Benchmarking Microservice Platforms and Applications in the Cloud

vorgelegt von

Martin Grambow, M.Sc.
ORCID: 0000-0001-0000-0000

an der Fakultät IV - Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Ingenieurswissenschaften
- Dr.-Ing. -

genehmigte Dissertation
Tag der wissenschaftlichen Aussprache: 01.01.2024

Promotionsausschuss

Vorsitzender	Prof. Dr. Henning Sprekeler
Gutachter	Prof. Dr. David Bermbach
Gutachter	Prof. Dr. Willy Hasselbrink
Gutachter	Prof. Dr. Odej Kao

Berlin 2024

Abstract

dsafsahfdusa fdsuiafds uaifhdsau fhdsauifhd safhdusa fdusafhd saufhds dusfhduafdsfd sachdaczu sacshudha sucdxsa cdascdiap uia sauchs chuahdsuachdsuacdshaucdsa csah cduiachupsucdhuicdhsaucda updasdiscdisas. dsafsahfdusa fdsuiafds uaifhdsau fhdsauifhd safhdusa fdusafhd saufhds dusfhduafdsfd sachdaczu sacshudha sucdxsa cdascdiap uia sauchs chuahdsuachdsuacdshaucdsa csah cduiachupsucdhuicdhsaucda updasdiscdisas. dsafsahfdusa fdsuiafds uaifhdsau fhdsauifhd safhdusa fdusafhd saufhds dusfhduafdsfd sachdaczu sacshudha sucdxsa cdascdiap uia sauchs chuahdsuachdsuacdshaucdsa csah cduiachupsucdhuicdhsaucda updasdiscdisas. dsafsahfdusa fdsuiafds uaifhdsau fhdsauifhd safhdusa fdusafhd saufhds dusfhduafdsfd sachdaczu sacshudha sucdxsa cdascdiap uia sauchs chuahdsuachdsuacdshaucdsa csah cduiachupsucdhuicdhsaucda updasdiscdisas. dsafsahfdusa fdsuiafds uaifhdsau fhdsauifhd safhdusa fdusafhd saufhds dusfhduafdsfd sachdaczu sacshudha su

Kurzdarstellung

dsafsahfdusa fdsuiafds uaifhdsau fhdsauifhd safhdusa fdusafhd saufhds dusfhduafdsfd sachdaczu sacshudha sucdxsa cdascdiap uia sauchs chuahdsuachdsuacdshaucdsa csah cduiachupsucdhuicdhsaucda updasdiscdisas. dsafsahfdusa fdsuiafds uaifhdsau fhdsauifhd safhdusa fdusafhd saufhds dusfhduafdsfd sachdaczu sacshudha sucdxsa cdascdiap uia sauchs chuahdsuachdsuacdshaucdsa csah cduiachupsucdhuicdhsaucda updasdiscdisas. dsafsahfdusa fdsuiafds uaifhdsau fhdsauifhd safhdusa fdusafhd saufhds dusfhduafdsfd sachdaczu sacshudha sucdxsa cdascdiap uia sauchs chuahdsuachdsuacdshaucdsa csah cduiachupsucdhuicdhsaucda updasdiscdisas. dsafsahfdusa fdsuiafds uaifhdsau fhdsauifhd safhdusa fdusafhd saufhds dusfhduafdsfd sachdaczu sacshudha sucdxsa cdascdiap uia sauchs chuahdsuachdsuacdshaucdsa csah cduiachupsucdhuicdhsaucda updasdiscdisas. dsafsahfdusa fdsuiafds uaifhdsau fhdsauifhd safhdusa fdusafhd saufhds dusfhduafdsfd sachdaczu sacshudha su

