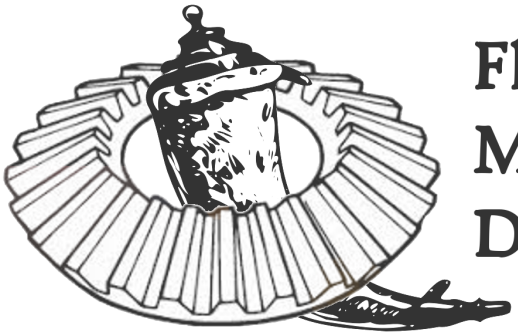
university of groningen

faculty of science and engineering

IMPROVING THE PERFORMANCE OF A PERFORMANCE MONITOR. THE CASE OF THE FLASK MONITORING DASHBOARD.

BOGDAN PETRE



Flask-
Monitoring
Dashboard

SUPERVISORS:
dr. V. Andrikopoulos
dr. M.F. Lungu

Groningen, July 2019

SUPERVISORS:
dr. V. Andrikopoulos
dr. M.F. Lungu

LOCATION:
Groningen

TIME:
July 2019

ABSTRACT

Application Performance Management (APM) solutions have seen widespread adoption in recent years, but their impact on the monitored application has not been thoroughly researched. This leaves developers unaware of potential slowdowns in their application caused by their APM of choice. Furthermore, it causes APM solutions to only focus on features and price, ignoring their impact on performance.

One APM tool that has enjoyed recent success with the open source community is the Flask Monitoring Dashboard (FMD), which tracks the performance of Flask web services. While it increased in features and popularity, its impact on the performance of monitored applications in terms of overhead was never studied.

We introduce a benchmarking framework, consisting of both micro- and macro-benchmarks, to measure the overhead of the FMD. We then perform an iterative performance improvement process of the tool, using the initial results of the benchmarks as reference. Our final results reveal a significant reduction of overhead across all monitoring levels supported by the FMD. This research shows how a combination of micro- and macro-benchmarking can guide other APM solutions towards improving their performance with respect to induced deterioration to the monitored system.

ACKNOWLEDGMENTS

I would like to thank my two supervisors, Vasilios Andrikopoulos and Mircea Lungu, for their help in conducting this research. Their guidance and constant feedback proved invaluable for the success of this thesis.

Secondly, I would like to thank Patrick Vogel for starting the Flask Monitoring Dashboard project and for introducing me to it one year ago.

Finally, I would like to thank my family and friends for their unwavering support, not only during the writing of this thesis, but throughout the two years of my master's degree. I couldn't have done it without you.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

FMD Flask Monitoring Dashboard

QoS Quality of Service

APM Application Performance Management

1 INTRODUCTION

Web applications have become an integral part of everyday life, being used by businesses, individuals, and governments alike. Be it in commerce, public administration, banking, education, or media, we are certain to encounter them in one form or another.

This widespread adoption of web applications, their increasing complexity, and ever larger user base[1], have given rise to new challenges facing application developers and system administrators. Non-functional requirements such as security, maintainability, and availability had to be reconsidered in order to stay relevant in the context of large scale web applications.

One quality attribute in particular has a direct and immediate impact on the success of web applications: performance. One study into the amount of time that web users are willing to wait for the retrieval of information showed this value to be only 2 seconds [1]. Furthermore, Google found that a delay of half a second in the response time resulted in a 20% drop in user traffic[2]. More recently, Amazon calculated that a delay in page load time of one second would result in a loss of $1.6 billion yearly revenue[3].

The above results should come as no surprise. After all, the driving force behind the growth of the Internet was rapid access to information. They do, however, show why performance is a key driver to modern web applications. A slow application leads to a loss of users, which in turn leads to a loss of revenue. In other words, *time is money!*

1.1 FLASK MONITORING DASHBOARD

Acknowledging the need of developers to get insights into the performance of their applications, a wide variety of software products, collectively known as Application Performance Management (APM) tools, have become available. Ranging from commercial (e.g. New Relic [2], Dynatrace [3], AppDynamics [4]) to open source (e.g. Kieker[5], Pinpoint [6]), they collect runtime information about the monitored system, which can then be used by developers to identify and correct eventual performance anomalies.

One such APM solution that has enjoyed recent success with the open source community is the Flask Monitoring Dashboard (FMD) [7].

1 The total number of Internet users has increased worldwide from 1.5 billion to 3.9 billion between 2008 and 2018 https://www.statista.com/statistics/273018/number-of-internet-users-worldwide/

2 http://glinden.blogspot.com/2006/11/marissa-mayer-at-web-20.html

3 https://www.fastcompany.com/1825005/how-one-second-could-cost-amazon-16-billion-sales

FMD is a Python package that specializes in monitoring the performance of web applications built with Flask [4], a popular Python web framework. At the time of writing, the project has received over 150 Github stars, 45 forks, and 50 thousand PyPi downloads [8], attesting to its popularity among Flask developers.

Initially, the FMD only measured the response times of endpoints and recorded when they were accessed and provided different visualizations based on this data. It then expanded its functionality, offering outlier detection and request profiling. However, as the number of features increased, so did the complexity and the resources required to use it. Given that we were deeply embedded in the development and maintaining of the FMD, we witnessed first-hand the growing number of issues related to worsening performance.

Additionally, we had the experience of deploying the FMD for Zeeguu[5], a Flask application in the field of education, which accelerates the learning process of a foreign language [9]. Using this concrete use case, we learned of some situations where the performance of Zeeguu was negatively affected.

1.2 RESEARCH QUESTION

A slight decrease in the performance of a system is to be expected when a monitoring tool is used. This is caused by what is known as the *probe effect*[6]:

> **probe effect** - an effect caused by the measuring instrument on the component or system being measured, such as a performance testing tool or a monitor.

Firstly, the performance drop can be attributed to the fact that the monitored system shares the same resources with the monitoring tool. Secondly, the APM solution often uses instrumentation techniques to gather the desired data, introducing further delays in the system.

This difference in performance caused by adding a monitoring tool to the system is called *overhead*. Ideally, the overhead should be as low as possible. However, when researching different APM solutions, we found there is little to no mention of the overhead. Indeed, the main selling points of various monitoring tools are limited to features, language support, ease of use, and pricing (in the case of commercial ones). Their own performance, i.e. their overhead on the monitored system, remains largely undiscussed. As we have previously seen in the case of the FMD, this approach can lead to serious performance issues for the monitored applications. This is all the more detrimental,

4 http://flask.pocoo.org/
5 https://www.zeeguu.org/
6 https://www.globetesting.com/en/probe-effect/

as those applications used APM solutions to detect and prevent such issues in the first place.

With this motivation in mind, the topic of this thesis can be summarized through the following research questions:

Q1: *What is the performance of the Flask Monitoring Dashboard?*

Q2: *How can the performance of the Flask Monitoring Dashboard be improved?*

1.3 APPROACH AND OVERVIEW

In order to successfully answer the research questions, it is essential to have a clear picture of the FMD, in terms of features, architecture, and implementation. This is the topic of Chapter 2.

Chapter 3 focuses on answering the first research question. We introduce a benchmarking framework designed to measure the overhead of the FMD. We then discuss the results of the benchmarks and identify several performance issues that can be addressed.

Our approach to answering the second question follows the five principles of a Canonical Action Research [10]:

1. The Principle of the Researcher-Client Agreement. For the purpose of this work, the thesis supervisors acted as the clients, while the author acted as the researcher.

2. The Principle of Cyclical Process Model. Each cycle contains the following five steps: diagnosis, action planning, intervention, evaluation, and reflection.

3. The Principle of Theory. It states that the actions of the researchers must be guided by theory.

4. The Principle of Change through Action. After the diagnosis step, action must be taken to improve the current state.

5. The Principle of Learning through Reflection. The process should lead to advancing the knowledge in the field.

We use the benchmarking framework introduced in Chapter 3 as a means to achieve both the diagnosis and evaluation steps of the cyclical process model. We discuss our three cycles, or iterations, in Chapters 4, 5, and 6. We tackle different FMD performance issues in each of these iterations.

Finally, in Chapter 7, we summarize the conclusions of the thesis and propose new research directions.

2 BACKGROUND

This chapter offers an overview of the FMD. We present the motivation behind it, its architecture, features, and the way they are implemented. We also discuss the user survey we conducted among the FMD user base.

2.1 MOTIVATION

The Flask Monitoring Dashboard originated from the desire to offer developers a better insight into the evolving performance of their Flask web services. Up until its launch, Python developers could only choose between using APM solutions that treated their web service as a black box, implementing their own monitoring solution, thus "reinventing the wheel", or not using any monitoring whatsoever [7].

The FMD addressed this need by offering an integrated solution for collecting and visualizing performance statistics, across both endpoints and versions of the applications.

2.2 ARCHITECTURE

A high level diagram of the FMD architecture is shown in Figure 2.1. The bottom half of the diagram shows a typical 3-tier web application, with the database and user interface, although these components are not mandatory. The FMD, seen in the top half, is attached to this application as a Pyhton package. From there, it collects different monitoring data, such as response times and timestamps of requests, and stores it into its own database (FMD DB). Users can then visualize the collected information through a web interface (FMD UI).

The FMD itself implements the Layers architectural pattern [11] and consists of three layers:

- *Data layer*: responsible for persisting the monitoring data.
- *Logic layer*: responsible for implementing the different features of the FMD, for accessing the Data layer, and for providing a REST API to be used by the Presentation Layer.
- *Presentation layer*: responsible for displaying the data to the user.

The Flask Monitoring Dashboard can be viewed as solving two different problems: data collection and data visualization. While the topic of this thesis covers only the performance of former, we will give an overview of both parts in the next two sections.

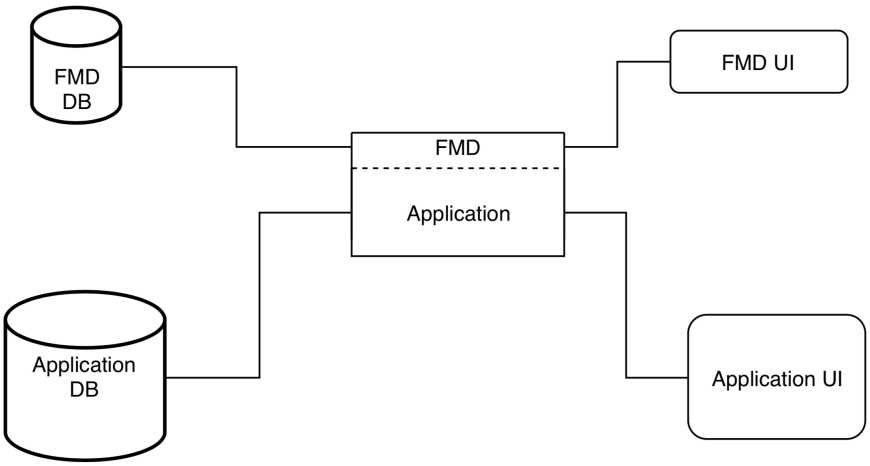


Figure 2.1: FMD Architecture

2.3 DATA COLLECTION

The Flask Monitoring Dashboard offers four main features: last requested, performance and utilization, profiler, and outlier detection. They are divided across four monitoring levels, from 0 to 3. These monitoring levels are *additive*, i.e. they contain all the features of the previous monitoring level, along with their own new features. For instance, Monitoring level 2 contains the last requested and performance utilization features, and adds the profiler.

Figure 2.2 shows the correspondence between monitoring levels and features. For every endpoint of the Flask application, the user can choose the monitoring level. The FMD then applies these monitoring levels for every endpoint by adding a wrapper to the the original

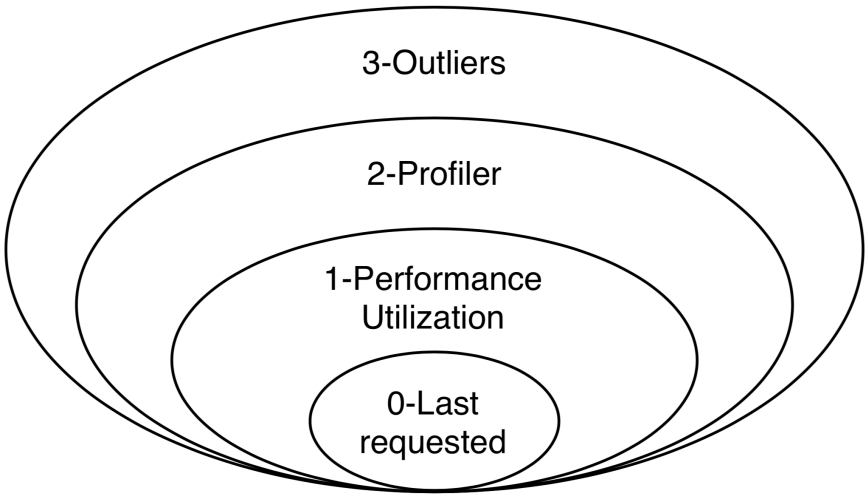


Figure 2.2: FMD Features

endpoint code, which implements the desired functionality. A more detailed description of each of the four features is given below.

2.3.1 *Last requested*

Last requested is the "lightest" feature of the FMD, both in terms of data collected and value to the user. For every endpoint, the FMD stores the timestamp when it was last accessed. This feature is included in all monitoring levels.

2.3.2 *Performance and Utilization*

Performance and Utilization can be seen as the core features of the FMD, and indeed of any Application Performance Management solution. For every request made to an endpoint with monitoring level 1, the FMD measures and stores its response time. Additionally, it collects information about the timestamp, version, and IP of the requester.

2.3.3 *Profiler*

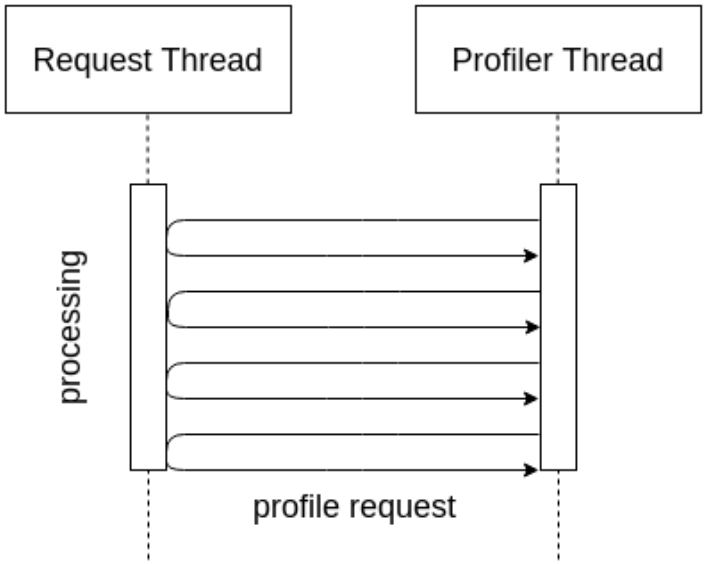


Figure 2.3: Profiler Thread

Monitoring level 2 introduces the most powerful functionality of the FMD: the statistical profiler[1]. This is implemented by launching an extra thread for every request of an endpoint set to monitoring level 2. This thread periodically samples the request thread and analyzes the current stack trace. By recording the stack trace of the request

1 Deterministic vs. statistical profiling https://docs.python.org/3/library/profile.html#what-is-deterministic-profiling

at different moments in time, the FMD infers the time spent in each function. This process is illustrated in Figure 2.3.

The author of this thesis implemented the profiler as part of his research internship, together with another student. As such, we know that the main motivation in choosing for a statistical, rather than deterministic profiler, was the lower overhead of the former. Our goal was not to obtain an exact reconstruction of the program execution code, but only to identify the *hot spots* in the code, i.e. operations that take a long time to execute, such as database or external API calls.

2.3.4 *Outlier detection*

The last feature of the FMD implements the functionality of detecting and collecting additional information about outliers. An outlier is simply a request that takes much longer than usual to execute, as seen in Equation 2.1:

$$response\_time_{outlier} > response\_time_{average} \times constant \tag{2.1}$$

where $response\_time_{average}$ is the average response time of all previous requests for the same endpoint, $response\_time_{outlier}$ is the response time of the outlier, and $constant$ is a user-defined outlier detection constant, with the default value of 2.5.

This is implemented by launching an additional thread for every request. This outlier detection thread is idle for an interval equal to $response\_time_{average} \times constant$, after which it checks the request thread. If the request thread is still running, it means that particular request is an outlier and extra information is collected and stored to the database: the current stack trace, the current physical resource utilization (CPU and memory), and the request environment, values, and headers.

2.4 DATA VISUALIZATION

Using the collected monitoring data, the FMD shows visualizations on two levels of abstraction:

- Application wide. Includes aggregated data of all endpoints. It offers visualizations such as an overview table (showing all the endpoints, their monitoring levels, mean response times and number of requests), per-hour and per-version API utilization heatmaps, stacked bar charts of daily API utilization, and box plots of API performance.
- Endpoint specific. Includes similar visualizations to the ones above, but for individual endpoints. Additionally, it includes

the tree with the execution paths and corresponding execution times for profiled requests, and an outlier page showing the extra information collected for slow requests of that endpoint.

2.5 USER SURVEY

As previously stated in Chapter 1, the Flask Monitoring Dashboard has enjoyed growing popularity with the open source community. In order to better understand the needs of the FMD user base and, consequently, improve the project, we conducted a user survey, the results of which are included in Annex C.

Reaching out to the FMD users proved to be a non-trivial task. Being an open source project, anybody can use the FMD freely and we collect no information about the users. Therefore, we decided to contact the people who starred the project on Github via email and politely ask them to fill out a 15 minutes form. The survey consisted of 16 questions, both open and multiple choice, covering four sections: discovery and usage, features, deployment, and final thoughts.

In total, nine respondents completed our questionnaire. While we do acknowledge that this represents a relatively small base, we could still acquire some useful insights, such as new feature suggestions, the size of the monitored applications, or the database management system used.

More relevant to the topic of this research, several users also complained about performance, both in Questions 2 and 5. One user mentioned that he never used the FMD in production because "the performance is not good enough", while another wrote that the profiler "slows the app" and does not think he will be using it again. Along with previous performance issues reported on Github[2], this further reinforces the motivation behind the thesis: investigating and improving the FMD performance.

2 Github issues related to performance: https://github.com/flask-dashboard/Flask-MonitoringDashboard/issues/187, https://github.com/flask-dashboard/Flask-MonitoringDashboard/issues/200

3 INITIAL STATE

This chapter describes the first phase of the project, which consists of determining the initial state of the FMD, in terms of its performance. We first review other works dealing with performance measurement, then discuss our approach and its implementation, and finally the results of the measurements.

3.1 RELATED WORK

Measuring the performance of software components is a topic that has been extensively covered in research and industry alike. A common approach is the use of benchmarks. The Merriam-Webster dictionary[1] gives the following definition of a benchmark:

> **benchmark** - a standardized problem or test that serves as a basis for evaluation or comparison (as of computer system performance).

Below, we highlight several works on the topic of benchmarking, giving us insight into how to implement benchmarks, how to use them, and how to analyze and visualize the results.

