measured by the HTTP benchmarking tools. We follow up with the durations for each framework's HTTP handlers and its nested RPC client performance. Lastly, we will examine each function's performance on the RPC server.

5.4.1 HTTP Endpoints

Each server's HTTP endpoint was called 100,000 times, except for the image endpoint, which was called 1000 times to avoid throttling.

In sum, the benchmarking software produced 1505000 requests. The boxplot's outliers are omitted because they make the plot harder to read. See Figure 4.

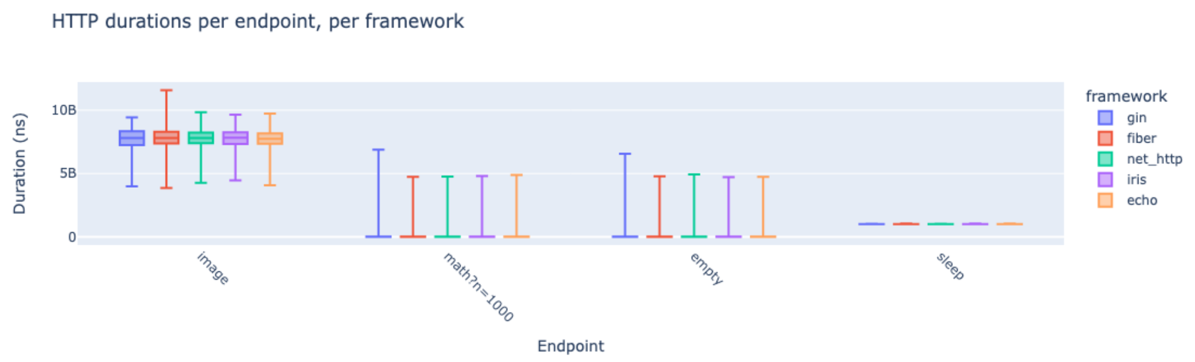

Figure 4: An overview of the complete roundtrip durations of the HTTP requests per framework per endpoint. $N = 1505000$. Each endpoint per framework was called 100000 times except `image`, which was called 1000 times.

5.4.2 Function durations on the RPC server side

Similar to the HTTP duration measurement in section 5.4.1, the plot in Figure 5 shows the distribution of function execution duration time on the RPC server side. Referencing Figure 1, this would correspond to step 3.

The number of function executions applies 1:1 to the ones from Figure 4.

5.4.3 RPC Client Durations

Figure 6 shows the durations for each RPC client's performance for each endpoint except the `/image` endpoint. The data for the image endpoint is incomplete and, therefore, omitted in this section.

In Figure 1, the RPC client covers the sending step 2 and receiving step 4.

5.4.4 Function Durations for each Handler

Each incoming function invocation request from the benchmarking server or user is saved inside this metric until the RPC server receives the function results.

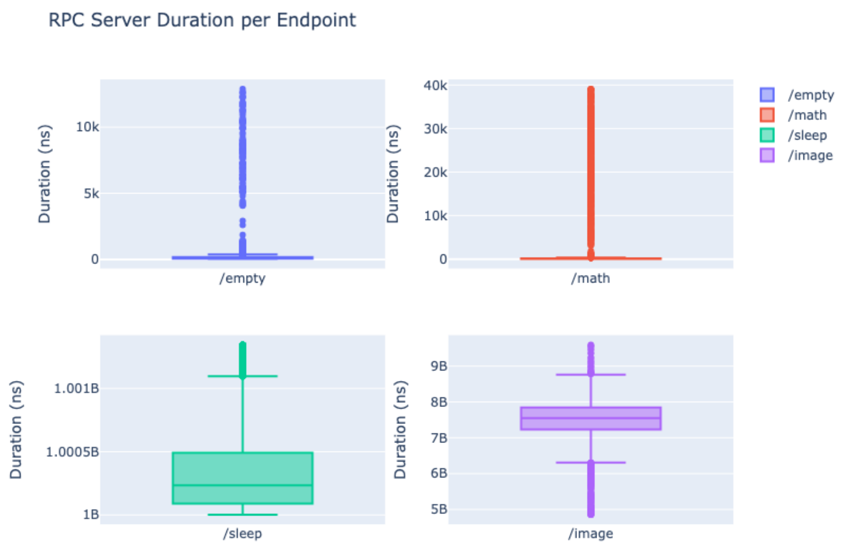

Figure 5: Function execution time after RPC invocation until completion. Each endpoint per framework has been called 10000 times, except `/image`, which was called 1000 times.