Danksagung

Preface

dsafsahfdusa fdsuiafds uaifhdsau fhdsauifhd safhdusa fdusafhd saufhds dusfhduafdsfd sachdaczu sacshudha sucdxsa cdascdiap uia sauchs chuahdsuachdsuacdshaucdsa csah cduiachupsucdhuicdhsaucda updasdiscdisas. dsafsahfdusa fdsuiafds uaifhdsau fhdsauifhd safhdusa fdusafhd saufhds dusfhduafdsfd sachdaczu sacshudha sucdxsa cdascdiap uia sauchs chuahdsuachdsuacdshaucdsa csah cduiachupsucdhuicdhsaucda updasdiscdisas. dsafsahfdusa fdsuiafds uaifhdsau fhdsauifhd safhdusa fdusafhd saufhds dusfhduafdsfd sachdaczu sacshudha sucdxsa cdascdiap uia sauchs chuahdsuachdsuacdshaucdsa csah cduiachupsucdhuicdhsaucda updasdiscdisas. dsafsahfdusa fdsuiafds uaifhdsau fhdsauifhd safhdusa fdusafhd saufhds dusfhduafdsfd sachdaczu sacshudha sucdxsa cdascdiap uia sauchs chuahdsuachdsuacdshaucdsa csah cduiachupsucdhuicdhsaucda updasdiscdisas. dsafsahfdusa fdsuiafds uaifhdsau fhdsauifhd safhdusa fdusafhd saufhds dusfhduafdsfd sachdaczu sacshudha sucdxsa cdascdiap uia sauchs chuahdsuachdsuacdshaucdsa csah cduiachupsucdhuicdhsaucda updasdiscdisas. dsafsahfdusa fdsuiafds uaifhdsau fhdsauifhd safhdusa fdusafhd saufhds dusfhduafdsfd sachdaczu sacshudha sucdxsa cdascdiap uia sauchs chuahdsuachdsuacdshaucdsa csah cduiachupsucdhuicdhsaucda updasdiscdisas. dsafsahfdusa fdsuiafds uaifhdsau fhdsauifhd safhdusa fdusafhd saufhds dusfhduafdsfd sachdaczu sacshudha sucdxsa cdascdiap uia sauchs chuahdsuachdsuacdshaucdsa csah cduiachupsucdhuicdhsaucda updasdiscdisas. dsafsahfdusa fdsuiafds uaifhdsau fhdsauifhd safhdusa fdusafhd saufhds dusfhduafdsfd sachdaczu sacshudha sucdxsa cdascdiap uia sauchs chuahdsuachdsuacdshaucdsa csah cduiachupsucdhuicdhsaucda updasdiscdisas. dsafsahfdusa fdsuiafds uaifhdsau fhdsauifhd safhdusa fdusafhd saufhds dusfhduafdsfd sachdaczu sacshudha sucdxsa cdascdiap uia sauchs chuahdsuachdsuacdshaucdsa csah cduiachupsucdhuicdhsaucda updasdiscdisas. dsafsahfdusa fdsuiafds uaifhdsau fhdsauifhd safhdusa fdusafhd saufhds dusfhduafdsfd sach-

daczu sacshudha sucdxsa cdascdiap uia sauchs chuahdsuachdsuacdshaucdsa csah cduiachupsucdhuicdhsaucda updasdiscdisas. dsafsahfdusa fdsuiafds uaifhdsau fhdsauifhd safhdusa fdusafhd saufhds dusfhduafdsfd sachdaczu sacshudha sucdxsa cdascdiap uia sauchs chuahdsuachdsuacdshaucdsa csah cduiachupsucdhuicdhsaucda updasdiscdisas. dsafsahfdusa fdsuiafds uaifhdsau fhdsauifhd safhdusa fdusafhd saufhds dusfhduafdsfd sachdaczu sacshudha sucdxsa cdascdiap uia sauchs chuahdsuachdsuacdshaucdsa csah cduiachupsucdhuicdhsaucda updasdiscdisas. dsafsahfdusa fdsuiafds uaifhdsau fhdsauifhd safhdusa fdusafhd saufhds dusfhduafdsfd sachdaczu sacshudha sucdxsa cdascdiap uia sauchs chuahdsuachdsuacdshaucdsa csah cduiachupsucdhuicdhsaucda updasdiscdisas. dsafsahfdusa fdsuiafds uaifhdsau fhdsauifhd safhdusa fdusafhd saufhds dusfhduafdsfd sachdaczu sacshudha sucdxsa cdascdiap uia sauchs chuahdsuachdsuacdshaucdsa csah cduiachupsucdhuicdhsaucda updasdiscdisas. dsafsahfdusa fdsuiafds uaifhdsau fhdsauifhd safhdusa fdusafhd saufhds dusfhduafdsfd sachdaczu sacshudha sucdxsa cdascdiap uia sauchs chuahdsuachdsuacdshaucdsa csah cduiachupsucdhuicdhsaucda updasdiscdisas. dsafsahfdusa fdsuiafds uaifhdsau fhdsauifhd safhdusa fdusafhd saufhds dusfhduafdsfd sachdaczu sacshudha sucdxsa cdascdiap uia sauchs chuahdsuachdsuacdshaucdsa csah cduiachupsucdhuicdhsaucda updasdiscdisas. dsafsahfdusa fdsuiafds uaifhdsau fhdsauifhd safhdusa fdusafhd saufhds dusfhduafdsfd sachdaczu sacshudha sucdxsa cdascdiap uia sauchs chuahdsuachdsuacdshaucdsa csah cduiachupsucdhuicdhsaucda updasdiscdisas. dsafsahfdusa fdsuiafds uaifhdsau fhdsauifhd safhdusa fdusafhd saufhds dusfhduafdsfd sachdaczu sacshudha sucdxsa cdascdiap uia sauchs chuahdsuachdsuacdshaucdsa csah cduiachupsucdhuicdhsaucda updasdiscdisas. dsafsahfdusa fdsuiafds uaifhdsau fhdsauifhd safhdusa fdusafhd saufhds dusfhduafdsfd sachdaczu sacshudha sucdxsa cdascdiap uia sauchs chuahdsuachdsuacdshaucdsa csah cduiachupsucdhuicdhsaucda updasdiscdisas. dsafsahfdusa fdsuiafds uaifhdsau fhdsauifhd safhdusa fdusafhd saufhds dusfhduafdsfd sachdaczu sacshudha sucdxsa cdascdiap uia sauchs chuahdsuachd-