Amza et al. [12] designed and implemented three benchmarks, one for each of three different classes of dynamic web applications, with varying characteristics: an online bookstore, an auction site, and an online bulletin board. They also created a client emulator, which generates the workload for the applications by making HTTP requests. They then use these benchmarks to identify the bottlenecks of the system (e.g. CPU, memory usage, disk usage). They do this by varying a number of parameters: ratio of read-write interactions, number of emulated clients, and the initial size of the database.

Georges et al. [13] discuss a statistically rigorous way of measuring the performance of Java applications. They argue that there is a lot of non-determinism in Java benchmarks, caused by e.g. JIT compilation[2], thread scheduling, and garbage collection, and therefore, the right way of dealing with it is by using statistical methods, in particular taking the mean and confidence intervals of multiple measurements. They then design an experiment in which they measure the execution time of several benchmarks and show how using less rigorous methods

1 https://www.merriam-webster.com/dictionary/benchmark
2 Just-In-Time compilation https://www.oracle.com/technetwork/articles/java/architect-evans-pt1-2266278.html

of analyzing the results (e.g. average, median, best etc.) may lead to incorrect conclusions. They also use violin plots to visualize the results because it shows the probability density of the distribution.

Ray et al. introduce Jackpine [14], a benchmark for the performance evaluation of Geographic Information System (GIS). Jackpine consists of two classes of benchmarks: micro and macro benchmarks. The former "test the basic topological relationships [e.g. equals, crosses, touches, etc.] and spatial analysis functions [e.g. length, area, etc.]", whereas the latter implement common usage scenarios consisting of a series of queries (e.g. closest street, risk area categories, lake properties, etc.). After implementing the benchmarks, they use them to measure the performance of several RDBMSs with geospatial support. Finally, they perform a comparison of average execution times between RDBMSs and between benchmarks.

Menascé [15] discusses best practices for benchmarking, load testing, and performance management in the context of web applications. He identifies several components any benchmark should have: workload specification, specification of metrics, and specification of the measurement procedure. He also discusses availability and end-to-end response time as Quality of Service (QoS) attributes.

Aderaldo et al. [16] propose a set of requirements for microservices applications in order for them to be adopted as benchmarking standards by the software engineering community. They list a total of twelve requirements, ranging from architectural (e.g. explicit topological view), to DevOps (e.g. easy access from a version control repository), and general (e.g. availability of the application in multiple programming languages). They go on to evaluate four benchmark candidates against the aforementioned requirements.

A particularly researched topic is performance measurement in virtualized environments. Cherkasova and Gardner [17] propose a framework for measuring the CPU overhead of I/O-intensive operations using the Xen Virtual Machine Monitor. Li et al. [18] compare the performance of three hypervisors using Hadoop[3] macrobenchmarks, observing significant performance differences between CPU-intensive and I/O-intensive tasks. They then confirm their findings using microbenchmarks. Expósito et al. [19] measure the overhead of network communication for HPC[4] applications deployed in the cloud.

The `performance` project [20] is an open source repository consisting of approximately 50 benchmarks bundled in the form of a Python package and it aims "to be an authoritative source of benchmarks for all Python implementations"[5].

3 https://hadoop.apache.org/
4 High Performance Computing
5 `performance` project documentation: https://pyperformance.readthedocs.io/

3.2 CONCEPT

Measuring the performance of FMD is equivalent to measuring the overhead of FMD on the monitored Flask application. To obtain the overhead, we need to measure the performance of the Flask application without FMD and compare it with the performance of the same Flask application, but with FMD attached. Furthermore, this comparison needs to be performed for each of the four monitoring levels of FMD and determine the overhead for each of them.

Before moving forward, we need to specify what we mean by "performance", i.e. to specify the metrics [15]. In the same way as FMD itself [7], we define performance as being the end-to-end response time of an endpoint. In other words, performance is measured as the time between the moment a request is sent to the server and the moment a reply is received.

Similar to Ray et al. [14] and Li et al. [18], we propose the use of micro and macro benchmarks to measure performance. The latter is meant to simulate a real world web application, whereas the former consists of individual endpoints designed to stress a type of physical resource, as it will be explained below. In both cases, the benchmarks have to be built as Flask applications, in order to run them with the FMD.

3.2.1 *Micro benchmarks*

There are four types of physical resources that a web application can use: CPU, memory, disk, and network.

It is expected that FMD, and indeed any application monitor, has a different influence on different endpoints, depending on the type of resource that endpoint uses. For instance, assuming the FMD uses a lot of CPU time, then its overhead would be larger for an endpoint that is CPU-intensive than for a disk-intensive endpoint.

Therefore, we need micro benchmarks that stress each of these resources. Then, comparing the different FMD overheads for different micro benchmark classes, we can draw conclusions as to what resource is used the most by FMD and direct the performance improvement efforts in that direction.

OBSERVATION: As discussed in Chapter 2, during the data collection phase, FMD does not send any data over the network. This means it has no network overhead. As a result, we can a priori drop this class of resources from our analysis.

3.2.2 *Macro benchmark*

While the micro benchmarks provide insight into the overhead of the FMD on endpoints using different physical resources, the macro benchmark is a way of aggregating this information by measuring the overhead of FMD on a typical web application. We can do this by simulating a normal user scenario and measuring the total response time of the application with and without the FMD. A macro benchmark has the advantage of taking our analysis away from the synthetic loads and closer to the real world, in which the FMD actually operates.

Another benefit is that the macro benchmark acts as a final validation of the changes made to the FMD. If, for instance, as a result of the performance improvements, the overhead of the FMD decreases for some benchmarks but increases for others, the results of the macro benchmark will dictate whether the changes should be kept or discarded.

3.2.3 *System Architecture*

With the micro and macro benchmarks in place, the next challenge becomes running them and collecting the results. This becomes a non-trivial task, as the number of parameters to vary between runs increases. Therefore, we need an extra component, called the CALLER, to streamline this process as much as possible.

The role of the CALLER is to automate all the tasks normally done by the user: setting up the environment, starting the benchmark applications, running the benchmarks, and recording the results. The user of the framework must be able to configure the parameters of the benchmarks through the CALLER, such as setting the number of runs, selecting which benchmarks to run, the monitoring levels, etc.

The final component we introduce is the VIEWER, which has to create visualizations of the results collected by the caller. These should give a clear picture of the overhead of FMD and allow the user to perform an exploratory analysis of the data.

Considering these two extra components, together with the ones previously, previously mentioned, the logical view of the system becomes the one in Figure 3.1.

3.3 IMPLEMENTATION

Next, we present the most important aspects of implementing the benchmarking framework, going through the logical components one by one. The code is also publicly available on Github [21].

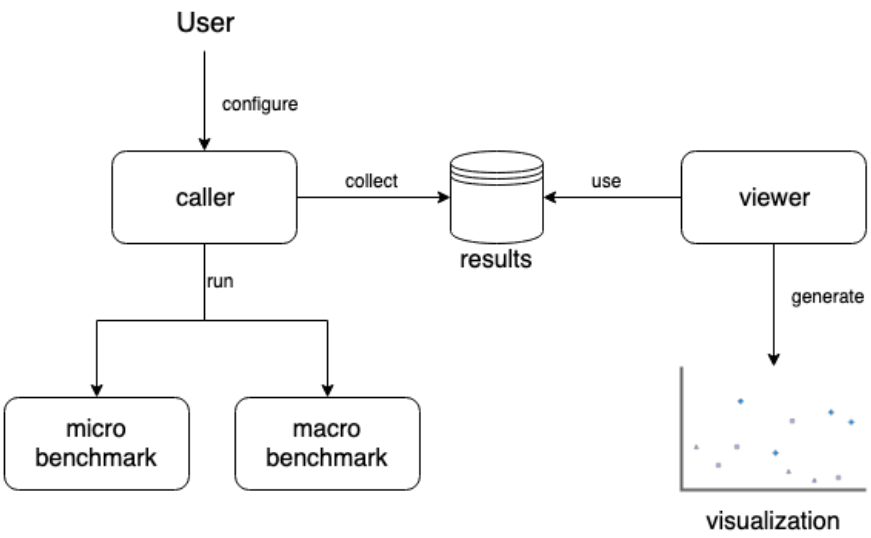


Figure 3.1: Logical view

Micro benchmarks

The micro benchmarks are presented in Table 3.1, together with a description and the type of physical resource used. Some of them are imported from the `performance` project [20] with slight modifications to the parameters (`pidigits`, `nbody`, `json_loads`, `sql_combined`, `sql_writes`, `sql_reads`), while the rest are Python implementations of the Fibonacci sequence (`fib`) and basic list operations (`list`).

Benchmark	Description	Resource type
`pidigits`	computes the first N digits of π	CPU
`nbody`	the N-body benchmark	CPU
fib	recursive Fibonacci sequence	memory
`list`	performs Python list operations	memory
`json_loads`	loads into memory N json object	memory
`sql_combined`	sql reads & writes using SQLAlchemy	disk
`sql_writes`	sql writes using SQLAlchemy	disk
`sql_reads`	sql reads using SQLAlchemy	disk

Table 3.1: Micro benchmarks

These micro benchmarks are then included in a minimal Flask web application, each of them being wrapped by an endpoint. To run a benchmark, one simply needs to make an HTTP request to its respective endpoint. The internal parameters of the benchmarks are tweaked (e.g. the number of π digits to compute, the value of N in the N-body benchmark, etc.) so that they all take approximately the same time to execute without the FMD, thus facilitating comparisons.

Macro benchmark

The macro benchmark is the Flask implementation[6] of the Conduit application, a Medium.com clone[7]. In choosing this application over many other open source Flask projects, we were guided by the requirements formulated by Aderaldo et al. [16]. Although their paper refers to microservice systems, we consider some of their requirements to be desirable for any web application aiming to be adopted as a benchmarking standard. Some of these requirements, which are fulfilled by our macro benchmark, are shown in Table 3.2.

Requirements	Assessment
Easy access from a version control repository	Available on Github
Support for continuous integration and automated testing	Yes, using CircleCI
Support for dependency management	Yes, using `pip`
Alternate versions	Multiple language and framework implementations available
Community usage & interest	The Github repository has over 25000 stars

Table 3.2: Macro benchmark requirements

Besides the actual Flask application, we also need a way to simulate users making requests to the system. For this purpose, we decided to use `Locust` [8], a load testing tool written in Python. The advantage over other load testing tools, such as `httperf`[9] or `Vegeta`[10], is that `Locust` is more geared towards simulating user behavior, rather than overloading the web server. This is achieved by using Python files to write test user scenarios and defining task sequences to be executed.

Our simulated users all have the following behavior: log in, read 3 articles, post one comment, read 2 articles, write one short article, read 2 articles, like one article, read 3 more articles. This scenario represents a mix of read and write interactions, similar to those used by Amza et al. [12]. The operations are performed immediately one after the other and the total time is recorded.

6 Flask Conduit implementation: https://github.com/gothinkster/flask-realworld-example-app
7 Conduit home page: https://github.com/gothinkster/realworld
8 Locust: https://github.com/locustio/locust
9 httperf: https://github.com/httperf/httperf
10 Vegeta: https://github.com/tsenart/vegeta

Caller

The caller is a Python program tasked with running the benchmarks. The algorithm used for running the micro benchmarks is shown in Listing 3.1.

```
1 parse configuration settings;
2 for every level in config_levels:
3     for every benchmark in config_benchmarks:
4         drop FMD tables;
5         start Flask application with FMD level;
6         sleep bm_cooldown;
7         call benchmark endpoint N times with perf;
8         store response times;
9         stop Flask application;
```

Listing 3.1: `caller` algorithm

In the case of the macro benchmark, the same algorithm is used, with the only exception being that at line 7, the `Locust` tool is used to simulate the users of the application by making HTTP requests.

The `config_levels` parameter represents the FMD monitoring levels to be tested. They can be any subset of $\{-1, 0, 1, 2, 3\}$, with -1 being no FMD, and $0, 1, 2, 3$ being the corresponding FMD levels. The `config_benchmarks` parameter can be any subset of the micro benchmarks in Table 3.1.

Before running every benchmark a number of times, the caller also clears the FMD database. This guarantees that the FMD database size does not influence the benchmarks. Then, the caller starts the Flask application with the desired monitoring level. To ensure that the server is fully up and running the caller also waits for `bm_cooldown` seconds before proceeding with running the benchmarks.

For the tasks of calling the endpoints and storing the response times, the caller uses the `perf` Python package [22], which is a "toolkit to write, run and analyze benchmarks"[11]. This was a major design decision with the purpose of increasing the reliability of the results. According to the documentation, the `perf` package lists the following advantages over the default `timeit`[12]:

> - It displays the average and the standard deviation [as opposed to the minimum for `timeit`].
> - It runs the benchmark in multiple processes [to account for the different number of dictionary hash collisions in each process because of the randomized has function and for ASLR[13]].

11 `perf` project documentation: https://perf.readthedocs.io/en/latest/
12 `timeit` package documentation: https://docs.python.org/3/library/timeit.html
13 Address Space Layout Randomization: https://searchsecurity.techtarget.com/definition/address-space-layout-randomization-ASLR

- By default, it skips the first value in each process to warmup the benchmark.
- It does not disable the garbage collector [in order to replicate a production environment].

The caller uses the API provided by `perf` to run the benchmarks and to store the results as JSON files. Analysis of these JSON files and comparisons between results is also possible through the command line interface of `perf`.

Viewer

The viewer is the component tasked with presenting the benchmark results in a visual and interactive way, which leads to a better understanding of the data.

It is written in Python and uses the open source `Plotly` package[14] to build the visualizations. First, it parses the JSON files in which the results are stored, then it filters the data to only leave in the information relevant for analysis, and finally, it builds two types of visualizations: violin plots and line charts.

Violin plots[23] take the best features of box plots: showing the median, spread, and the asymmetry in the data, and add the density function of the distribution. This allows the user to identify non-Gaussian distributions and clusters in the data, improving the quality of the exploratory analysis. For each micro benchmark, the viewer creates a visualization containing a violin plot for every FMD level tested (including no FMD).

The viewer also generates line charts representing the overhead for every set of benchmark results run with different loads. Each point represents the difference between the mean value of one benchmark, with one load, for one FMD monitoring level, and the mean of that benchmark, with the same load, without the FMD. The goal of these charts is to visually represent the dependency, if any, between the FMD overhead and the base response time of an endpoint.

3.4 RESULTS

This section presents the results of the benchmarks and describes the measurement procedure through which they were obtained, which is an essential aspect of any benchmark [15].

3.4.1 *Measurement procedure*

All results were obtained after running the benchmarks on a dual core Intel i5 2.7 GHz, 8 GB of 1866MHz LPDDR3 RAM, 256GB SSD Mac-

14 Plotly project: https://github.com/plotly/plotly.py

book Pro 2015[15]. During the benchmarks runs, all other applications were closed, as well as the Wi-Fi and Bluetooth adapters, to minimize system jitter as much as possible. The Turbo Boost[16] feature of the Intel processor was also disabled, in order to further reduce system variation.

In the case of the micro benchmarks, using the `perf` package to call the endpoints, we specified 20 processes and 5 values. This means $20 * 5 = 100$ measurements. Additionally, `perf` uses one extra process in the beginning that calls the endpoint until the execution times become stable, and then, each of the 20 process does a warmup call before the 5 measured calls. The database used for FMD is SQLite[17] and the Flask development server was used to serve the application. The parameters of all the micro benchmarks were tuned so that the execution takes approximately the same without the FMD, thus facilitating comparisons.

In the case of the macro benchmark, we used the same `perf` package, but this time to execute the `Locust` file that would simulate users loading the web server. This time, we only used 5 processes and 5 values with the `perf` framework, meaning $5 * 5 = 25$ measurements, because we noticed that the relative variance was much smaller, as the mean benchmark execution time took at least one order of magnitude more than the micro benchmarks. The database used for both the Flask application and the FMD was again SQLite. The application database is approximately 63MB and contains 500 users, every user has 10 articles of 10KB each, 30 comments per user, 3 tags per article, every user likes 3 other users and 3 articles.

Another point worth mentioning is that the load generator (i.e. the `Locust` tool) and the system under test (i.e. the Flask application) run on the same machine, as opposed to the common practice of using separate machines (for instance in Amza et al. [12]). However, this is acceptable in our case, because we are not interested in the performance of a web server or a web application, but rather the difference in performance the FMD makes. This means that as long as everything else except for the FMD monitoring levels stays the same, our results are valid.

To maintain the integrity of the measurements, we used the same version of FMD, 2.1.1, as well as fixed versions for all external packages used, for all the benchmarks run in this chapter. All further improvements of the FMD will branch out from version 2.1.1, such that all code changes can be isolated and their performance impact accurately measured.

15 Machine full technical specifications: https://support.apple.com/kb/sp715?locale=en_US

16 Intel Turbo Boost: https://www.intel.com/content/www/us/en/architecture-and-technology/turbo-boost/turbo-boost-technology.html

17 SQLite: https://www.sqlite.org/index.html

3.4.2 *Micro benchmarks*

The average response times of the micro benchmarks are shown in Table 3.3. The percentage difference between each FMD level and the baseline (no FMD) is also shown.

	FMD level				
Benchmark	None	0	1	2	3
`pidigits`	103	108 (+5%)	108 (+5%)	121 (+18%)	123 (+20%)
`nbody`	99	104 (+5%)	105 (+6%)	118 (+19%)	120 (+21%)
`fib`	124	128 (+3%)	126 (+1%)	142 (+14%)	148 (+19%)
`list`	102	106 (+4%)	107 (+5%)	119 (+16%)	121 (+18%)
`json_loads`	99	103 (+4%)	104 (+5%)	117 (+19%)	120 (+21%)
`sql_combined`	105	111 (+6%)	115 (+9%)	420 (+301%)	436 (+316%)
`sql_writes`	94	101 (+8%)	105 (+12%)	335 (+257%)	368 (+292%)
`sql_reads`	102	103 (+1%)	105 (+3%)	223 (+118%)	221 (+116%)

Table 3.3: Results of the micro benchmarks in ms

We can immediately make serveral observations based on the table:

1. Levels 2 and 3 have a significantly higher overhead (14% - 316%) than levels 0 and 1 (1% - 12%). This means that running the profiler is extremely costly.

2. The profiler affects different benchmarks differently. In particular, the profiler overhead (levels 2 and 3) for disk micro benchmarks (118% - 316%) is far greater than for CPU benchmarks (18% - 21%) or memory benchmarks (14% - 21%).

3. The overhead for levels 0 and 1 is almost the same, ranging between 1% and 12%.

4. The overhead of levels 2 and 3 on the recursive `fib` benchmark is no greater than that on other non-recursive memory endpoints, even though the profiler analyzes the stack trace.

Looking at average response times of the benchmarks only tells half the story, as it gives no information about the median, spread, outliers or clusters and, as such, results might prove unreliable[13]. To get a better picture of the results, we need to also look at the distribution of the data. We can do this by analyzing the violin plots of the results, shown in Figure 3.2 (note that the y axis is different for each benchmark).