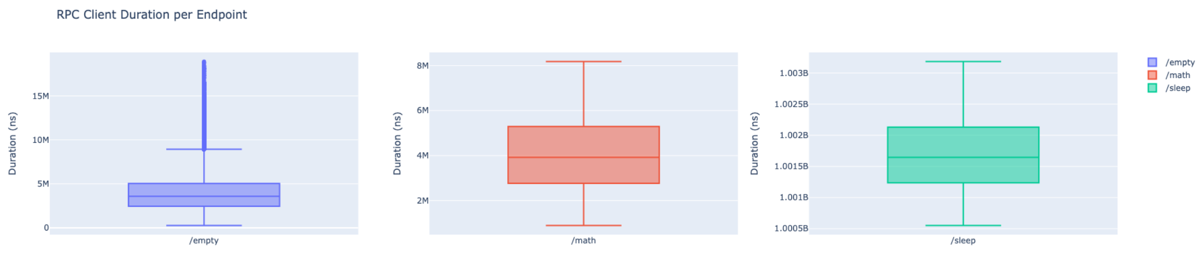

Figure 6: Corresponding to Figure 1 this duration covers the sending of step 2 and the receiving of step 4; framework independent.

Figure 7 displays the performance in terms of duration for each endpoint categorized by the underlying web framework.

In reference to Figure 1 this would correspond from receiving step 1 to sending step 5.

5.4.5 Response Deltas

Since we are interested in a possible bottleneck occurring between receiving step 4 and sending step 5, we group the requests by their corresponding invocation ID inside the HTTP to RPC route, map the nearest HTTP request invocation time to that request, and subtract the duration of the HTTP endpoint invocation duration from the HTTP duration.

Figure 8 displays the delta distribution respective to their corresponding framework and endpoint. The complete relation is found in Figure 9. Both plots are cleaned from outliers.

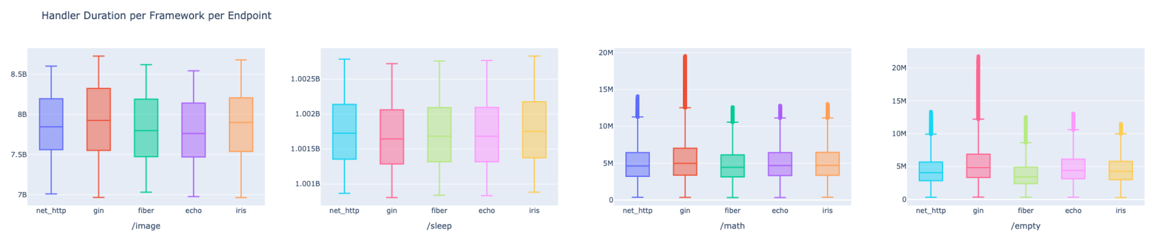

Figure 7: Each endpoints implementation in each framework's duration times.

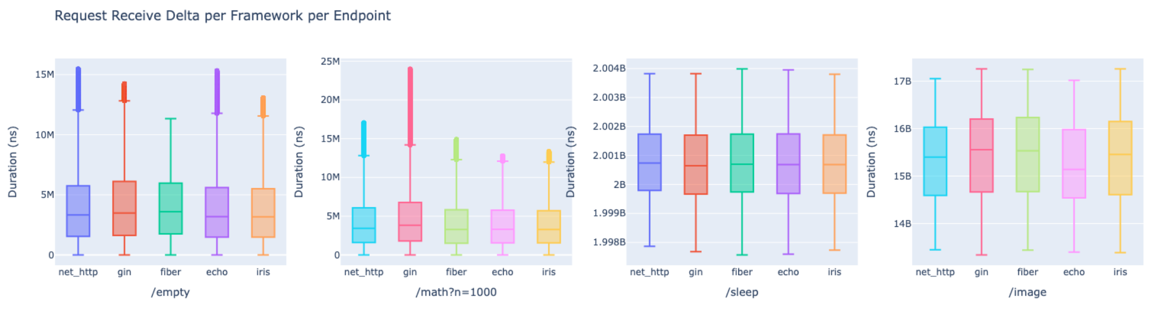

Figure 8: Delta distribution for each framework for each endpoint.

6 Discussion

Looking at Figure 9, it immediately becomes clear that the delta occupies at least one-third of the whole function roundtrip for small workloads. Even more interesting is that the image endpoint exceeds two-thirds of the delta time, confirming a bottleneck in this area of the testing infrastructure.