suacdshaucdsa csah cduiachupsucdhuicdhsaucda updasdiscdisas. dsafsahfdusa fdsuiafds uaifhdsau fhdsauifhd safhdusa fdusafhd saufhds dusfhduafdsfd sachdaczu sacshudha sucdxsa cdascdiap uia sauchs chuahdsuachdsuacdshaucdsa csah cduiachupsucdhuicdhsaucda updasdiscdisas. dsafsahfdusa fdsuiafds uaifhdsau fhdsauifhd safhdusa fdusafhd saufhds dusfhduafdsfd sachdaczu sacshudha sucdxsa cdascdiap uia sauchs chuahdsuachdsuacdshaucdsa csah cduiachupsucdhuicdhsaucda updasdiscdisas. dsafsahfdusa fdsuiafds uaifhdsau fhdsauifhd safhdusa fdusafhd saufhds dusfhduafdsfd sachdaczu sacshudha sucdxsa cdascdiap uia sauchs chuahdsuachdsuacdshaucdsa csah cduiachupsucdhuicdhsaucda updasdiscdisas. dsafsahfdusa fdsuiafds uaifhdsau fhdsauifhd safhdusa fdusafhd saufhds dusfhduafdsfd sachdaczu sacshudha sucdxsa cdascdiap uia sauchs chuahdsuachdsuacdshaucdsa csah cduiachupsucdhuicdhsaucda updasdiscdisas. dsafsahfdusa fdsuiafds uaifhdsau fhdsauifhd safhdusa fdusafhd saufhds dusfhduafdsfd sachdaczu sacshudha sucdxsa cdascdiap uia sauchs chuahdsuachdsuacdshaucdsa csah cduiachupsucdhuicdhsaucda updasdiscdisas. dsafsahfdusa fdsuiafds uaifhdsau fhdsauifhd safhdusa fdusafhd saufhds dusfhduafdsfd sachdaczu sacshudha sucdxsa cdascdiap uia sauchs chuahdsuachdsuacdshaucdsa csah cduiachupsucdhuicdhsaucda updasdiscdisas. dsafsahfdusa fdsuiafds uaifhdsau fhdsauifhd safhdusa fdusafhd saufhds dusfhduafdsfd sachdaczu sacshudha sucdxsa cdascdiap uia sauchs chuahdsuachdsuacdshaucdsa csah cduiachupsucdhuicdhsaucda updasdiscdisas. dsafsahfdusa fdsuiafds uaifhdsau fhdsauifhd safhdusa fdusafhd saufhds dusfhduafdsfd sachdaczu sacshudha sucdxsa cdascdiap uia sauchs chuahdsuachdsuacdshaucdsa csah cduiachupsucdhuicdhsaucda updasdiscdisas. dsafsahfdusa fdsuiafds uaifhdsau fhdsauifhd safhdusa fdusafhd saufhds dusfhduafdsfd sachdaczu sacshudha sucdxsa cdascdiap uia sauchs chuahdsuachdsuacdshaucdsa csah cduiachupsucdhuicdhsaucda updasdiscdisas.