Examining the violin plots, we draw the following conclusions:

1. The profiler (levels 2 and 3) increases not only the average response time, but also the spread of the distribution.

2. The spread increases the most in the case of the disk benchmarks.

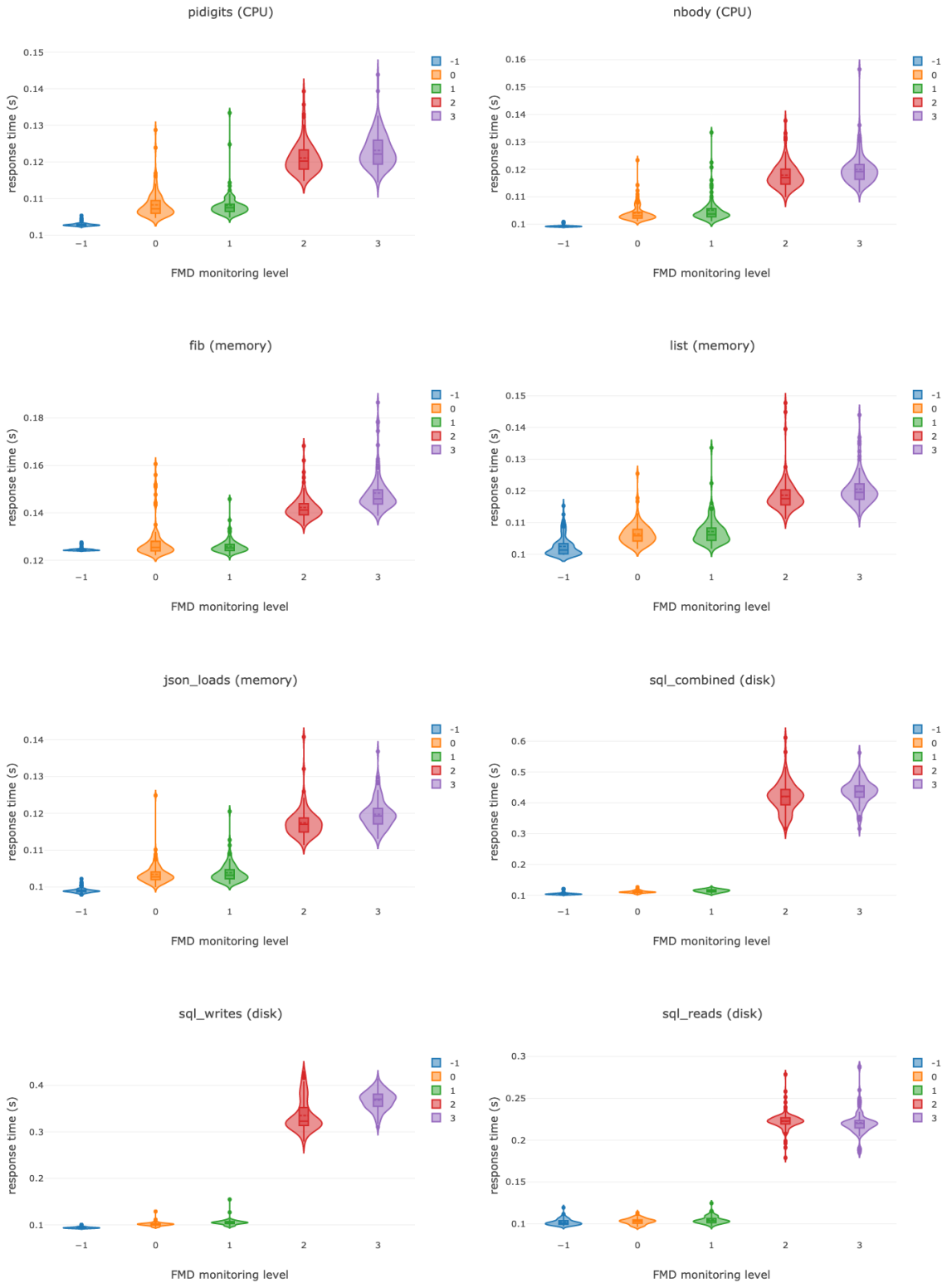


Figure 3.2: Violin plots of the micro benchmarks

3. The profiler increases both the mean and the variance of the response times significantly.

4. The benchmarks most affected by the profiler are those using disk resources: sql_combined, sql_writes, and sql_reads.

5. Monitoring levels 0 and 1 are very similar to each other in terms of overhead, and very different from levels 2 and 3, which add significant overhead.

6. In the case of the non-disk benchmarks, the spread of the distribution for monitoring levels 0 and 1 is significantly increased, due to the disk operations performed by the FMD.

Another interesting finding is that the large overhead and the large response time variance of FMD levels 2 and 3 for the `sql_combined`, `sql_writes`, and `sql_reads` benchmark is indeed caused by the disk operations, and not by using SQLAlchemy or SQLite. To prove this, we implemented:

A. Benchmarks that write and read directly to and from files on the disk. The results, seen in Figures A.1 and A.2, show again a large variance and many outliers.

B. A configuration setting allowing the FMD to use any RDBMS not just SQLite, as long as it is supported by SQLAlchemy. The results of the SQL benchmarks when using MySQL, shown in Figure A.3, indicate that MySQL actually adds more overhead and variance than SQLite.

3.4.3 *Varying the base response times of the endpoints*

So far, we have expressed the overhead as relative to the base duration of the benchmark, in terms of percentages. While this made it easier to compare the data from Table 3.3, this was only possible because all endpoints took approximately the same time to run without the FMD. This, of course, is not the case in the real world.

To drive this point across, consider the following scenario: an endpoint E with a parameter P takes 100 ms to execute without the FMD and 110 ms to execute with monitoring level K of FMD. Suppose that with a different parameter, P_2, E without FMD takes 200 ms to execute. How long will it take for endpoint E with parameter P_2 and monitoring level K to execute? Will it be 210 ms or 220 ms? Will the overhead be 10 ms, 10%, or something entirely different? In other words, is the overhead constant, or does it depend on the duration of the endpoint? Can we express the overhead as a function $f(x)$, where x is the base duration of the endpoint?

To answer these questions, we ran the same micro benchmarks from Table 3.1, with and without the FMD, but this time with different

loads. The parameters of the benchmarks were tweaked so that the base durations take 50 ms, 100 ms, 200 ms, 500 ms, and 1000 ms. The goal is to use the value of the mean overhead in these five discrete points to interpolate the overhead function $f(x)$.

The results of these measurements are shown in Figure 3.3. The X axis shows the mean response time of a specific endpoint E with different parameters P_1, P_2, P_3, P_4, P_5. The Y axis shows the absolute value of the overhead, computed as difference between the mean response time of E without the FMD and the mean response time of E with FMD at monitoring level K_i, $i \in \{0, 1, 2, 3\}$. The points corresponding to each monitoring level are then linearly interpolated. The vertical lines for each point represent the standard deviation of the response times for endpoint E, monitoring level K_i, and load L_j. Note that the Y axis differs for each chart.

It is important to mention that *all* measurements are affected by noise. The sources of this noise are numerous and, despite our best efforts, some remain beyond our control. These include, but are not limited to: the Python garbage collector and randomized hash function, the Address Space Layout Randomization (ASLR) of the operating system, CPU cache misses, background OS processes, and, most notably, disk I/O operations variance. The effect of the latter can be seen in the increased number of outliers for all FMD monitoring levels in Figure 3.2.

While the noise affecting a measurement is no greater than a few percentage points in most cases, measuring the overhead means subtracting such measurements, and thus amplifying the noise. This leads to many instances in which monitoring levels 3 and 1 have a higher overhead than monitoring levels 2 and 0 respectively, even though they do slightly more work. The same phenomena also explains why in some cases the overhead for monitoring levels 0 and 1 seems to be negative, which would imply that the endpoint performs better with the FMD, which of course is impossible.

A visual analysis of Figure 3.3 leads to the following observations:

1. The overhead of monitoring levels 0 and 1 is either approximately constant or grows slowly with the base duration.

2. The overhead of monitoring levels 2 and 3 clearly increases as the base duration increases.

3. The standard deviation tends to increase as the base duration increases, but the correlation is weaker.

Beyond the mere visual exploratory analysis of charts, we want to be able to quantify the above observations. An effective way of achieving this is to use regression. The purpose of regression is to "write the numeric output [in our case the overhead], called the dependent variable, as a function of the input [in our case the mean response

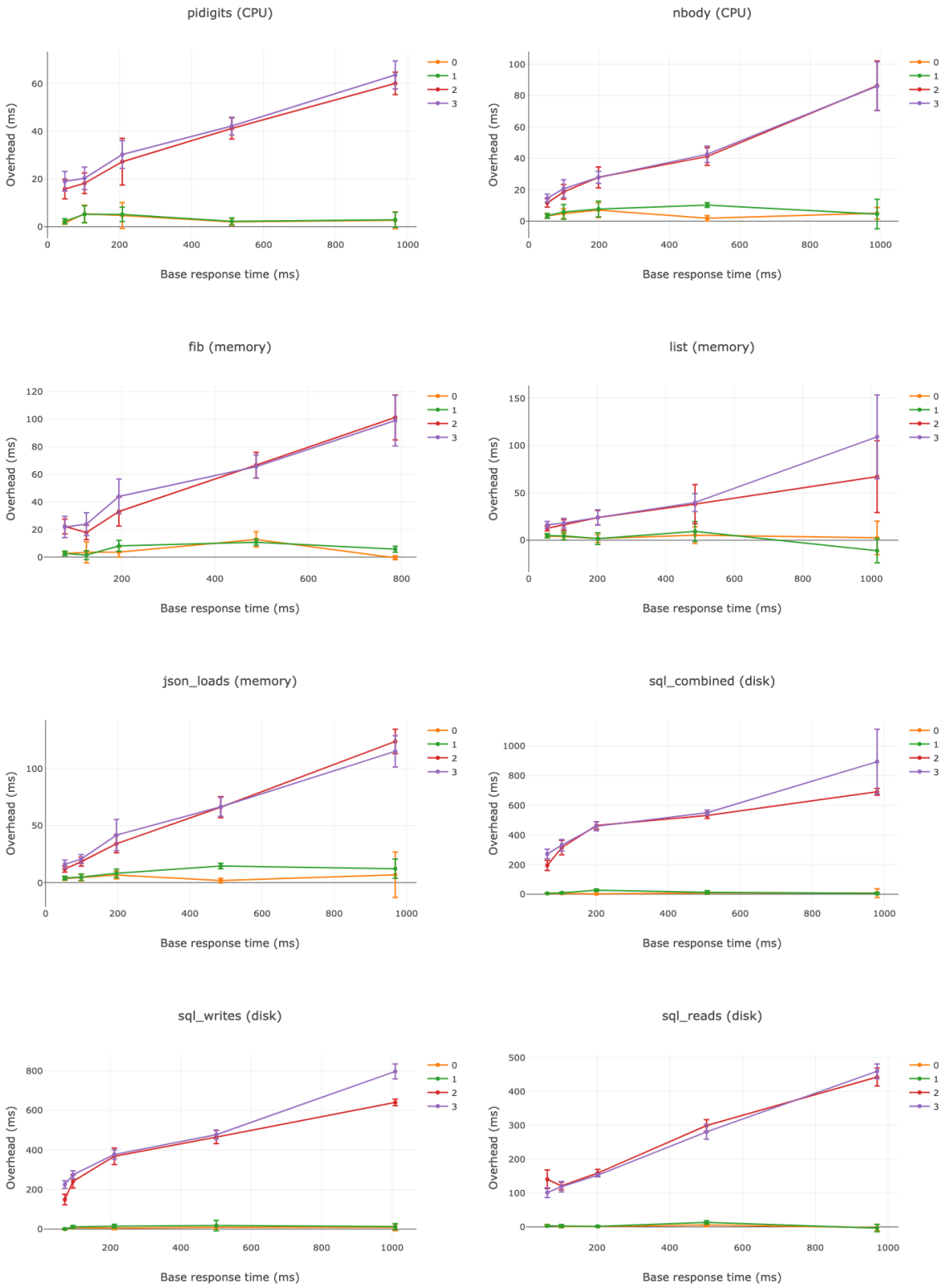


Figure 3.3: Overheads of the micro benchmarks

time without the FMD], called the independent variable" [24]. We have the following regression equation:

$$r = f(x) + \epsilon \tag{3.1}$$

where r is the observed overhead value, ϵ is the random noise, and $f(x)$ is the function we want to determine. At this point, for each FMD monitoring level, we should normally test multiple regression models for $f(x)$, fit them to the data, and pick the one that offers the best trade off between bias and variance.

In our case however, we can bypass this process and argue for a linear regression model for levels 2 and 3, and for an approximately constant model for levels 0 and 1:

$$f_{2,3}(x) = w_0 + w_1 x \qquad f_{0,1}(x) = w_0 \tag{3.2}$$

The first argument in favor of this choice is represented by the shapes of the line charts in Figure 3.3. The overhead of monitoring levels 2 and 3 increases with the base execution time in an approximately linear fashion. However, there seems to be no dependency between the overhead of monitoring levels 0 and 1 and the base duration.

The second argument stems from the inside knowledge of the way the FMD monitors requests. As described in Chapter 2, each monitoring level adds a wrapper to the request, the pseudo code of which can be seen in Listings 3.2, 3.3, 3.4, and 3.5.

```
-Main thread-
execute request
start thread T0----------thread T0-
return response         update endpoint info to db
```

Listing 3.2: Level 0 wrapper

```
-Main thread-
execute request
get duration
start thread T1----------thread T1-
return response         update request, endpoint info to db
```

Listing 3.3: Level 1 wrapper

Monitoring levels 0 and 1 do the same thing, regardless of the execution time of the request: start a new thread which updates a row in the `Endpoint` table, and, in the case of level 1, adds a new row in the `Request` table.

Monitoring levels 2 and 3 *do* depend on the execution time of the request. The longer the request lasts, the more loops the profiler (thread T2 in Listings 3.4 and 3.5) performs and the more profiling information it has to write to the database.

With these arguments in mind, we performed a linear regression with the least squares estimates methods, between the values of the

```
-Main thread-
start thread T2----------thread T2-
execute request         while not stopped:
get duration                profile request
stop thread T2----------update request, endpoint,
return response          profiler info to db
```

Listing 3.4: Level 2 wrapper

```
-Main thread-
start thread T3--------------------------------thread T3-
start thread T2----------thread T2-            get avg duration from db
execute request         while not stopped:     sleep for avg*constant
get duration              profile request      if outlier:
                                                  capture outlier info
stop thread T2----------stop thread T3---------if outlier:
return response         update endpoint,          update outlier info
                         request, profiler          to db
                         info to db
```

Listing 3.5: Level 3 wrapper

overhead and the base durations, for every endpoint, for every monitoring level. The results of the regression are shown in Table 3.4.

			Benchmark							
			pidigits	nbody	fib	list	json loads	sql combined	sql writes	sql reads
FMD level	0	w_0	3.7 ms	4.5 ms	4.5 ms	3.8 ms	3.9 ms	4.9 ms	3.8 ms	3.2 ms
		w_1	-0.001	0	0	0	0.002	0.004	0.006	-0.003
		R^2	0.09	0	0	0.04	0.11	0.35	0.39	0.20
	1	w_0	4.1 ms	6.2 ms	3.7 ms	7.2 ms	5.4 ms	15.1 ms	9.1 ms	4.7 ms
		w_1	-0.001	0	0.005	-0.014	0.009	-0.007	0.007	-0.003
		R^2	0.14	0	0.21	0.55	0.6	0.09	0.17	0.06
	2	w_0	14.8 ms	9.2 ms	8.9 ms	10.9 ms	7.3 ms	267 ms	197 ms	100 ms
		w_1	0.048	0.075	0.117	0.056	0.121	0.462	0.465	0.359
		R^2	0.99	0.98	0.99	0.99	0.99	0.85	0.91	0.98
	3	w_0	17.2 ms	11.4 ms	15.2 ms	5.7 ms	13.7 ms	266 ms	214 ms	76.7 ms
		w_1	0.048	0.072	0.107	0.096	0.106	0.632	0.572	0.396
		R^2	0.99	0.99	0.98	0.96	0.99	0.97	0.98	0.99

Table 3.4: Linear regression coefficients of the micro benchmarks

Coefficients w_0 and w_1 are those from equation 3.2, while R^2 is the coefficient of determination and is calculated as:

$$R^2 = 1 - E_{RSE} \tag{3.3}$$

where E_{RSE} is the relative square error of the model to the actual data points. R^2 ranges between 1, which means a perfect fit between the data and the model, and 0, which means a poor model.

Our model hypothesis is confirmed by the data in Table 3.4. All models for monitoring levels 2 and 3 have an R^2 score between 0.85 and 0.99 (with all but two being between 0.96 and 0.99), meaning that they provide good estimates for the underlying overhead function. This allows us to make reliable observations on the overhead for levels 2 and 3 based on the estimated models:

1. For the CPU benchmarks (the first two), the overhead is the lowest, with the slope w_1 ranging between 4.8% and 7.5%, and the intercept w_0 between 9.2 ms and 17.2 ms.

2. The memory benchmarks have a slightly higher overhead, although still very close to that of CPU, the slopes w_1 being around 11%, with intercepts w_0 between 9 ms and 15 ms.

3. The disk benchmarks suffer by far the worst overhead, with slopes w_1 between 36% and 63% and intercepts w_0 between 77 ms and 270 ms.

In the case of monitoring levels 0 and 1, the R^2 scores are significantly lower, meaning there is no linear dependency between the overhead and the execution time of the endpoint, just as we anticipated. In this case, it is more useful to use the means of the overhead [24] for every endpoint, across multiple base execution times, as seen in Table 3.5.

		Benchmark							
		`pidigits`	`nbody`	`fib`	`list`	`json loads`	`sql combined`	`sql writes`	`sql reads`
FMD	0	3.3 ms	4.4 ms	4.4 ms	3.6 ms	4.6 ms	6.5 ms	6.2 ms	2.0 ms
level	1	3.6 ms	6.4 ms	5.8 ms	2.0 ms	8.8 ms	12.5 ms	11.7 ms	3.4 ms

Table 3.5: Mean overhead values for monitoring levels 0 and 1

NOTE: In some cases, when running the benchmarks with monitoring level 3, the FMD threw the error seen in Annex B.1. The unpredictable character of this error indicates to a race condition as the likely culprit and fixing it will be part of the improvement process.

3.4.4 *Macro benchmark results*

In the same way we did for the micro benchmarks, we ran the macro benchmark without the FMD, as a baseline, and then with the four

FMD monitoring levels. This time, however, we varied the load by increasing the number of concurrent users making requests, rather than increasing the base duration of one benchmark, thus simulating a real world situation. The results are aggregated in Table 3.6. They show the mean response times, along with the percentage difference from the base response time. The violin plots of the results can be seen in Figure 3.4.