A possible explanation for this behavior could be that HTTP servers, especially those focusing on bandwidth efficiency, compress their information before sending it back to the client. Iris indirectly indicates this, allowing the developer to specify which compression algorithm to use.[33]

Additionally, some requests may include JSON parsing, which can also add to the overhead.

The web framework data shows minimal differences between each framework's performance and its endpoint implementations. Gin seems to perform the worst in terms of speed and consistency. Compared to the other frameworks regarding Fiber, a performance boost did not notably occur.

[33] https://www.iris-go.com/docs/#/?id=api-examples

7 Conclusion

For this report, a minimal test infrastructure was created and tested with a selection of the most popular Go web frameworks acting as a load balancer in a quasi-FaaS application. The benchmarking confirmed a bottleneck between receiving a function result and sending it back in an HTTP response to the requester. The data indicates that this insight applies across multiple frameworks and endpoints, independent of their work and information size.

7.1 Future work

It would be interesting to see if the occurrence of the bottleneck is a Go-specific phenomenon or a language-independent issue in that use case.

Also interesting would be a non-HTTP or TCP-based load-balancing act, which handles uncompressed information throughput to confirm this problem and possibly mitigate this bottleneck.

References

[23a] "Fission/Fission: Fast and Simple Serverless Functions for Kubernetes". In: (2023). URL: https://github.com/fission/fission (visited on 05/31/2023).

[23b] "Knative Documentation". In: (May 2023). URL: https://github.com/knative/docs (visited on 05/31/2023).

[23c] "Kubernetes - GitHub Repository". In: (May 2023). URL: https://github.com/kubernetes/kubernetes (visited on 05/31/2023).

[23d] "Nuclio - "Serverless" for Real-Time Events and Data Processing". In: (May 2023). URL: https://github.com/nuclio/nuclio (visited on 05/31/2023).

[23e] "Openfaas/Faas: OpenFaaS - Serverless Functions Made Simple". In: (2023). URL: https://github.com/openfaas/faas (visited on 05/31/2023).

[23f] "OpenWhisk". In: (May 2023). URL: https://github.com/apache/openwhisk (visited on 05/31/2023).

[Bro+23] Marc Brooker et al. *On-demand Container Loading in AWS Lambda*. 2023. arXiv: 2305.13162 [cs.DC].

[Clo24] Cloudflare. "Why use serverless computing?" In: (2024). URL: https://www.cloudflare.com/learning/serverless/why-use-serverless/ (visited on 04/04/2024).

[DKK22] Jonathan Decker, Piotr Kasprzak, and Julian Martin Kunkel. "Performance Evaluation of Open-Source Serverless Platforms for Kubernetes". In: *Algorithms* 15.7 (2022). ISSN: 1999-4893. DOI: 10.3390/a15070234. URL: https://www.mdpi.com/1999-4893/15/7/234.

[Inc24] Amazon Inc. "AWS Lambda Features". In: (2024). URL: https://aws.amazon.com/lambda/features/ (visited on 04/04/2024).

[JC18] David Jackson and Gary Clynch. "An Investigation of the Impact of Language Runtime on the Performance and Cost of Serverless Functions". In: *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*. 2018, pp. 154–160. DOI: 10.1109/UCC-Companion.2018.00050.

[RPT22] Rohan Basu Roy, Tirthak Patel, and Devesh Tiwari. "IceBreaker: Warming Serverless Functions Better with Heterogeneity". In: *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '22. Lausanne, Switzerland: Association for Computing Machinery, 2022, pp. 753–767. ISBN: 9781450392051. DOI: 10.1145/3503222.3507750. URL: https://doi.org/10.1145/3503222.3507750.

[Sch+22] David Schall et al. "Lukewarm Serverless Functions: Characterization and Optimization". In: *Proceedings of the 49th Annual International Symposium on Computer Architecture*. ISCA '22. New York, New York: Association for Computing Machinery, 2022, pp. 757–770. ISBN: 9781450386104. DOI: 10.1145/3470496.3527390. URL: https://doi.org/10.1145/3470496.3527390.