Table of Contents

Acronyms

ECU European currency unit

EU EuropÃ¤ische Union

cg CGcall graph

Part I

Foundations

Chapter 1

Introduction

Cloud computing has revolutionized the way businesses operate by providing a scalable and cost-effective infrastructure for various microservice-oriented applications. Due to the numerous advantages offered by using cloud computing, a large portion of applications are nowadays hosted as composite services in the cloud. There, the individual services that constitute an application can be scaled up and down as needed.

While functional requirements ensure the proper execution of service tasks, non-functional requirements such as performance contribute significantly to the user experience. The speed of services in the cloud directly correlates with user satisfaction and, subsequently, the financial success of businesses. Slow services can lead to frustrated users, negatively impacting their experience and potentially driving them away. Moreover, decreased performance imply higher cloud infrastructure costs as service tasks have longer execution times and or more cloud infrastructure is needed. Thus, ensuring compliance with non-functional requirements such as performance is very important for services hosted in cloud environments.

Software performance changes are costly and often hard to detect pre-release. Thus, continuous performance evaluations are essential for identifying and addressing potential performance issues promptly. While embedding functional tests in continuous integration and deployment (CI/CD) pipelines for ensuring functional requirements is standard practice, continuous benchmarking, i.e., the regular assessment of non-functional properties such as performance in a realistic staging environment before releasing a new application version, is not yet widely adopted and comes with its own challenges.

In this thesis, we delve into three of these challenges and propose suitable solutions to enable continuous performance evaluations of microservice-based applications in cloud computing environments and its platforms.

1.1 Problem Statements

Figure X illustrates an abstract benchmarking scheme of a microservice-oriented application hosted on a cloud provider platform and evaluated from a client perspective. Here, the client sends an artificial but realistic load of requests to the application and measures performance parameters. This specific load, i.e., the specific parameter values and their syntax, differs for each service and thus these requests had to be created individually and manually for each application: Creating a new customer in a customer management service requires different values than reserving seats on a train. While there are approaches to derive these parameters and the load from functional tests (which are often available for applications), they are usually unrealistic because they mostly cover error situations and customers typically enter correct postal codes and not letters. Furthermore, a second issue with manually created benchmark loads is that they do not adapt with the ongoing development of the application code itself. Instead, they must be adjusted in addition to the code changes, e.g., when a new parameter value is introduced for the customer service, making them difficult to maintain (see problem A).

While application benchmarks, which evaluate a complete application in a realistic environment from a client perspective, are hard to set up as many components are involved, microbenchmarks evaluate an application on function-level and repeatably call the respective function while collecting metrics. Thus, a microbenchmark suite, i.e., a set of microbenchmarks, can provide fast and easy performance metrics and insights, but does not cover the interaction and integration of different components. Moreover, it is hard to estimate the impact of a detected performance change in a microbenchmark for the performance metrics of the overall application. E.g., a slower backup import might raise an alarm in the respective microbenchmark but this does not have an impact on running applications as this task only occurs rarely. Nevertheless, they are easy to embed in CI/CD pipelines and provide a fast performance feedback for developers. With larger microbenchmark suites, however, the execution time of the microbenchmarks increase drastically and often takes several hours, making them impractical to use and evaluate performance metrics for every single code change in detail; hence, trade-offs have to be made(see problem B).

Hosting services in the cloud means that service owners hand over a certain degree of control to the cloud provider. While the cloud provider's customer can configure certain parameters and the cloud provider complies with certain general quality metrics, the customer has little influence on the specific execution of a service task and its performance metrics. Particularly when using the Function as a Service (FaaS) model, where service owners only share the source code and the cloud provider takes care of service execution, scaling, etc., there are only limited options for customers. For realistic comparison and study of different FaaS systems and their configuration options, FaaS application developers rely on FaaS benchmarking frameworks.

Existing frameworks, however, tend to evaluate only single isolated aspects and a more holistic application-centric benchmarking framework is still missing (see problem C).