	FMD monitoring level				
Users	None	0	1	2	3
1	8.2 s	8.8 s (+7%)	8.8 s (+7%)	12.1 s (+48%)	12.0 s (+46%)
2	10.6 s	10.9 s (+3%)	11.7 s (+10%)	19.8 s (+86%)	20.5 s (+93%)
5	19.6 s	21.3 s (+9%)	21.4 s (+10%)	42.4 s (+117%)	45.9 s (+135%)
10	38.6 s	42.9 s (+11%)	40.9 s (+6%)	89.0 s (+130%)	89.6 s (+132%)

Table 3.6: Results of the macro benchmark

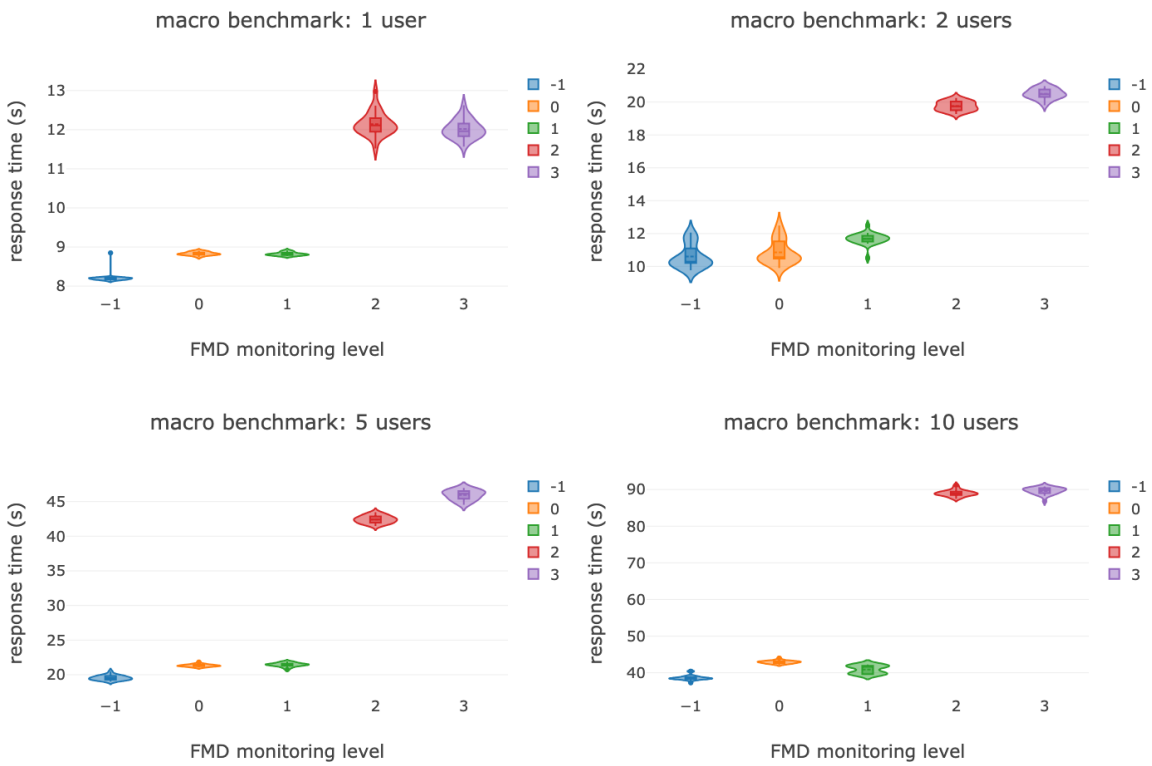


Figure 3.4: Violin plots of the macro benchmark with different loads

Looking only at the run with one user, the results are very much in line with what we were expecting to see after having analyzed the micro benchmarks before. We again see that monitoring levels 0 and 1 have a similarly low overhead, while that of monitoring levels 2 and

3 is significantly higher. The profiler increases the variance, while its relative overhead, of almost 50%, or 0.5, falls in the range of 0.36 - 0.63 of the proportional terms of the linear regression we performed for the disk-intensive micro benchmarks. This is unsurprising, considering that the macro benchmark consists of a series of endpoints that read and write from the database.

As we look at the response time when multiple users are simultaneously performing the same requests, we notice an increase in the overhead for the profiler, up to 135%. This can be attributed to the fact that for each user, there is always one thread constantly occupied with profiling the current request. For instance, in the case of 10 concurrent users, there are 10 concurrent profiler threads at any given time (assuming the time between consecutive requests to be negligible). As the number of threads exceeds the number of cores (which in the case of the benchmarking machine is 4), the performance degrades. Similar to the micro benchmarks, we modeled the dependency of the overhead to the base execution time of the benchmark with a linear regression model. The dependency can be visualized in Figure 3.5, while the regression coefficients are in Table 3.7. The table shows an overhead of 150% of the base response time for monitoring levels 2 and 3.

	FMD level			
	0	1	2	3
w_0	-0.78 s	0.5 s	-7.37 s	-6.45 s
w_1	0.131	0.049	1.504	1.518
R^2	0.98	0.83	0.99	0.99

Table 3.7: Linear regression coefficients of the macro benchmark

We see that for all monitoring levels, the overhead fits very well to a linear regression model, including for levels 0 and 1. At first, this seems to contradict one of the conclusions of the micro benchmarks results, that the overhead for monitoring levels 0 and 1 is independent of the base duration of the endpoint. However, in the case of the macro benchmark, the base duration increased not as a result of longer running endpoints, but of more endpoints. Given that monitoring levels 0 and 1 do the same operation for every request, regardless of its length, it is only normal that their overhead increases with the number of requests.

3.5 SUMMARY

In this chapter, we introduced our benchmarking framework, both in terms of concept and implementation, we discussed the motivation behind using each of the benchmarks, and presented and analyzed the results. The conclusions we drew in this chapter are relevant not

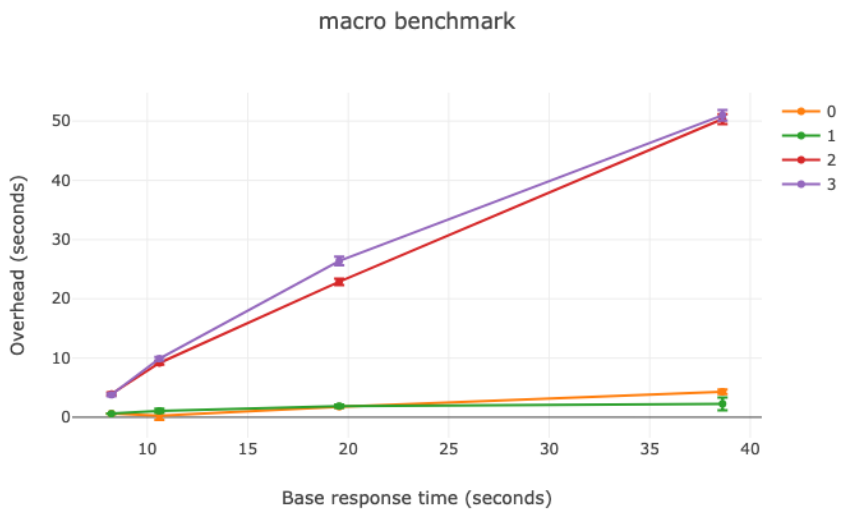


Figure 3.5: Overhead of the macro benchmark

only for the accurate description of the FMD overhead, but also for guiding our optimization efforts. Therefore, it is useful to reiterate some of the key takeaways regarding the performance overhead of the FMD, whcih represent future points of improvements (PoI):

PoI1 Monitoring levels 0 and 1 (no profiler) have similarly low overhead, while monitoring levels 2 and 3 (profiler) have similarly high overhead.

PoI2 Monitoring levels 0 and 1 have a fixed overhead, while monitoring levels 2 and 3 have an overhead proportional to the base duration of the endpoint.

PoI3 All monitoring levels increase the variance of the response times.

PoI4 In the case of concurrent requests, the benefits of using separate threads are diminished as the load increases.

PoI5 The overhead of all monitoring levels is lower for CPU and memory-intensive endpoints than for disk-intensive endpoints.

In the following chapter, which covers the performance improvements we made to the FMD, we will use the benchmark results of this chapter as a baseline for assessing the quality of the changes.

4 FIRST ITERATION

This chapter describes the first iteration of the FMD performance improvement process. We discuss the issues we decided to address, the changes we made, and their results.

4.1 RELATED WORK

While we already had some leads on what issues to tackle based on the results of Chapter 3, we also reviewed some of the existing literature discussing the performance of software projects.

Luo et al. [25] introduce a recommendation system, called "PerfImpact", which uses genetic algorithms to automatically detect combinations of input values that may cause performance regressions, and then mines the code changes likely to be responsible. Heger et al. [26] try to achieve the same goal, but their approach is to use performance unit tests and the revision history of software repositories. Sandoval et al. [27] argue that benchmarking the system every time a commit is made is unfeasible, so they propose a certain form of sampling, which they show is able to detect most performance regressions. Shang et al. [28] cluster multiple performance tests together and then use statistical tests to determine the ones to be used for discovering performance regressions.

While the above papers deal with automatically *detecting* performance issues (and do so in interesting and innovative ways), they do not discuss *solving* these issues, which is in fact what we are trying to achieve. Therefore, we have to look somewhere else for guidance throughout our iterative improvement process.

4.2 IMPROVEMENT DIRECTION

A good starting point for deciding on our course of action is represented, as mentioned above, by the results of Chapter 3, which are summarized by the five points of improvement (PoI) at the end of the chapter. In other circumstances, this would have been the only input of our iterative process.

However, given the open source nature of the Flask Monitoring Dashboard, we also have to consider the interests of the FMD community. There are two channels of communication that we use to learn about these interests:

- Github issues[1]. They are the main way for the users to report any problems they encounter with the FMD. Over time, more than 90 issues were opened.
- User survey (Annex C). This gives us insight into what features the users value most, what type of applications they use the FMD for, and what improvements they would like to see.

After analyzing these sources of input, we decided that the focus of our first iteration should be the FMD monitoring level 2, in particular replacing the profiler with the outlier detection feature. The reasoning behind this decision rests primarily on the fact that, as seen throughout all the benchmark results of Chapter 3, monitoring levels 2 and 3 have very similar, high, overhead. Since the only difference between level 2 and 3 is the outlier detection, it follows that this feature has, by itself, a comparatively low overhead.

However, in the current configuration, if a user wanted to use the outlier detection feature, she could only do it by choosing monitoring level 3, which also includes the profiler. In other words, instead of paying a small overhead cost for outlier detection, users are currently forced to pay the significantly larger overhead cost of the profiler. This might also be the reason why, according to Question 4 of the user survey, most users know about this feature, but choose not to use it.

A further reason in favor of replacing the profiler with the outlier detection feature as the first iteration is that we view this as a so-called "low hanging fruit". The implementation effort is low, as we don't add any new functionality, nor do we optimize existing features, but rather shift one feature for another in one monitoring level. On the other hand, we expect the benefits to be significant, which is to say we expect a noticeable decrease of the overhead for monitoring level 2.

4.3 IMPLEMENTATION

```
-Main thread-
start thread T2----------thread T2-
execute request         get avg duration from db
get duration            sleep for avg*constant
                        if outlier:
                            capture outlier info
stop thread T2----------update endpoint, request info to db
return response         if outlier:
                            update outlier info to db
```

Listing 4.1: New level 2 wrapper

1 FMD Github issues: https://github.com/flask-dashboard/Flask-MonitoringDashboard/issues

Performing this switch of functionality was relatively easy. Monitoring levels 0, 1, and 3 stayed the same, while for monitoring level 2 we replaced thread T2 (the outlier thread) of Listing 3.4 with thread T3 (the profiler thread) of Listing 3.5. Listing 4.1 shows the pseudo code of the new monitoring level 2.

The only change between the old and the new outlier threads is that now, the outlier thread in monitoring level 2 is also responsible for updating the endpoint and request information, whereas before, it was the task of the profiler thread.

4.4 RESULTS

We again ran the eight micro benchmarks we used in Chapter 3. As we left untouched all other monitoring levels, we only ran the benchmarks for monitoring level 2.

Figure 4.1 shows the new overhead (red), together with the old overhead (yellow), for monitoring level 2, and their dependency to the base duration of the endpoint. The first thing we notice is the significant decrease in the absolute values of the overhead, for all benchmarks. In all cases, the overhead is lower than 40 ms, and usually around 20 ms.

The second observation is that now, the overhead is no longer proportional to the base duration of the endpoint. This can be seen by looking at the shape of the overhead in Figure 4.1, but it also could have been predicted by analyzing Listing 4.1. The additional outlier detection thread (T2) performs a constant amount of work, regardless of the request duration. In this case, reiterating the argument from the previous chapter, it is more useful to compute the means of the overhead [24], shown in Table 4.1.

		Benchmark							
		`pidigits`	`nbody`	`fib`	`list`	`json loads`	`sql combined`	`sql writes`	`sql reads`
FMD level	2	12.1 ms	16.3 ms	19.2 ms	18.1 ms	19.1 ms	18.9 ms	16.0 ms	24.4 ms

Table 4.1: Mean overhead values for monitoring level 2

Even though the mean overhead for monitoring level 2, which ranges between 12 ms and 24 ms is larger than the mean overhead for monitoring levels 0 and 1, which is between 3 ms and 13 ms (Table 3.5), we can still place monitoring levels 0, 1, and 2 in the same category, in light of the overhead being independent of the endpoint base duration.

Given the significant performance improvements of monitoring level 2 for all micro benchmarks, we considered that no new insights could be gained by running the macro benchmark, so we opted against it.

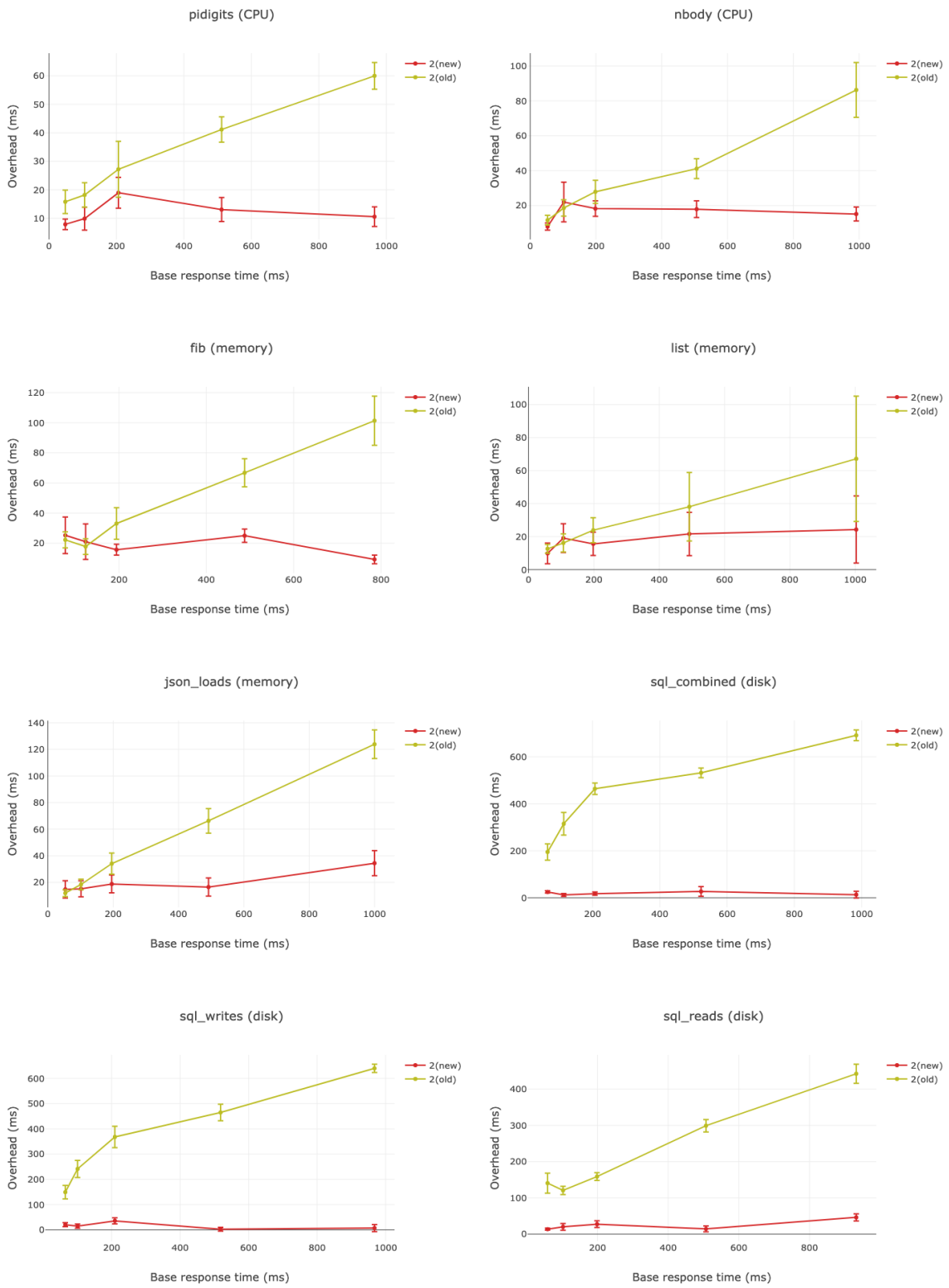


Figure 4.1: Overhead of monitoring level 2 before and after the first iteration

4.5 DATABASE SIZE

During the implementation of this iteration, we discovered another potential performance issue in the code of the outlier detection thread. In particular, line 3 of Listing 4.1: `get avg duration from db`. This line performs a query to the database, where it asks for the average duration of *all* previous requests for the given endpoint (the code is given in Listing B.2).

As no sliding window technique was used, our assumption was that this would cause a performance decrease as the number of requests for an endpoint increases. This performance drop would be missed by our benchmarks, as we ignore the influence of the FMD database size by always starting with an empty database before every benchmark run.

To test this hypothesis, we selected one micro benchmark, and we ran it with FMD databases containing different numbers of previous requests for the selected endpoint: 0, 1000, 5000, 10000, 50000, and 100000 requests. We opted for the `sql_combined` benchmark because it had the worst overall overhead in Chapter 3, and, as such, it was the most likely to show any performance regression.

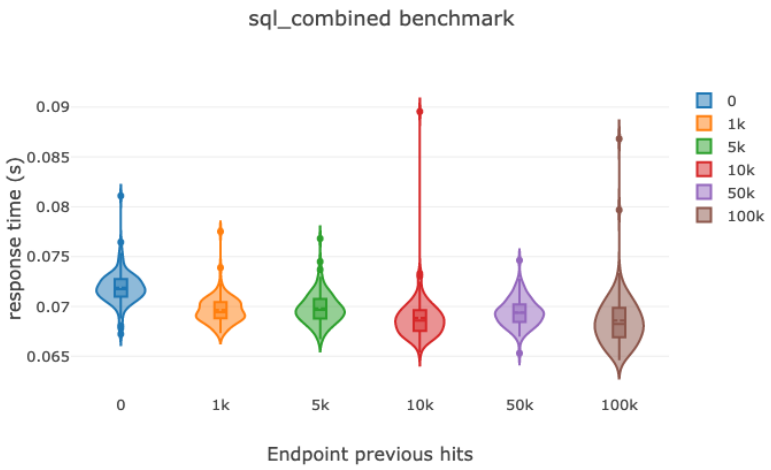


Figure 4.2: Influence of the number of endpoint hits on the performance of monitoring level 2

However, looking at the results in Figure 4.2, we see virtually no performance decrease as the number of previous endpoints increases. If anything, we see a slight performance *increase*, which could only be attributed to random measurement noise.