A Code samples

```
...
├─ ...
├─ cmd
│  └─ main.go
└─ pkg
   ├─ ...
   ├─ fns
   │  ├─ srpc
   │  │  ├─ rpc_client.go
   │  │  └─ rpc_server.go
   │  ├─ empty.go
   │  ├─ math.go
   │  ├─ image_transform.go
   │  ├─ sleeper.go
   │  └─ ...
   └─ webserver
      ├─ echo.go
      ├─ fiber.go
      ├─ gin.go
      ├─ iris.go
      ├─ net_http.go
      └─ ...
```

Listing 1: Core project structure

```go
package fns
// ...
func estimatePi(n int64) float64 {
    // ...
    // Code for approximating pi
    // ...
    return pi
}

func Math(points int64) (RPCResponse, error) {
    return RPCResponse{FN: fmt.Sprintf("%.5f", estimatePi(points))}, nil
}
```

Listing 2: Implementation of the `/math` function located in `pkg/fns/math.go`. This code would reside inside the RPC server.

```
...
├─ ansible
│  ├─ playbooks
│  │  └─ ...
│  ├─ README.md
│  └─ inventory.ansible.yml
└─ ...
```

Listing 3: Ansible source structure

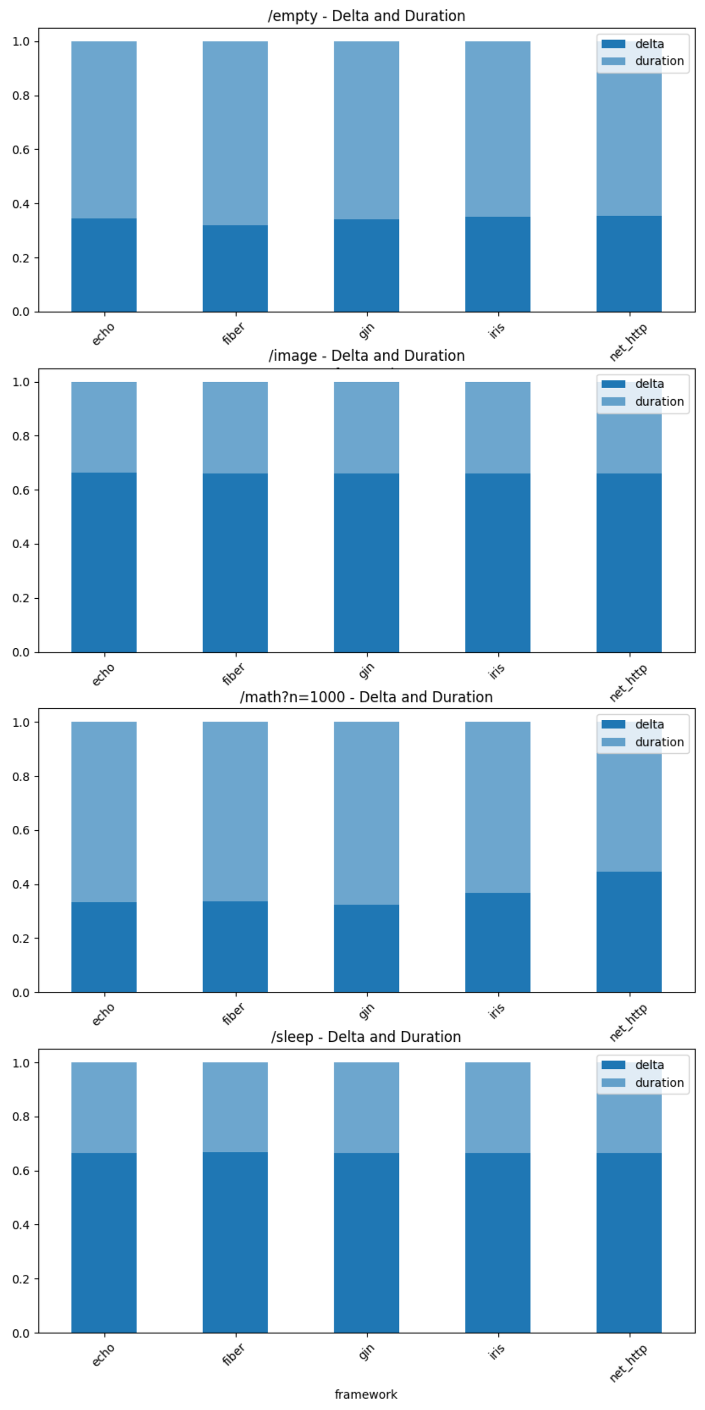

Figure 9: The complete roundtrip duration for each endpoint on each framework. It displays the bottleneck position on a relative scale.