1.2 Contributions

This thesis addresses the problems outlined above with the following main contributions published in:

- Martin Grambow, Fabian Lehmann, and David Bermbach. "Continuous Benchmarking: Using System Benchmarking in Build Pipelines". In: *Proc. of the Workshop on Service Quality and Quantitative Evaluation in new Emerging Technologies (SQUEET '19)*. IEEE, 2019, pp. 241–246
- Martin Grambow et al. "Benchmarking Microservice Performance: A Pattern-based Approach". In: *Proc. of the 35th ACM Symposium on Applied Computing (SAC 2020)*. ACM, 2020
- Martin Grambow, Erik Wittern, and David Bermbach. "Benchmarking the Performance of Microservice Applications". In: *SIGAPP Applied Computing Review*. ACM, 2020, pp. 20–34
- Martin Grambow et al. "Using application benchmark call graphs to quantify and improve the practical relevance of microbenchmark suites". In: *PeerJ Computer Science*. PeerJ, 2021
- Martin Grambow et al. "Using Microbenchmark Suites to Detect Application Performance Changes". In: *Transactions on Cloud Computing*. IEEE, 2023
- Martin Grambow et al. "BeFaaS: An Application-Centric Benchmarking Framework for FaaS Platforms". In: *Proc. 9th IEEE International Conference on Cloud Engineering (IC2E '21)*. IEEE, 2021, pp. 1–8

1.2.1 Automatic Benchmark Workload Generation

We enable an automatic and realistic benchmark workload generation for microservice applications using machine-generated service description files, i.e., open api specification files or also known as swagger files. These service descriptions can be automatically generated based on source code, thereby adapting automatically to changes in the code. We leverage this principle and generate benchmark loads for the respective application, which may also consist of multiple services, with minimal manual effort based on these description files: Assuming a REST-based microservice interface, developers describe the benchmark workload based on abstract interaction patterns. At runtime, our approach uses the respective interface description to automatically resolve and bind the workload patterns to the concrete endpoint before exe-

cuting the benchmark and collecting results. Our approach is not limited to a single service, but is also capable to resolve complex data dependencies across microservice endpoints.

1.2.2 Optimized Microbenchmark Suites

Optimized microbenchmark suites, which only include a small practically relevant subset of the full microbenchmark suite, can shorten the execution times of microbenchmarks drastically and enable a fast performance evaluation which can be embedded in regular CI/CD pipelines. To this end, we show how the practical relevance of microbenchmark suites can be improved and verified based on the application flow during an application benchmark run. We propose an approach to determine the overlap of common function calls between application and microbenchmarks, describe a heuristic which identifies redundant microbenchmarks, and present a recommendation algorithm which reveals relevant functions that are not covered by microbenchmarks yet. A microbenchmark suite optimized in this way can easily test all functions determined to be relevant by application benchmarks after every code change, thus, significantly reducing the risk of undetected performance problems. Through two use cases – removing redundancies in the microbenchmark suite and recommendation of yet uncovered functions – we decrease the total number of microbenchmarks and increase the practical relevance of both suites. It is, however, unclear whether microbenchmarks and application benchmarks detect the same performance problems and one can be a proxy for the other. To verify our approach, we thus explore whether microbenchmark suites can detect the same application performance changes as an application benchmark in real software development cycles. For this, we run extensive benchmark experiments with both the complete and the optimized microbenchmark suites of the two time-series database systems, for a commit history of 70 code changes for *InfluxDB* and 110 for *VictoriaMetrics*, and compare their results to the results of corresponding application benchmarks. Our results show that it is possible to detect application performance changes using an optimized microbenchmark suite if frequent false-positive alarms can be tolerated. By utilizing the differences and synergies of application benchmarks and microbenchmarks, our approach potentially enables effective software performance assurance with performance tests of multiple granularities.

1.2.3 A Benchmarking Framework for FaaS Environments

We propose BeFaaS, an extensible application-centric benchmarking framework for FaaS environments that focuses on the evaluation of FaaS platforms through realistic and typical examples of FaaS applications. BeFaaS includes four FaaS application-centric benchmarks reflecting typical FaaS use cases and currently supports commercial cloud FaaS platforms (AWS Lambda, Azure Functions, Google Cloud Functions) and the tinyFaaS edge serverless platform, and is extensible for additional workload profiles and platforms. The framework supports federated benchmark runs in which the benchmark application is distributed over

multiple FaaS platforms running on a mixture of cloud, edge, and fog nodes. Moreover, BeFaaS implements tracing features which collect fine-grained measurements which can be used for a detailed post-experiment drill-down analysis, e.g., to identify cold starts or other request-level effects. In our study using BeFaaS to compare different setups and FaaS providers, our experiment results show that (i) network transmission is a major contributor to response latency for function chains, (ii) this effect is exacerbated in hybrid edge-cloud deployments, (iii) the trigger delay between a published event and the start of the triggered function ranges from about 100ms for AWS Lambda to 800ms for Google Cloud Functions, and (iv) Azure Functions shows the best cold start behavior for our workloads.