There are still two aspects that fall outside the scope of our test:

1. The average is computed by the RDBMS used. In our test, we used SQLite, but the results might differ for other RDBMSs.

2. A performance decrease might be visible after a certain number of previous requests, as we only stopped at 100 thousand.

Nevertheless, since this possible issue was never reported, we could not reproduce it ourselves, and our time is limited, we decided to leave this part of the code base in the current state and focus our development resources somewhere they can yield better returns.

5 SECOND ITERATION

The second step of our iterative performance improvement process is presented in this chapter. We discuss the concept, implementation, and results of our second intervention on the source code of the FMD.

5.1 CONCEPT

Despite the positive results obtained in the previous chapter, we could not help but wonder why the overhead for monitoring level 2 (the outlier), albeit no longer proportional to the base endpoint duration, is still noticeably larger than the overhead of monitoring levels 0 and 1. After all, the thread responsible for collecting the outliers is mostly idle (i.e. sleeps) and, after the request is processed, updates the endpoint and request information to the database, which is the same action performed by monitoring level 1. However, the mean overhead of monitoring level 2 ranges between 12 and 24 ms (Table 4.1), whereas the mean overhead of monitoring level 1 ranges between 2 and 12 ms (Table 3.5).

The reason for this difference can be found in line 3 of Listing 4.1, namely `get avg duration from db`. This performs a query to the database, retrieving the average duration of an endpoint, so that the outlier knows how long it has to wait idly.

Our idea is to remove this query by keeping the average duration of the endpoint in memory, in the form of a global dictionary. This would replace the costly disk access of the database query with a fast dictionary memory access. To do this, we also need the number of requests (or hits) for every endpoint, so that when a new request with duration D is processed, the new average can be recomputed using the formula:

$$new\_average = \frac{old\_average * hits + D}{hits + 1} \tag{5.1}$$

Lastly, we can also keep a copy of the `last_requested` information for each endpoint. This would in effect remove all database access operations from monitoring level 0, while maintaining the same functionality. The final structure of this global dictionary becomes that in Listing 5.1.

What we are in fact proposing is using an *in-memory cache*. Caching in itself is nothing new, being used whenever it is needed to store a frequently used resource in a location that is faster to access (e.g. CPU cache, disk cache, web cache, etc.). Our solution most closely

```
{
    Endpoint name: average duration,
                   number of hits,
                   last requested
}
```

Listing 5.1: Global dictionary structure

resembles the principles of in-memory databases [29], such as Redis[1] and VoltDB[2]. However, using one of these databases out of the box would have required a complete rethinking of the FMD architecture, which at the moment can only work with SQL databases, whereas the aforementioned ones are NoSQL.

Another design decision of our in-memory cache was not to include the request, outlier, and profiler data. This was done to prevent the memory cache to expand too much, potentially outside the bounds of the RAM memory of the machine. All the information we did include, namely `average duration`, `number of hits`, `last requested`, has a fixed size and only gets updated, rather than appended to.

5.2 IMPLEMENTATION

We implemented the cache as a global dictionary inside a python module called `cache`. The dictionary uses the names of the endpoints as keys. For convenience, we created the `EndpointInfo` class (Listing 5.2). This includes the values of the global dictionary, containing the `average duration`, `number of hits`, and `last requested` information. The class also encapsulates the logic of accessing its members, with the methods `set_last_requested`, `set_duration`, and `get_duration`. It is important to notice the use of mutexes to protect the access to shared resources. This is a requirement of the multi-threaded environment of web applications in which the FMD operates, where multiple concurrent requests might determine simultaneous reads and writes of the same global variable.

The `cache` module also includes two functions for synchronizing the data between itself and the database:

1. `init_cache()` initializes the cache by getting all the endpoint names and endpoint information from the database. It is executed when the Flask application, along with FMD, is started.

2. `flush_cache()` updates the database with the most recent information, which resides in the cache. It is executed when the Flask application, along with the FMD, shuts down.

1 Redis: https://redis.io/
2 VoltDB: https://www.voltdb.com/

```
class EndpointInfo(object):
    """
    Info about an endpoint that is stored in the memory cache.
    """
    def __init__(self, last_requested=None, average_duration=None
        , hits=None):
        self.last_requested = last_requested
        self.average_duration = average_duration if
            average_duration else 0
        self.hits = hits if hits else 0

    def set_last_requested(self, last_requested):
        with mutex:
            self.last_requested = last_requested

    def set_duration(self, duration):
        with mutex:
            self.average_duration = (self.average_duration * self
                .hits + duration)/float(self.hits + 1)
            self.hits += 1

    def get_duration(self):
        with mutex:
            return self.average_duration
```

Listing 5.2: `EndpointInfo` class

Using the cache, we proceeded to replacing the `last_requested` and `average_duration` database calls with cache writes and reads, for all four monitoring level wrappers, as seen in Listings 5.3, 5.4, 5.5, 5.6.

```
-Main thread-
execute request
start thread T0----------thread T0-
return response         update last\_requested info to cache
```

Listing 5.3: Level 0 wrapper, iteration 2

```
-Main thread-
execute request
get duration
start thread T1----------thread T1-
return response         update last\_requested, average info to cache
                        update request info to db
```

Listing 5.4: Level 1 wrapper, iteration 2

Using two versions of the same data, one in the database and one in memory, means we also have to consider the problem of eventual consistency [30]. By design, the cache holds the most recent copy of

```
-Main thread-
start thread T2----------thread T2-
execute request         get avg duration from cache
get duration            sleep for avg*constant
                        if outlier:
                            capture outlier info
stop thread T2----------update last_requested, average info to cache
return response         update request info to db
                        if outlier:
                            update outlier info to db
```

Listing 5.5: New level 2 wrapper, iteration 2

```
-Main thread-
start thread T3--------------------------------thread T3-
start thread T2----------thread T2-          get avg duration from cache
execute request         while not stopped:     sleep for avg*constant
get duration              profile request      if outlier:
                                                  capture outlier info
stop thread T2----------stop thread T3---------if outlier:
return response       update last_requested,         update outlier info
                         average info to cache   to db
                        update request,
                         profiler info to db
```

Listing 5.6: Level 3 wrapper, iteration 2

the data and that is the one used both by the FMD frontend (i.e. data visualization) and backend (i.e. data collection). While this guarantees the correctness of the data while the Flask application and FMD are running, problems might appear at shutdown.

Let us consider the case in which, for some reason, the cache is not flushed to the database and ask what will happen with the cache the next time the Flask application and FMD start. The `hits` value will still be correct, as this is computed by counting the entries in the `Request` table for every endpoint. In the same way, the `average_duration` value will also be correct. The only incorrect value will be that of `last_requested`.

Can this situation appear, in which the cache is not flushed to the database at shutdown? To answer this, we must first explain how the `flush_cache()` function is called. We use the Python `atexit` package[3] to register the `flush_cache()` function to be executed when the process containing the Flask application terminates. Under normal termination, this works as expected. However, under certain situations, discussed in the `atexit` documentation, the registered functions will not be called:

3 `atexit` package: https://docs.python.org/3.7/library/atexit.html

> Note: The functions registered via this module are not called when the program is killed by a signal not handled by Python, when a Python fatal internal error is detected, or when os._exit() is called.

For instance, if the Flask application is shut down using the SIGKILL signal, the cache is not flushed to the database. In the worst case scenario, this would mean losing the `last_requested` information of the endpoints since the last time the application was started. To mitigate this, we introduced an additional cache flush every time the user accesses the overview page of the FMD. This addition should have minimal impact on the usability of the FMD frontend, as the overview page already performs multiple, much heavier, queries to the FMD database in order to display the data. However, it manages to improve the worst case scenario, going from losing the `last_requested` information *since the last time the application was started* to losing it *since the last time the overview page was checked*.

5.3 RESULTS

To test the quality of our source code intervention in terms of performance enhancements, we ran the micro benchmarks. Given the insight we obtained of the fixed nature of the overhead for monitoring levels 0, 1 (in Chapter 3), and 2 (in Chapter 4), it made no sense to run the benchmarks for all possible base response times, as we would have only seen the same result, affected by different levels of noise. Therefore, we only ran them in the case of a base response time of approximately 100 ms.

The violin plots of the results are shown in Figure 5.1. We can see very similar results between the base case and the monitoring levels, for all benchmarks, both in terms of mean response times and variance.

		Benchmark							
		`pidigits`	`nbody`	`fib`	`list`	`json loads`	`sql combined`	`sql writes`	`sql reads`
FMD level	0 - old	3.3 ms	4.4 ms	4.4 ms	3.6 ms	4.6 ms	6.5 ms	6.2 ms	2.0 ms
	0 - new	0.6 ms	0.7 ms	0.2 ms	1.9 ms	-0.8 ms	-0.7 ms	0.0 ms	-0.3 ms
	1 - old	3.6 ms	6.4 ms	5.8 ms	2.0 ms	8.8 ms	12.5 ms	11.7 ms	3.4 ms
	1 - new	2.7 ms	3.5 ms	2.3 ms	-1.0 ms	4 ms	1.8 ms	2.1 ms	1.5 ms
	2 - old	12.1 ms	16.3 ms	19.2 ms	18.1 ms	19.1 ms	18.9 ms	16.0 ms	24.4 ms
	2 - new	4.6 ms	4.7 ms	4.4 ms	4.4 ms	2.2 ms	3.0 ms	2.4 ms	3.8 ms

Table 5.1: Mean overhead values for monitoring levels 0, 1, and 2

The improvement in performance becomes even clearer in Table 5.1. The *new* label refers to the mean overhead values after implementing

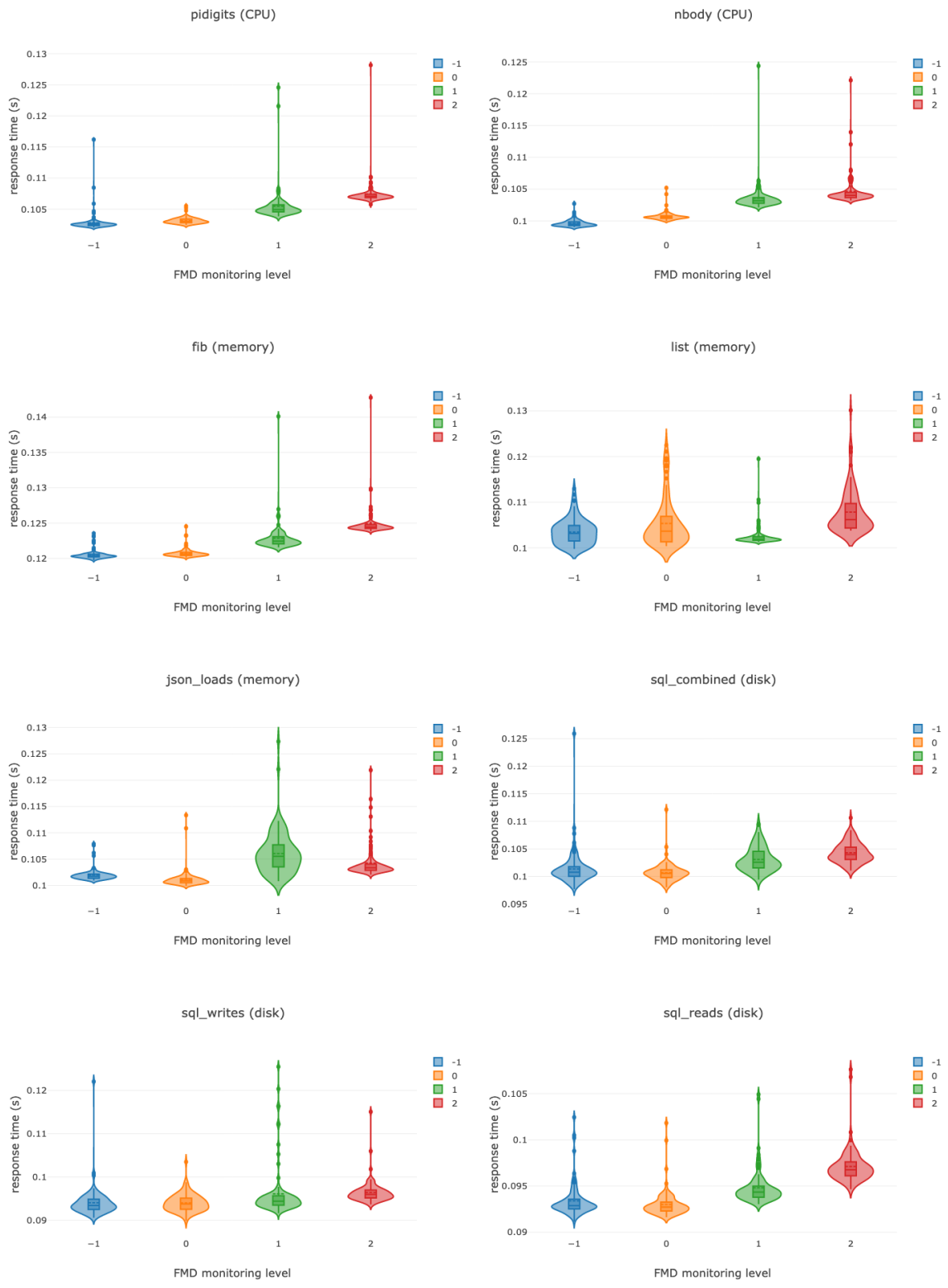


Figure 5.1: Violin plots of the micro benchmarks after the second iteration

the cache. The *old* label refers to the mean overhead values in the initial state for monitoring levels 0 and 1, and after iteration 1 for monitoring level 2. We can see improvements in the overheads of all monitoring levels, most noticeably level 2, which now has an overhead of less than 5 ms for all benchmarks, and level 0, for which the name now accurately describes the value of the overhead.

Although monitoring level 3 was not the target of this iteration, we expect to see a slight improvement there too, due to it containing the outlier functionality, which, as we saw, had an improved performance.

Table 5.2 shows the regression coefficients for monitoring level 3, after running the eight micro benchmarks with the five base response times. (50 ms, 100 ms, 200 ms, 500 ms, 1000 ms). The results are inconclusive, as in most cases one term (w_0 or w_1) decreases, while the other one increases.

			Benchmark							
			`pidigits`	`nbody`	`fib`	`list`	`json loads`	`sql combined`	`sql writes`	`sql reads`
FMD level	3 (old)	w_0	17.2 ms	11.4 ms	15.2 ms	5.7 ms	13.7 ms	266 ms	214 ms	76.7 ms
		w_1	0.048	0.072	0.107	0.096	0.106	0.632	0.572	0.396
		R^2	0.99	0.99	0.98	0.96	0.99	0.97	0.98	0.99
	3 (new)	w_0	16.9 ms	13.8 ms	23.8 ms	14.8 ms	14.6 ms	31.4 ms	22.9 ms	41.3 ms
		w_1	0.045	0.049	0.54	0.019	0.084	0.803	0.750	0.340
		R^2	0.95	0.93	0.94	0.82	0.90	0.95	0.95	0.95

Table 5.2: Linear regression coefficients of the micro benchmarks - level 3

Figure 5.2 shows the overhead of monitoring level 3 after the second iteration (purple), compared to the overhead in the initial state (yellow). While we do see a slight reduction of the overhead overall, we consider there is still room for improvement, which will be the objective of the third iteration.

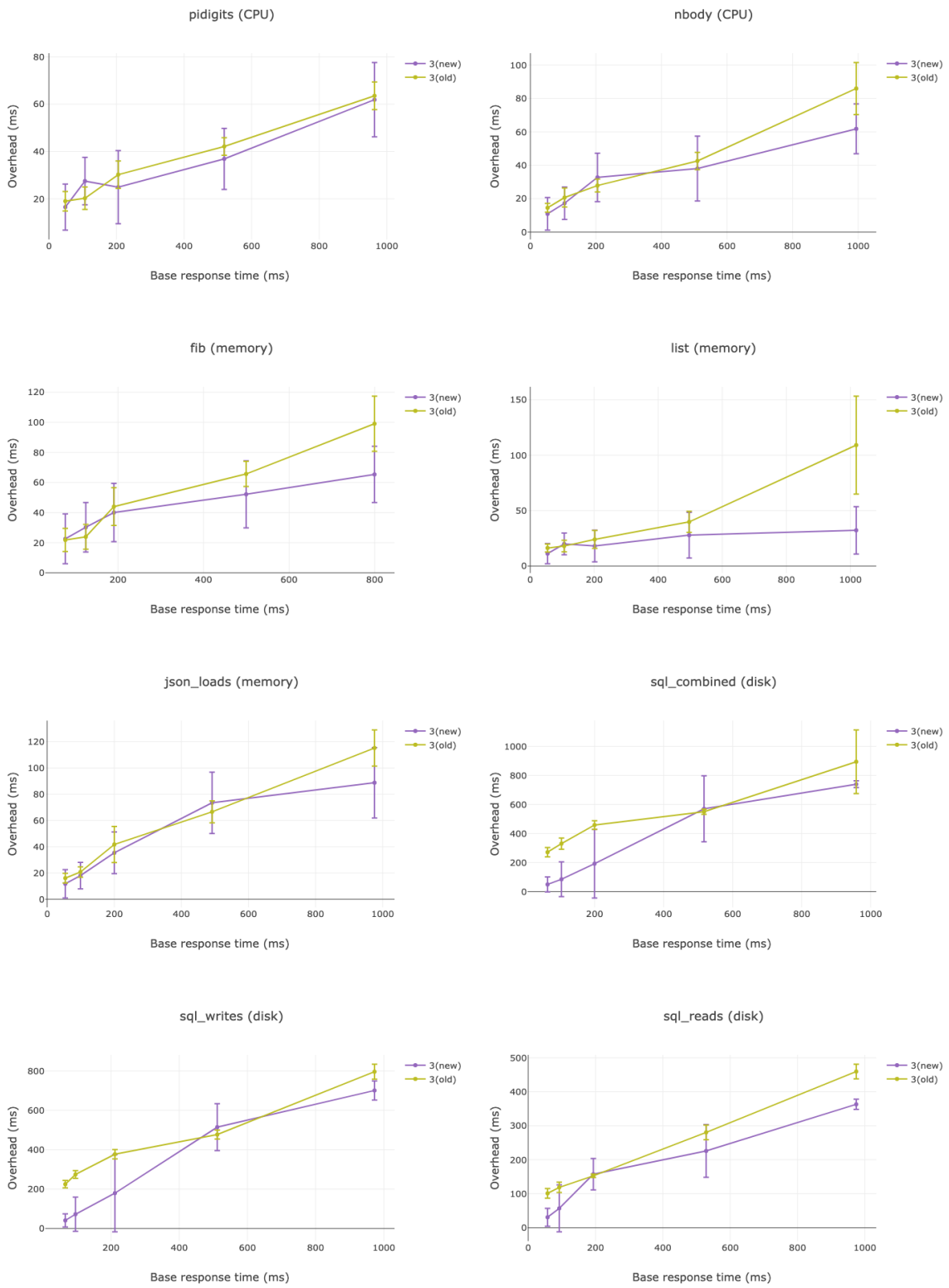


Figure 5.2: Overhead of monitoring level 3 before and after the second iteration

6 THIRD ITERATION

Following the positive results of the previous iterations in terms of improving the performance of monitoring levels 0, 1, and 2, this chapter focuses on reducing the overhead of monitoring level 3.