1.3 Outline

This thesis contains five parts. The first part covers this introduction, a chapter on background information, and a chapter embedding the contributions in the related scientific work context.

The following parts II, III, and IV each present, evaluate, and discuss one proposed contribution.

Part II first presents the automatic benchmark workload generation based on abstract interaction patterns and service description files. The second chapter then evaluates the workload generation using a microservice-based application with several connected services. Lastly, the third chapter discusses advantages and limitations of the approach.

The first chapter in part III describes the algorithms to optimize microbenchmark suites based on call graph information from application benchmarks. Next, the second and third chapter each evaluate the optimization. The second by applying the optimizations to two open-source time series database systems to determine the potential improvements in execution time and code coverage. The third by using optimized suites in a simulated real CI/CD pipelines for multiple code changes to verify if the optimized microbenchmark suites can serve as a proxy for an application benchmark. The third part ends with a chapter discussing the findings and limitations of optimized microbenchmark suites.

Part IV starts with a detailed presentation of BeFaaS, our extensible application-centric benchmarking framework for FaaS environments and its features. Chapter two then describes the study design of our experiments using BeFaaS and presents its results. The third chapter then discusses the framework and addresses limitations.

The final part V concludes this thesis with a final discussion on benchmarks in CI/CD lines and the proposed approaches, a summary of the main findings, and an outlook on the topic.

Chapter 2

Background

Benchmarking aims to determine quality of service (QoS) by stressing a system under test (SUT) in a standardized way while observing its reactions. In contrast to monitoring, which is about non-intrusive and passive observation of a (production) system, benchmarking typically runs in a non-production environment and aims to answer how a system will react on specific changes or stresses, and is about comparison of system alternatives, system versions, configurations, or deployments. During a benchmark run, several scenario-specific metrics are measured which are then subsequently evaluated in an offline analysis. In order to derive valid findings, a benchmark design must comply with various general requirements such as fairness, portability, and repeatability [16, 24, 83, 59].

This chapter introduces the basic background information which is necessary for understanding the proposed concepts and their embedding in the scientific context. In this thesis, we deal with two different kinds of benchmarks outlined in the next two sections: application benchmarks, which evaluate complete (microservice) applications, and microbenchmarks, which evaluate individual functions or methods of an application or microservice. Moreover, as we argue and motivate to include a continuous benchmark step in existing build pipelines for software to ensure non-functional properties. We cover related concepts for this step in section three. Next, we outline the approach-specific related background in the following sections in chronological order: Section four introduces microservice applications, REST interfaces, and its interface descriptions, related to the benchmark workload generation. Section five outlines software call graphs and its analysis, related to the optimization of microbenchmark suites. Finally, section 6 summarizes the FaaS concept and the basic characteristics of FaaS platforms.

2.1 Application Benchmark

Application benchmarks evaluate non-functional properties of an SUT by deploying the respective system and all related components in a production-like test or staging environment and stressing these with an artificial but realistic workload [24, 16]. Thus, application benchmarks are often seen as the gold standard, because they evaluate the respective systems using a realistic load in the actual runtime environment and, depending on the use case, also using specific load scenarios (e.g., increased visits and checkouts during the Christmas season). A well-designed application benchmark can provide answers to many performance-related questions and also can be used to compare different versions of an SUT. This is especially relevant for the context of this thesis, in which a dedicated benchmark step as part of a CI/CD pipeline is envisioned [69, 160]. On the other hand, however, continuous benchmarking for early performance regression detection using an application benchmark is expensive, complex, and time-consuming [33, 152]. Besides the setup and configuration of all relevant components, which can already take a considerable amount of time, all experiments have to run for a certain time and usually have to be executed several times to get reliable results, especially in cloud environments [102].

During the design phase, it is necessary to think in detail about the specific requirements of the application benchmark and its objectives. While defining (and generating) the workload, many aspects must be taken into account to ensure that the requirements of the benchmark are not violated and to guarantee a relevant result later on [83, 16, 24, 59]. This is especially difficult in dynamic cloud environments, because it is hard to reproduce results due to performance variations inherent in cloud systems, random fluctuations, and other cloud-specific characteristics [107, 52, 59, 134]. To set up an SUT, all components have to be defined and initialized first. This can be done with the assistance of automation tools (e.g., [77, 76]). However, automation tools still have to be configured first, which further complicates the setup of application benchmarks. During the benchmark run, all components have to be monitored to ensure that there is no bottleneck inside the benchmarking system, e.g., to avoid quantifying the resources of the benchmarking client's machine instead of the maximum throughput of the SUT. Finally, the collected data needs to be transformed into relevant insights, usually in a subsequent offline analysis [16]. Together, these factors imply that a really *continuous* application benchmarking, e.g., applied to every code change, will usually be prohibitively expensive in terms of time but also in monetary cost.