6.1 RELATED WORK

From the start, we were inclined to believe that the large overhead of the profiler is caused by the high sampling rate. Looking at lines 4 and 5 of Listing 5.6, we see that the profiler thread continuously profiles the request, as frequent as the processor allows it. This "as frequent as possible" strategy was chosen when the profiler was first implemented, with the goal of increasing the accuracy and to give the user as clear of a picture as possible.

To find if this was indeed the best sampling strategy, we looked at other statistical profilers, both for Python and other languages, and looked at their sampling strategy. Luckily, there are many such open source tools, and Table 6.1 shows the results of our investigation. It contains the name of the profiler, the programming language it addresses, the default sampling period, and the ability to configure it.

Name	Language	Sampling period	Configuration
statprof[31]	Python	1 ms	Yes
pyinstrument[32]	Python	1 ms	Yes
perftools[33]	Python	5 ms	No
Pyflame[34]	Python	10 ms	Yes
Py-Spy[35]	Python	10 ms	Yes
Flamegraph[36]	Ruby	1 ms	No
perftools.rb[37]	Ruby	10 ms	Yes
async-profiler[38]	Java	10 ms	Yes

Table 6.1: Statistical profilers

6.2 SAMPLING

All the profilers in the previous table used a sampling period T between 1 ms and 10 ms. In order to compare these sampling periods with that of FMD, we have to quantify the "as frequent as possible" strategy currently used, i.e. count the samples the FMD profiler takes. We did this for all eight micro benchmarks, in the case of the 1000 ms base response times. The results, shown in Table 6.2 along with the

equivalent sampling period, show very different values for disk and non-disk benchmarks.

Benchmark	Samples	Sampling period
`sql_writes`	~1500	1.1 ms
`sql_combined`	~1000	1.7 ms
`sql_reads`	~500	3.4 ms
non-disk benchmarks	~30	35 ms

Table 6.2: Initial sampling periods

The number of samples scale linearly with the base response time, such that the sampling period remains approximately constant for the same benchmark. For instance, in the case of the 100 ms base response time, the profiler collects approximately 100 samples for `sql_combined` and only 3-4 samples for non-disk benchmarks.

The large differences in sampling period can be explained by the fact that in the case of the disk benchmarks, the thread handling the request is mostly idle, waiting for the disk resource. This allows the profiler thread to be run more often and take more stack trace samples. On the other hand, the non-disk benchmarks stress the CPU more, so the profiler thread is scheduled less frequently.

The higher number of samples, and therefore the larger amount of stack lines stored in the database, causes the larger monitoring level 3 overhead observed for the disk benchmarks, compared to the non-disk benchmarks, in Figure 5.2. Another consequence is that any value we choose for the new sampling period, as long as it is smaller than 35 ms, will only affect the results of the three disk benchmarks.

Selecting an appropriate sampling period is a trade off between accuracy and performance. If the sampling period is too low (as it currently is by default), then the performance degrades. If the sampling period is too high, then a lot of the information would be lost. To illustrate this risk of undersampling, we analyzed the results of the profiler on the `sql_combined` benchmark. We used the shortest version, of 50 ms base response time, which had the highest risk of being undersampled.

The `sql_combined` benchmarks performs four database operations: deleting the existing entities from the tables, adding a `Person` entity, adding an `Address` entity, and finally reading all newly inserted entities from the database. Figures A.4 and A.5 show the information captured by the profiler, using different sampling periods T: 1.7 ms (current default), 5 ms, 10 ms, and 20 ms. We see that in the case of the 1.7 ms and 5 ms sampling periods, all four database operations are present. However, as the sampling period increases, the profiler starts *missing* information because the operations take less than the sampling period. For T=10 ms, the profiler misses the delete operation,

while for T=20 ms, it misses everything except for the database read, to which it incorrectly attributes the entire duration of the endpoint.

Figures A.4 and A.5 show not only how the profiler accuracy decreases with the increase of the sampling period, but also how the performance increases. The total duration goes down from 113 ms to 74 ms, for 1.7 ms and 20 ms respectively. We verified this observation by running the `sql_benchmark` with different sampling periods. This time, we used the longest version of the benchmark, taking 1000 ms without the profiler, to limit the effect of random noise.

Sampling period	Overhead
1.7 ms	~750 ms
5 ms	~400 ms
10 ms	~300 ms
20 ms	~240 ms

Table 6.3: Sampling period vs. overhead

Table 6.3 shows how the overhead decreases, as we increase the sampling period. The decrease is however not linear, indicating diminishing returns as the sampling frequency decreases.

Having established the trade off between accuracy and performance, we should normally attempt to find the intersection of the two curves and select that point as the default sampling period. Accuracy, however, is highly subjective and varies from application to application. This leads us to reconsider the purpose of a statistical profiler, which is to find *hot spots* in the code, i.e. code fragments that take a long time to execute, such as slow network calls, heavy database operations, etc. Taking a conservative approach, we could consider a *hot spot* as any operation taking longer than 10 ms. Following Nyquist-Shannon theorem [39], requiring a sampling frequency at least twice of the signal frequency, we obtain a sampling period $T = \frac{10ms}{2} = 5ms$. This value also falls in the middle of the 1 - 10 ms range we found to be used by other statistical profilers in Table 6.1. We therefore decide to set the new default sampling period to 5 ms.

6.3 RESULTS

We ran the micro benchmarks for monitoring level 3 using the new default sampling period. Figure 6.1 shows the new results in purple, compared to the monitoring level 3 overhead of the previous iteration, in yellow. As previously predicted, the CPU and memory benchmarks show very similar results to the previous iteration, due to the fact that the new sampling period did not influence the actual number of samples taken by the profiler. We do see a slight performance regression for the memory benchmarks, especially for the longer versions

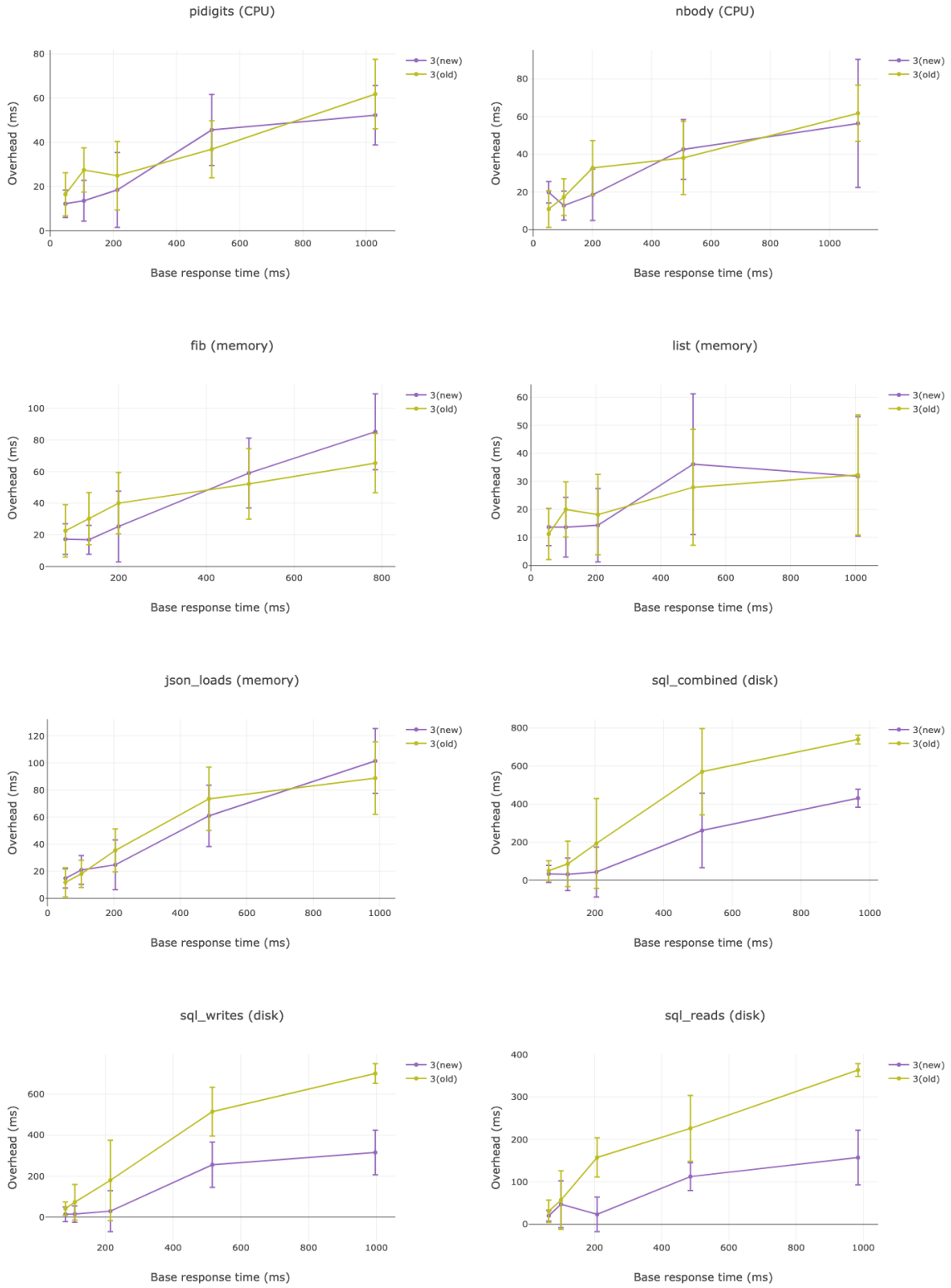


Figure 6.1: Overhead of monitoring level 3 before and after the third iteration

(500 and 1000 ms base response times). This might be explained by a different thread scheduling, causing the profiler to take up CPU cycles that were initially reserved to accessing the memory.

			Benchmark		
			`sql combined`	`sql writes`	`sql reads`
FMD	3 (old)	w_0	31.4 ms	22.9 ms	41.3 ms
		w_1	0.803	0.750	0.340
		R^2	0.95	0.95	0.95
level	3 (new)	w_0	-19.1 ms	-14.1 ms	16.9 ms
		w_1	0.478	0.363	0.150
		R^2	0.98	0.90	0.90

Table 6.4: Linear regression coefficients of the disk benchmarks - level 3

On the other hand, we notice significant improvements, on the order of hundreds of milliseconds, in the case of the disk benchmarks, for all base response times. Table 6.4 shows the parameters of the linear regression we performed for the three disk benchmarks. Notice how both the fixed and proportional components decreased since the previous iteration. The proportional overhead now ranges between 15% and 48%, down from 34% to 80%.

6.4 DISCUSSION

In Chapter 3, we mentioned one benefit of the macro benchmark as being the validation of any changes done to the FMD code that had mixed results with the micro benchmarks. So far, this has not been necessary. Over the course of this process, the results of the micro benchmarks have improved with each iteration. This meant that we had at every step the guarantee that the changes were in the right direction.

Nevertheless, we have thus far always operated in the realm of synthetic workloads. To quantify the benefits of our changes for a typical web application, we reran the macro benchmark. Figure 6.2 shows the results with the right set of violin plots. We included the original results on the left to facilitate comparisons. The exact values, together with the percentage difference from the base duration of the benchmark, are in Table 6.5.

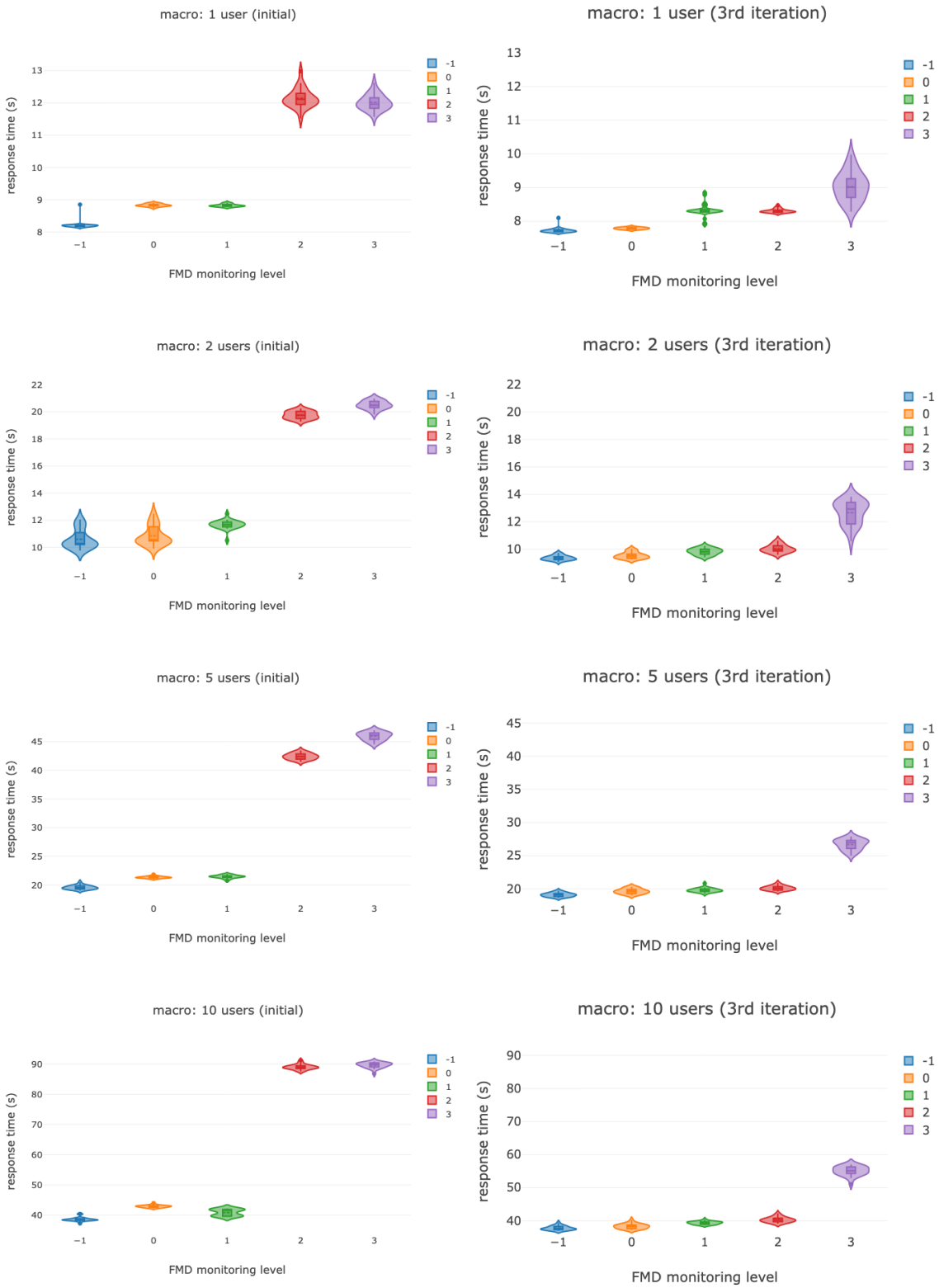


Figure 6.2: Results comparison of the macro benchmark

		FMD monitoring level				
	Users	None	0	1	2	3
initial	1	8.2 s	8.8 s (+7%)	8.8 s (+7%)	12.1 s (+48%)	12.0 s (+46%)
	2	10.6 s	10.9 s (+3%)	11.7 s (+10%)	19.8 s (+86%)	20.5 s (+93%)
	5	19.6 s	21.3 s (+9%)	21.4 s (+10%)	42.4 s (+117%)	45.9 s (+135%)
	10	38.6 s	42.9 s (+11%)	40.9 s (+6%)	89.0 s (+130%)	89.6 s (+132%)
third iteration	1	7.7 s	7.8 s (+1%)	8.3 s (+7%)	8.3 s (+7%)	9.0 s (+17%)
	2	9.3 s	9.5 s (+2%)	9.8 s (+5%)	10.0 s (+7%)	12.7 s (+35%)
	5	19.1 s	19.6 s (+3%)	19.8 s (+4%)	20.1 s (+5%)	26.7 s (+40%)
	10	37.8 s	38.3 s (+1%)	39.3 s (+4%)	40.3 s (+7%)	55.1 s (+46%)

Table 6.5: Results of the macro benchmark after the 3rd iteration

Looking at the above figures and table, the aggregated impact of the performance improvements becomes apparent. We list below the key observations:

- Monitoring level 3 has reduced its overhead from 46%-135% to 17%-46%.
- Monitoring level 2 has enjoyed the greatest overhead reduction, from 48%-130% to 5%-7%, due in part to a shift of functionality (the profiler was replaced by outlier detection), but also to the implementation of the memory cache.
- The overhead of monitoring level 1 decreased the least, with between 2% and 6%.
- Monitoring level 0 has become insignificant in terms of overhead, with between 1% and 3%.

The above observations are further supported by the results of the linear regression, shown in Table 6.6 and Figure 6.3 (notice the different scales used).

		FMD level			
		0	1	2	3
initial	w_0	-0.78 s	0.5 s	-7.37 s	-6.45 s
	w_1	0.131	0.049	1.504	1.518
	R^2	0.98	0.83	0.99	0.99
third iteration	w_0	0.05 s	0.18 s	0 s	-2.15 s
	w_1	0.014	0.035	0.064	0.514
	R^2	0.68	0.93	0.97	0.99

Table 6.6: Linear regression coefficients of the macro benchmark after the 3rd iteration

Having reached this point of the research, we must answer the question of how to proceed from now on. On the one hand, for

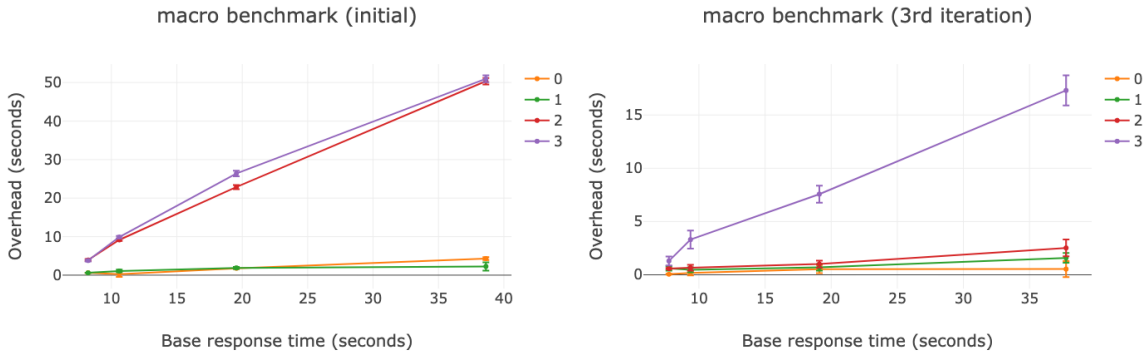


Figure 6.3: Overhead comparison of the macro benchmark

monitoring levels 0, 1, and 2, it is clear, both through the results of the micro- and macro-benchmarks, that any further improvement would have a minimal effect on performance.