2.2 Microbenchmarks

Instead of benchmarking the entire SUT at once, microbenchmarks focus on benchmarking small code fragments, e.g., single functions [1]. Here, only individual critical or often used functions are benchmarked on a smaller scale (hundreds of invocations) to ensure that there is no performance drop introduced with a code change or to estimate rough function-level metrics, e.g., average execution duration or throughput. Instead of (possibly) compiling, starting and configuring various components, it is usually enough to compile the corresponding code files and start the microbenchmark suite with the respective configuration. Similar to unit tests, they can even be run inside the local development environment. Microbenchmarks are thus usually easier to set up and to execute as there is no complex SUT which needs to be initialized first and a single microbenchmark takes considerably less time than the execution of an application benchmark.

Microbenchmarks are more suitable for frequent use in CI/CD pipelines but also have to cope with variability in cloud environments [105, 101, 102, 27]. Moreover, they cannot cover all aspects of an application benchmark and are, depending on the concrete use-case, usually considered less relevant individually because it is unclear whether they cover relevant parts of the production system or if detected performance changes will affect the production system [79].

Microbenchmarks are usually defined in only a few lines of code and evaluate an SUT on function level by calling the respective function under test repeatedly with artificial parameter values for a specified duration and a specified number of iterations. Multiple microbenchmarks together form a microbenchmark suite, which is usually executed several times in a row to measure execution durations at different times and thus get reliable results. To average out the effects of random fluctuations in the cloud environment, microbenchmark suites usually follow the Randomized Multiple Interleaved Trials (RMIT) execution order [2, 1].

2.3 Continuous Benchmarking

Continuous Integration (CI) and Continuous Deployment (CD) are two modern paradigms which aim to improve, automate, and accelerate the software development process leading to shorter release cycles. CI defines the process of integrating new software changes into the master version, including adapting and running corresponding test cases which ensures that the software is extensively tested before it is merged into a production branch of a system [63]. CD describes the automated process of releasing and deploying new software versions. Once a new release has been thoroughly tested in the CI process, it is automatically rolled-out to the production system so that frequent daily releases are possible; this shortens the release

[1] We use the term *function* to refer to any form of subroutine, no matter how they are called in the respective programming language.

cycle. Both processes, CI and CD, are designed to run multiple times per day, depending on how many features are implemented per day and the release policy. Thus, 10 minutes are a guiding value for the total run time of both processes so that developers can get early feedback on their software changes. In practice, however, this is not always realistic so that multi-tiered deployment pipelines are usually used instead. Here, after a first integration stage with integration and component tests, the second level with long running tests is not always executed. Often, it is run over night or directly before releasing a new software version. Finally, the test environment should be as close to the production environment as possible.

Continuous Benchmarking aims to not only verify functional properties in CI/CD lines, but also to ensure non-functional properties such as performance before releasing a new software version. Because both application benchmarks and the execution of a microbenchmark suite can often take several hours, this step is more suitable for nightly CI/CD pipelines. To avoid maintaining the required hardware and infrastructure for the benchmarks, renting it from cloud providers as needed and executing the benchmarks in cloud environments is a good option, which also provides a realistic environment as software nowadays often runs in the cloud. Strong, ongoing and random performance fluctuations, which are typical in cloud environments, however, make it difficult to execute relevant benchmarks, hence various aspects need to be considered to derive reliable findings. On the one hand, experiments must be repeated several times and for a reasonable duration. On the other hand, certain techniques such as RMIT execution for microbenchmark suites or the use of duet benchmarking for application benchmarks can help to obtain more meaningful results.

2.4 Microservice Interfaces and Descriptions

Lewis and Fowler [108] describe microservices as independently deployable and scalable components. In contrast to a monolithic system which combines all application logic in a single artifact, the microservice architecture splits the logic into a suite of services that communicate with one another over network. This separation allows parts of an application (i.e., individual services) to be evolved and operated (e.g., horizontally scaled) irrespective of one another. Within an application, individual services[2] can be written in different programming languages or use different storage technologies, resulting in a heterogeneous environment. Being separate deployment units, individual services can independently be shut down, replaced or updated at will, or new service instances can be deployed at runtime to counteract performance bottlenecks. Given these characteristics, all services must be designed to tolerate failures, as no service can expect correctly typed data or assume that a required service is always available. A challenge for microservice architectures is the lack of debugging and logging capabilities, especially in complex setups including a multitude of services.