On the other hand, the overhead of monitoring level 3, although greatly reduced, still leaves room for further optimization. However, this would require a far greater refactoring effort, with uncertain odds of success. Furthermore, as first explained in Chapter 2, the profiler was never meant to be a general purpose feature, but rather a specialized one, to be used to help identify "hot spots" in an endpoint only when other features of the FMD have marked that endpoint as being slow. Lastly, Question 4 of the user survey shows that only two out of the nine respondents use the profiler, further decreasing the impact of any potential improvement.

With this cost-benefits analysis in mind, and in the interest of time, we decide against performing further iterations.

6.5 OPEN ISSUES

Although the iterative process is at an end, there are still several matters that need to be addressed. Looking at the Points of Improvement (PoI) at the end of Chapter 3, we notice two points insufficiently covered:

- Variance. While we focused on reducing the mean response times, the variance has received a lower priority. This is particularly true for the profiler, as seen in Figure 6.2.
- Concurrency. As the number of concurrent users grows, the overhead of the profiler as percentage of the base response time increases faster too (Table 6.5).

Lastly, all the changes to the code describes in the previous chapters were performed on a separate branch of the repository. The Flask Monitoring Dashboard is an active project and over the course of this

research, other people have contributed to it too. Therefore, we still need to merge our changes into the `master` branch of the repository.

7 CONCLUSIONS AND FUTURE WORK

In this final chapter of the thesis, we summarize the findings of our research and discuss possible ways in which it can be extended.

7.1 CONCLUSIONS

At the start of the thesis, we set out to answer two research questions:

Q1: *What is the performance of the Flask Monitoring Dashboard?*

Q2: *How can the performance of the Flask Monitoring Dashboard be improved?*

We covered the first question in Chapter 3. There, we introduced a new benchmarking framework, designed to measure the response time of a Flask application, with and without the FMD, and to visualize the results. After running the benchmarks, we discussed the overhead, both in terms of absolute values and shape. By means of visual analysis, code inspection, and linear regression, we discovered that, depending on the monitoring level used, the overhead was either fixed or proportional to the base response time of the endpoint.

Chapters 4, 5, and 6 revolved around answering the second research question. Each of them presented one iteration of the cyclical process we used, following the guidelines of a Canonical Action Research[10]. In the first iteration, we dramatically reduced the overhead of monitoring level 2, by simply shifting two features between each other (the profiler and the outlier detection features). In the second iteration, we reduced the fixed overhead of monitoring levels 0, 1, and 2 by implementing a memory cache. Finally, the third iteration showed how a careful selection of the sampling period improved the performance of monitoring level 3. Annex D summarizes the current expected overhead values of the FMD, at the end of our iterative process.

Our contribution to the scientific community is twofold. Firstly, we showed how a combination of micro and macro benchmarks can be used to determine the performance of a system. The micro benchmarks helped us identify large overhead values affecting endpoints using particular types of physical resources (in our case the disk). The macro benchmark was used to better estimate the performance impact of the FMD on a real web application, which in the end is the one that matters the most.

Secondly, we showed how a performance improvement process of a software system can be achieved in an iterative fashion. By using

the previously introduced benchmarks, we identified performance issues, acted upon them, and then reran the benchmarks to validate the changes we introduced.

While our research was centered around the Flask Monitoring Dashboard, the same approach can be generalized and used for other APM solutions. Indeed, the performance impact of any middleware solution on software system can be evaluated and improved in the same manner.

7.2 FUTURE WORK

While we did address both initial research questions, over the course of the thesis several new investigation directions became apparent. Given the limited resources and time frame of our research, these will constitute the topic of future work.

Having seen the effectiveness of benchmarks in identifying performance issues, one possible research direction is to automatically detect performance regressions from the version control system. Similar systems [25][28] already exist and one could investigate how the benchmarks we introduced could be integrated with them. Also, it is important to consider the significant time constraint introduced by running performance tests for every commit in a version control system. Only a small subset of benchmarks should be used, and with fewer runs, which would, in turn, reduce the accuracy of the measurements.

For all the benchmarks we ran during our research, we used an empty FMD database. This was done for practical reasons, as to always have the same conditions between measurements. We do expect that a larger database would negatively impact performance, but this effect has to be quantified. A clear understanding of the relation between database size and performance would help us make better decisions as to what and how much information the FMD should store.

While we defined the performance of the Flask Monitoring Dashboard as the overhead it incurs on the monitored application, we should also consider the performance of the web interface, i.e. the data visualization component, as defined in Chapter 2. This has a direct impact on usability and, as such, optimizing the calls the FMD makes to its database should be considered as a future improvement direction.

In Chapter 6, we argued for a default sampling period of the statistical profiler of 5 ms, as we considered it to be the optimal trade off between accuracy and performance. However, it is possible that for one endpoint we could be interested in all operations taking longer than 10 ms, while for another that value could be 50 ms. In the latter case, the profiler would still be oversampling the request. A solution to this predicament is to implement a variable sampling period. There

are two ways in which this can be achieved. The first is to let the user specify for each endpoint the desired sampling period. The second is to allow users to specify how many samples the profiler should take. Based on this value and on the average response time of an endpoint (now stored in the memory cache), the profiler would determine the sampling period.

A final possible research direction is measuring the overhead of the FMD in terms of CPU and memory. This would be particularly relevant in the context of cloud computing, where cloud consumers pay depending on the amount of requested resources. The same benchmarks could be used, but an additional probe needs to be implemented, which would be sampling the CPU and memory utilization of the system running them.

A

VISUALIZATIONS

This appendix contains some of the visualizations referenced in the thesis.

A.1 MICRO BENCHMARKS

Figures A.1 and A.2 show violin plots of micro benchmarks we considered using, but decided against it because they were too unstable. Figure A.3 shows the violin plots of the disk micro benchmarks we *did* use in our framework, but with MySQL database system.

Figure A.1: File writes benchmark

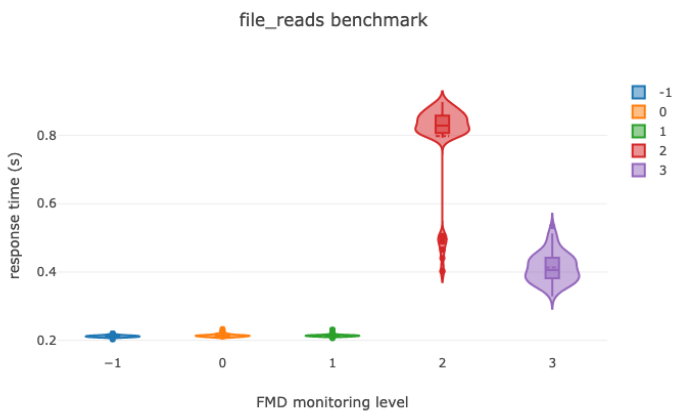


Figure A.2: File writes benchmark

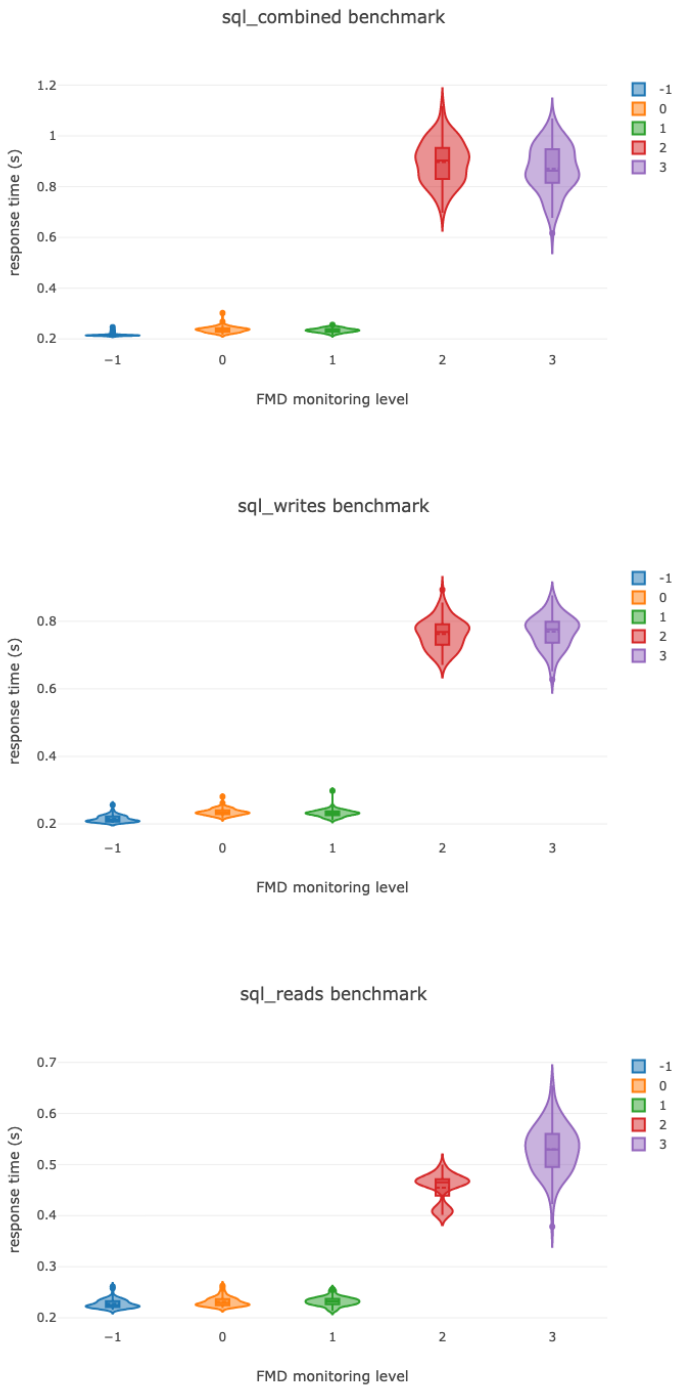


Figure A.3: SQL benchmarks with MySQL

A.2 PROFILER

Figures A.4 and A.5 show the decreasing accuracy of the statistical profiler as the sampling period increases.

Request id: 15 Request date: 19-06-2019 20:23 Expand all

Code-line	Duration	Percentage
@app.route('/sql_combined/')	113 ms	100.0%
@duration	113 ms	100.0%
def sql_combined_endpoint():	113 ms	100.0%
func()	113 ms	100.0%
disk.sql_combined_bm()	113 ms	100.0%
bm_sqlalchemy.bench_sqlalchemy(loops=SQL_COMBINED, writes=2, reads=19)	113 ms	100.0%
session.query(Person).delete(synchronize_session=False)	7.4 ms	6.5%
session.commit()	22.9 ms	20.3%
session.commit()	31.8 ms	28.2%
session.query(Person).all()	50.8 ms	45.0%
thread.start()	0 ms	0.0%

(a) T = 1.7 ms

Request id: 38 Request date: 19-06-2019 20:32 Expand all

Code-line	Duration	Percentage
@app.route('/sql_combined/')	87.9 ms	100.0%
@duration	87.9 ms	100.0%
def sql_combined_endpoint():	87.9 ms	100.0%
func()	87.9 ms	100.0%
disk.sql_combined_bm()	87.9 ms	100.0%
bm_sqlalchemy.bench_sqlalchemy(loops=SQL_COMBINED, writes=2, reads=19)	87.9 ms	100.0%
session.commit()	11.5 ms	13.0%
session.commit()	29.8 ms	33.9%
session.query(Person).all()	41.1 ms	46.8%
session.query(Person).delete(synchronize_session=False)	5.5 ms	6.3%
thread.start()	0 ms	0.0%

(b) T = 5 ms

Figure A.4: Profiler accuracy with sampling periods 1.7 ms and 5 ms

Request id: 49 Request date: 19-06-2019 20:35 Expand all

Code-line	Duration	Percentage
@app.route('/sql_combined/')	79.2 ms	100.0%
@duration	79.2 ms	100.0%
def sql_combined_endpoint():	79.2 ms	100.0%
func()	79.2 ms	100.0%
disk.sql_combined_bm()	79.2 ms	100.0%
bm_sqlalchemy.bench_sqlalchemy(loops=SQL_COMBINED, writes=2, reads=19)	79.2 ms	100.0%
session.commit()	34.8 ms	44.0%
session.query(Person).all()	33.9 ms	42.8%
session.commit()	10.4 ms	13.2%
thread.start()	0 ms	0.0%

(a) T = 10 ms

Request id: 74 Request date: 19-06-2019 20:41 Expand all

Code-line	Duration	Percentage
@app.route('/sql_combined/')	74.1 ms	100.0%
@duration	74.1 ms	100.0%
def sql_combined_endpoint():	74.1 ms	100.0%
func()	74.1 ms	100.0%
disk.sql_combined_bm()	74.1 ms	100.0%
bm_sqlalchemy.bench_sqlalchemy(loops=SQL_COMBINED, writes=2, reads=19)	74.1 ms	100.0%
session.query(Person).all()	74.1 ms	100.0%
thread.start()	0 ms	0.0%

(b) T = 20 ms

Figure A.5: Profiler accuracy with sampling periods 10 ms and 20 ms

B CODE

This appendix contains some of the code snippets and console logs referenced in the thesis.

B.1 CONSOLE LOGS

```
Exception in thread Thread-180:
Traceback (most recent call last):
  File "/usr/local/Cellar/python/3.7.2_2/Frameworks/Python.
      framework/Versions/3.7/lib/python3.7/threading.py", line
      917, in _bootstrap_inner
    self.run()
  File "/Users/bogdan/ENVB/lib/python3.7/site-packages/flask_
      monitoringdashboard/core/profiler/stacktraceProfiler.py",
      line 77, in run
    self._on_thread_stopped()
  File "/Users/bogdan/ENVB/lib/python3.7/site-packages/flask_
      monitoringdashboard/core/profiler/stacktraceProfiler.py",
      line 93, in _on_thread_stopped
    self._outlier_profiler.add_outlier(request_id)
UnboundLocalError: local variable 'request_id' referenced before
    assignment
```

Listing B.1: Monitoring level 3 error

B.2 CODE SNIPPETS

```python
def get_avg_duration(db_session, endpoint_id):
    """ Returns the average duration of all the requests of an
        endpoint. If there are no requests for that endpoint,
        it returns 0.
    :param db_session: session for the database
    :param endpoint_id: id of the endpoint
    :return average duration
    """
    result = db_session.query(func.avg(Request.duration).label('
        average')). \
        filter(Request.endpoint_id == endpoint_id).one()
    if result[0]:
        return result[0]
    return 0
```

Listing B.2: Get average duration

C

USER SURVEY

This appendix contains the results of the survey we conducted among members of the FMD community.

C.1 DISCOVERY AND USAGE

Q1: How did you find the Flask Monitoring Dashboard?
U1: I wrote a paper about it.
U2: Browsing through repositories on Github
U3: Google search for flask monitoring
U4: Google, looking for APM solutions
U5: Browsing through repositories on Github
U6: Google search flask & usage monitoring
U7: Browsing through repositories on Github
U8: Searching on PyPi, Googled for a Python/Flask tracing tool.
U9: Browsing through repositories on Github, In the awesome-flask repository on Github, Searching on PyPi

Q2: Have you evaluated other api-performance monitoring alternatives? If so, what made you decide for Flask-MonitoringDashboard?
U1: they were too complex to install
U2: Simplicity and complete metrics analysis
U3: No alternatives tested yet
U4: I never used FMD in production, the performance is not good enough.
U5: No
U6: No
U7: not yet
U8: Yes; a paid solution - datadoghq. Chose FMD since it was free to test. We came from a Ruby stack, using AppSignal, loved it, paid for it, but doesn't support Python.
U9: I've no found anything that can be compared with this awesom tool :D

Q3: How often do you consult the Flask-MonitoringDashboard to analyze the performance of your Flask application?

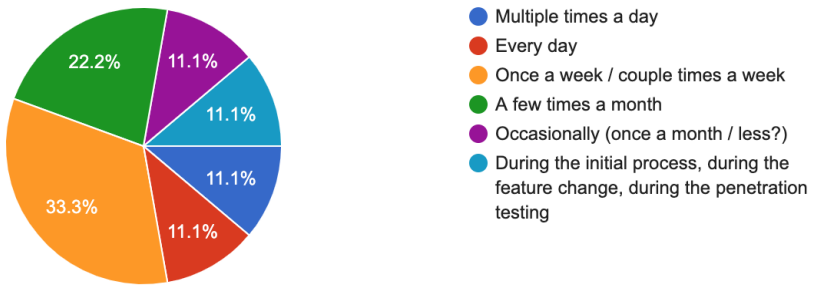


Figure C.1: Q3 responses

C.2 FEATURES

Q4: What features do you use the most in the Flask-MonitoringDashboard

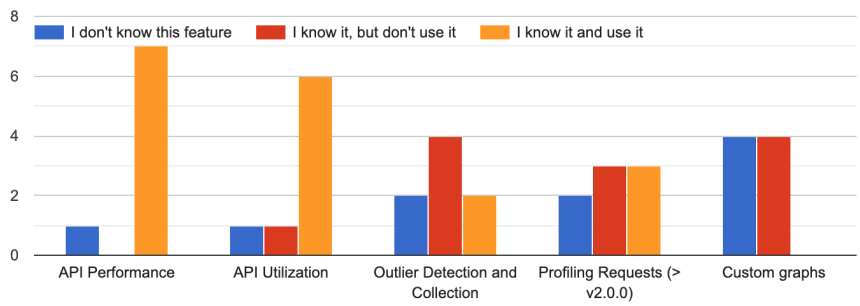


Figure C.2: Q4 responses

Q5: What features would you like to see implemented/improved in the Flask-MonitoringDashboard?

U1: a way of archiving the past history of the profiled info... it slows the app and i don't think i'll ever use it again... although since this is not sure, this is why i'm saying archiving

U2: metrics on users, how many times the same requests ?

U3: Use custom functions for parsing flask request into DB/graphs (or a custom functions called for each requests)

U4: Separate of concerns. An agent to get metrics from my application and another app to receive this data. It is the way to scale.

U5: Seems more like AWS Lambda dashboards

U6:

U7: dockerize the API, give same basic examples

U8: 1. I can't get deployed version to work, we're using Heroku and Heroku handles the repo a little differently. Haven't spent too much exploring. 2. Multiple ways to sort the endpoints on API performance. By number of request, by median execution time, by impact (maybe something like median execution time X # of requests) 3. Support for Postgresql still pretty broken

U9:

C.3 DEPLOYMENT

Q6: How many Flask-applications are you monitoring with Flask-MonitoringDashboard?

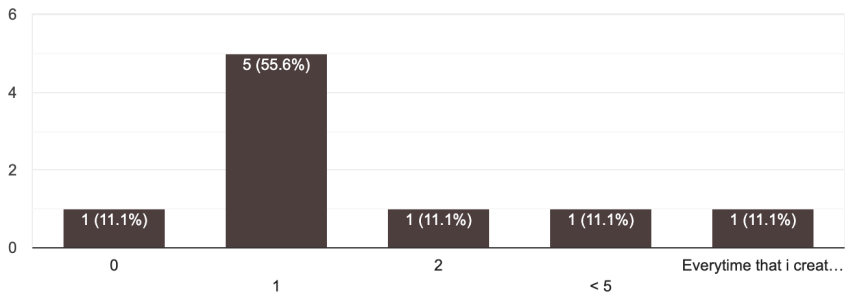


Figure C.3: Q6 responses

Q7: What type of Flask-application(s) do you use the Flask-MonitoringDashboard for? (multiple answers possible)

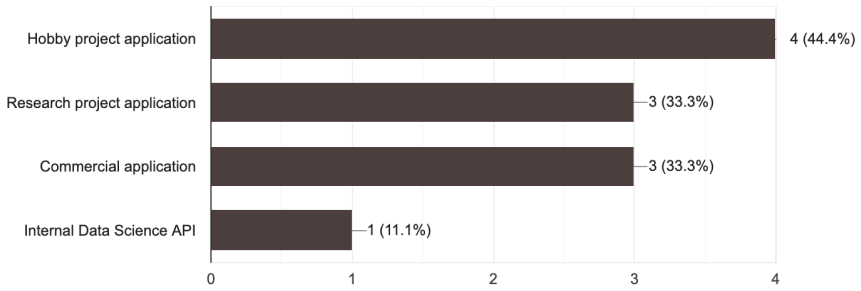


Figure C.4: Q7 responses

Q8: How are your application(s) deployed?

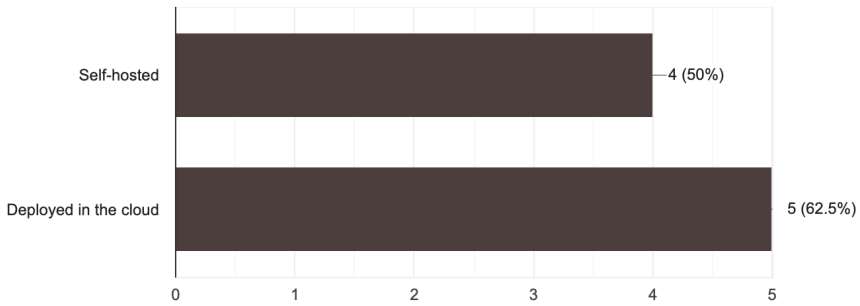


Figure C.5: Q8 responses

Q9: In what region is/are your Flask application(s) deployed?

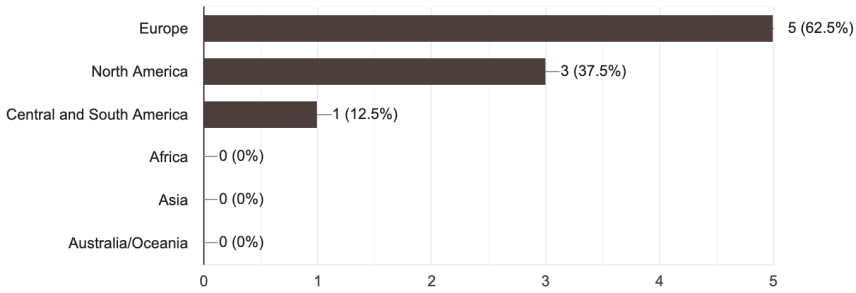


Figure C.6: Q9 responses

Q10: How many developer-years have contributed to your app until now?

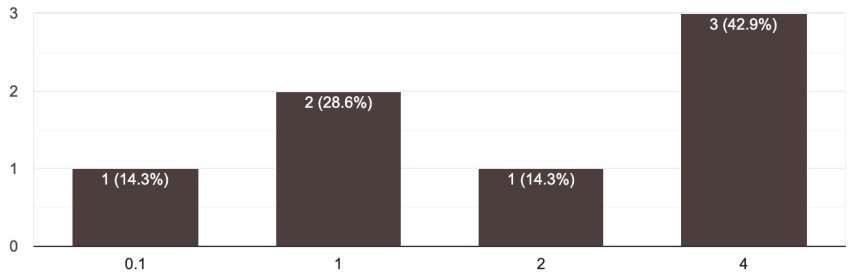


Figure C.7: Q10 responses

Q11: How many developers are currently working on the app?

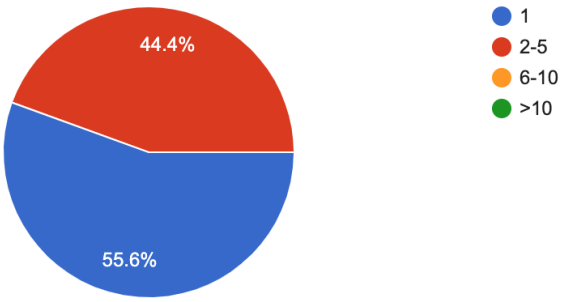


Figure C.8: Q11 responses

Q12: How many requests does your Flask-application(s) receive on a weekly basis for all endpoints?

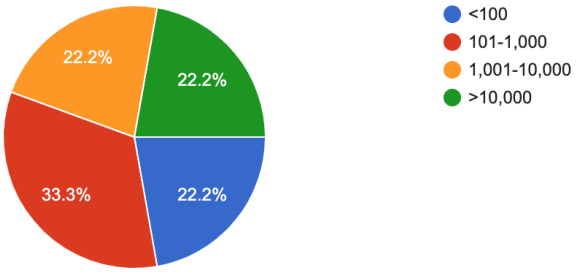


Figure C.9: Q12 responses

Q13: How many endpoints are there on average in your Flask application(s)?

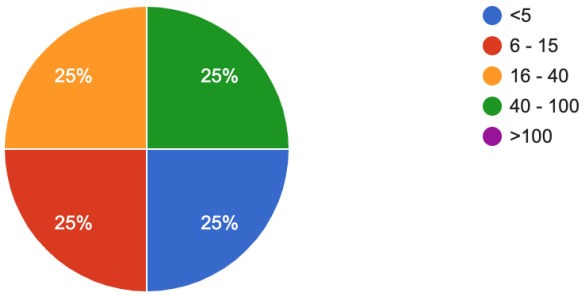


Figure C.10: Q13 responses

Q14: What database do you use for the Flask-MonitoringDashboard?

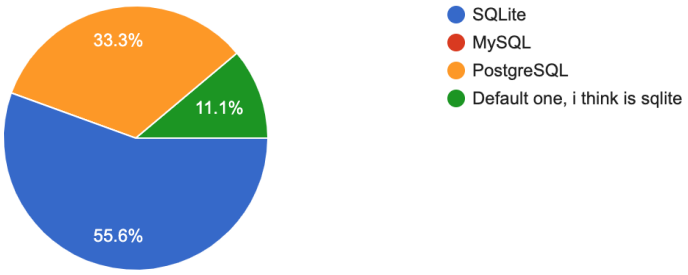


Figure C.11: Q14 responses

C.4 FINAL THOUGHTS

Q15: Would you recommend Flask-MonitoringDashboard to others?

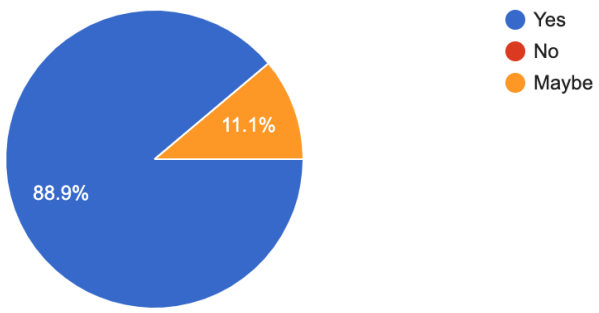


Figure C.12: Q15 responses

Q16: Any other comments, improvements or suggestions?

U1: automatic migration from sqlite to mysql automatic back-up and cleanup of old data from the db - my installation is now more than 1yr old and the app is getting a bit slow

U2: no

U3:

U4: Keep it simple, it has too many features.

U5: No

U6: Very efficient no hassle setup, was impressed by that

U7: Docker compose, config for a reverse proxy, and some basic examples

U8:

U9:

D

FINAL OVERHEAD VALUES

This appendix contains the current expected overhead values, at the end of our performance improvement process.

D.1 MICRO BENCHMARKS

FMD level	Benchmark							
	`pidigits`	`nbody`	`fib`	`list`	`json loads`	`sql combined`	`sql writes`	`sql reads`
0	0.6 ms	0.7 ms	0.2 ms	1.9 ms	-0.8 ms	-0.7 ms	0.0 ms	-0.3 ms
1	2.7 ms	3.5 ms	2.3 ms	-1.0 ms	4 ms	1.8 ms	2.1 ms	1.5 ms
2	4.6 ms	4.7 ms	4.4 ms	4.4 ms	2.2 ms	3.0 ms	2.4 ms	3.8 ms

Table D.1: Final mean overhead values for monitoring levels 0, 1, and 2

FMD level		Benchmark							
		`pidigits`	`nbody`	`fib`	`list`	`json loads`	`sql combined`	`sql writes`	`sql reads`
3	w_0	16.9 ms	13.8 ms	23.8 ms	14.8 ms	14.6 ms	-19.1 ms	-14.1 ms	16.9 ms
	w_1	0.045	0.049	0.54	0.019	0.084	0.478	0.363	0.150
	R^2	0.95	0.93	0.94	0.82	0.90	0.98	0.90	0.90

Table D.2: Final linear regression coefficients for monitoring level 3

D.2 MACRO BENCHMARK

Users	FMD monitoring level				
	None	0	1	2	3
1	7.7 s	7.8 s (+1%)	8.3 s (+7%)	8.3 s (+7%)	9.0 s (+17%)
2	9.3 s	9.5 s (+2%)	9.8 s (+5%)	10.0 s (+7%)	12.7 s (+35%)
5	19.1 s	19.6 s (+3%)	19.8 s (+4%)	20.1 s (+5%)	26.7 s (+40%)
10	37.8 s	38.3 s (+1%)	39.3 s (+4%)	40.3 s (+7%)	55.1 s (+46%)

Table D.3: Final overhead values of the macro benchmark

BIBLIOGRAPHY

[1] Fiona Nah. "A Study on Tolerable Waiting Time: How Long Are Web Users Willing to Wait?" In: vol. 23. Jan. 2003, p. 285. DOI: 10.1080/01449290410001669914.

[2] New Relic. *New Relic APM*. July 9, 2019. URL: https://newrelic.com/products/application-monitoring.

[3] Dynatrace. *Dynatrace*. July 9, 2019. URL: https://www.dynatrace.com/platform/application-performance-management/.

[4] AppDynamics. *AppDynamics APM*. July 9, 2019. URL: https://www.appdynamics.com/product/application-performance-management/.

[5] André van Hoorn, Jan Waller, and Wilhelm Hasselbring. "Kieker: A Framework for Application Performance Monitoring and Dynamic Software Analysis." In: *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*. ICPE '12. Boston, Massachusetts, USA: ACM, 2012, pp. 247–248. ISBN: 978-1-4503-1202-8. DOI: 10.1145/2188286.2188326. URL: http://doi.acm.org/10.1145/2188286.2188326.

[6] Naver. *Pinpoint*. https://github.com/naver/pinpoint. 2019.

[7] P. Vogel, T. Klooster, V. Andrikopoulos, and M. Lungu. "A Low-Effort Analytics Platform for Visualizing Evolving Flask-Based Python Web Services." In: *2017 IEEE Working Conference on Software Visualization (VISSOFT)*. 2017, pp. 109–113. DOI: 10.1109/VISSOFT.2017.13.

[8] Patrick Vogel. *Flask-MonitoringDashboard*. https://github.com/flask-dashboard/Flask-MonitoringDashboard. 2019.

[9] Mircea F. Lungu, Luc van den Brand, Dan Chirtoaca, and Martin Avagyan. "As We May Study: Towards the Web As a Personalized Language Textbook." In: *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. CHI '18. Montreal QC, Canada: ACM, 2018, 338:1–338:12. ISBN: 978-1-4503-5620-6. DOI: 10.1145/3173574.3173912. URL: http://doi.acm.org/10.1145/3173574.3173912.

[10] Robert Davison, Maris Martinsons, and Ned Kock. "Principles of canonical action research." In: *Inf. Syst. J.* 14 (Jan. 2004), pp. 65–. DOI: 10.1111/j.1365-2575.2004.00162.x.

[11] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture - Volume 1: A System of Patterns*. Wiley Publishing, 1996. Chap. 2. Architectural Patterns, pp. 31–52. ISBN: 0471958697, 9780471958697.

[12] Cristiana Amza, Anupam Chanda, Alan L. Cox, Sameh Elnikety, Romer Andres Gil, Karthick Rajamani, Willy Zwaenepoel, and Emmanuel Cecchet. "Specification and implementation of dynamic Web site benchmarks." In: 2002.

[13] Andy Georges, Dries Buytaert, and Lieven Eeckhout. "Statistically Rigorous Java Performance Evaluation." In: *SIGPLAN Not.* 42.10 (Oct. 2007), pp. 57–76. ISSN: 0362-1340. DOI: 10.1145/1297105.1297033. URL: http://doi.acm.org/10.1145/1297105.1297033.

[14] S. Ray, B. Simion, and A. Demke Brown. "Jackpine: A benchmark to evaluate spatial database performance." In: *2011 IEEE 27th International Conference on Data Engineering*. 2011, pp. 1139–1150. DOI: 10.1109/ICDE.2011.5767929.

[15] Daniel A. Menasce. "Load Testing, Benchmarking, And Application Performance Management For The Web." In: *in Proc. 2002 Computer Management Group Conference*. 2002, pp. 271–281.

[16] C. M. Aderaldo, N. C. Mendonça, C. Pahl, and P. Jamshidi. "Benchmark Requirements for Microservices Architecture Research." In: *2017 IEEE/ACM 1st International Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering (ECASE)*. 2017, pp. 8–13. DOI: 10.1109/ECASE.2017.4.

[17] Ludmila Cherkasova and Rob Gardner. "Measuring CPU Overhead for I/O Processing in the Xen Virtual Machine Monitor." In: *Proceedings of the Annual Conference on USENIX Annual Technical Conference*. ATEC '05. Anaheim, CA: USENIX Association, 2005, pp. 24–24. URL: http://dl.acm.org/citation.cfm?id=1247360.1247384.

[18] J. Li, Q. Wang, D. Jayasinghe, J. Park, T. Zhu, and C. Pu. "Performance Overhead among Three Hypervisors: An Experimental Study Using Hadoop Benchmarks." In: *2013 IEEE International Congress on Big Data*. 2013, pp. 9–16. DOI: 10.1109/BigData.Congress.2013.11.

[19] Roberto R. Expósito, Guillermo L. Taboada, Sabela Ramos, Juan Touriño, and Ramón Doallo. "Performance Analysis of HPC Applications in the Cloud." In: *Future Gener. Comput. Syst.* 29.1 (Jan. 2013), pp. 218–229. ISSN: 0167-739X. DOI: 10.1016/j.future.2012.06.009. URL: http://dx.doi.org/10.1016/j.future.2012.06.009.

[20] Victor Stinner. *performance: Python Performance Benchmark Suite*. https://github.com/python/performance. 2019.

[21] Bogdan Petre. *Benchmark: Measuring the overhead of Flask Monitoring Dashboard*. https://github.com/bogdanp05/Benchmark. 2019.

[22] Victor Stinner. *perf: Toolkit to write, run and analyze benchmarks*. https://github.com/vstinner/perf. 2019.

[23] Jerry L. Hintze and Ray D. Nelson. "Violin Plots: A Box Plot-Density Trace Synergism." In: *The American Statistician* 52.2 (1998), pp. 181–184. DOI: 10.1080/00031305.1998.10480559. eprint: https://amstat.tandfonline.com/doi/pdf/10.1080/00031305.1998.10480559. URL: https://amstat.tandfonline.com/doi/abs/10.1080/00031305.1998.10480559.

[24] Ethem Alpaydin. *Introduction to Machine Learning*. 2nd. The MIT Press, 2010. Chap. Parametric Methods - Regression, pp. 73–76. ISBN: 026201243X, 9780262012430.

[25] Q. Luo, D. Poshyvanyk, and M. Grechanik. "Mining Performance Regression Inducing Code Changes in Evolving Software." In: *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*. 2016, pp. 25–36. DOI: 10.1109/MSR.2016.013.

[26] Christoph Heger, Jens Happe, and Roozbeh Farahbod. "Automated Root Cause Isolation of Performance Regressions During Software Development." In: *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*. ICPE '13. Prague, Czech Republic: ACM, 2013, pp. 27–38. ISBN: 978-1-4503-1636-1. DOI: 10.1145/2479871.2479879. URL: http://doi.acm.org/10.1145/2479871.2479879.

[27] Juan Pablo Sandoval Alcocer, Alexandre Bergel, and Marco Tulio Valente. "Learning from Source Code History to Identify Performance Failures." In: *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*. ICPE '16. Delft, The Netherlands: ACM, 2016, pp. 37–48. ISBN: 978-1-4503-4080-9. DOI: 10.1145/2851553.2851571. URL: http://doi.acm.org/10.1145/2851553.2851571.

[28] Weiyi Shang, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. "Automated Detection of Performance Regressions Using Regression Models on Clustered Performance Counters." In: *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*. ICPE '15. Austin, Texas, USA: ACM, 2015, pp. 15–26. ISBN: 978-1-4503-3248-4. DOI: 10.1145/2668930.2688052. URL: http://doi.acm.org/10.1145/2668930.2688052.

[29] H. Garcia-Molina and K. Salem. "Main memory database systems: an overview." In: *IEEE Transactions on Knowledge and Data Engineering* 4.6 (1992), pp. 509–516. ISSN: 1041-4347. DOI: 10.1109/69.180602.

[30] Eric Brewer. "Towards robust distributed systems." In: Jan. 2000, p. 7. DOI: 10.1145/343477.343502.

[31] Andy Wingo. *statprof - statistical profiling for Python*. https://github.com/bos/statprof.py. 2019.

[32] Joe Rickerby. *pyinstrument - Call stack profiler for Python. Shows you why your code is slow!* https://github.com/joerick/pyinstrument. 2019.

[33] Eben Freeman. *nylas-perftools:Distributed profiling on the cheap*. https://github.com/nylas/nylas-perftools. 2019.

[34] Uber Open Source. *Pyflame: A Ptracing Profiler For Python*. https://github.com/uber/pyflame. 2019.

[35] Ben Frederickson. *Py-Spy: A sampling profiler for Python programs.* https://github.com/benfred/py-spy. 2019.

[36] Sam Saffron. *Flamegraph profiling support for Ruby 2.0*. https://github.com/SamSaffron/flamegraph. 2019.

[37] Aman Gupta. *gperftools for ruby code*. https://github.com/tmm1/perftools.rb. 2019.

[38] Andrei Pangin. *Sampling CPU and HEAP profiler for Java featuring AsyncGetCallTrace + perf_events*. https://github.com/jvm-profiling-tools/async-profiler. 2019.

[39] C.E. Shannon. "Communication in the Presence of Noise." In: *Proceedings of the IRE* 37.1 (1949), pp. 10–21. DOI: 10.1109/jrproc.1949.232969. URL: https://doi.org/10.1109/jrproc.1949.232969.