[2]In the following, we will refer to microservices as either 'services' or 'microservices' interchangeably.

Microservices communicate over the network, relying on networked application programming interfaces (APIs). APIs can differ in the communication protocols (e.g., TCP, HTTP) and data formats (e.g., JSON, binary data, XML) they rely on. In this thesis, we focus on APIs following the REST architectural style. Being heavily inspired by HTTP, REST APIs evolve around resources being identified by hierarchical URLs, and use HTTP methods to interact with these resources (e.g., POST to create one or GET to receive one). REST APIs do not rely on client state (stateless), and evolve around the communication of resource representations (typically in JSON or XML) between clients and servers [136].

Richardson's maturity model [62] divides REST APIs into three levels: While level 0 APIs use HTTP only to tunnel requests to an endpoint, level 1 introduces resources which can be addressed following hierarchical URIs. Level 2 additionally demands that APIs use HTTP verbs to indicate whether to create (POST), get (GET), update (PUT or PATCH), or delete (DELETE) a resource. Finally, level 3 inserts links (URIs) to corresponding services and or resources into the server responses at runtime, realizing *RESTful* APIs. In this thesis, we assume APIs to comply at least with level 2 of this maturity model – specifically, we rely on the use of HTTP methods for defining abstract operations.

In addition to human-readable API documentation targeting (client) developers, REST APIs are often described in a machine-understandable way using description files such as OpenAPI [3] or RAML [4]. For the sake of simplicity, we decided to only consider OpenAPI in our work as possible interface contract.[5] OpenAPI files are written in YAML or JSON and describe where to reach an API, available operations, its expected inputs, and possible outputs. Although the current version, OpenAPI 3.0, supports so-called link definitions to express relationships between two requests, they are not designed to describe complex interaction patterns which we develop and present in this thesis.

2.5 Software Call Graphs

Software source code in an object-oriented system is organized in classes and functions. At runtime, executed functions call other classes and functions, which leads to a program flow that can be depicted as a call graph. This graph represents which functions call which other functions and adds additional meta information such as the duration of the executed function. Regardless of whether software is evaluated by an application benchmark or microbenchmark, both types evaluate the same source code and algorithms. Since an application benchmark is designed to simulate realistic operations in a production-near environment, it can reasonably be assumed that it can serve as a baseline or reference execution to quantify relevance in

[3] `https://swagger.io/docs/specification/about/`
[4] `https://github.com/raml-org/raml-spec/`
[5] Translations between formats are possible, using for example `https://apimatic.io/transformer`.

the absence of a real production trace. On the other hand, microbenchmarks are written to check the performance of individual functions and multiple microbenchmarks are bundled as a microbenchmark suite. If these graphs are available for an application benchmark and the respective microbenchmark suite, it is possible to compare and analyze the flow of both graphs. In this thesis, we use this relation between both benchmark types to optimize microbenchmark suites, among other things.

2.6 FaaS Platforms

All major cloud providers offer Function-as-a-Service (FaaS) solutions where users only have to take care of their source code (functions) while the underlying infrastructure and environment are abstracted away by the provider. FaaS applications are composed of individual functions which act microservices and are deployed on a FaaS platform that handles, e.g., the execution and automatic scaling. Developers do not have direct control of the infrastructure and can only define high-level parameters, such as the region in which the function should run [26]. Due to this, FaaS platforms are easy to use but comparing cloud platform performance [106, 18] is challenging, as the cloud variability is further compounded by an additional, unknown infrastructure component.

Chapter 3

Related Work

Related work in this area deals with the requirements for benchmarks in general, application-specific characteristics, and more effective benchmark execution. Furthermore, contributors expand on the analysis of problems and examine the influence of environmental factors on the benchmark run in more detail.

Current work focuses on application-specific benchmarks. Our approach can use all of these application benchmarks as a baseline. As long as a call graph can be generated from respective SUT during the benchmark run, this graph can serve as input for our approach.

In the following, we discuss related work to our study focusing on benchmarking in CI/CD pipelines, approaches to reducing the overall benchmark execution time, dealing with cloud variability, and approaches for detecting and quantifying performance changes. To the best of our knowledge, we are the first who use optimized microbenchmark suites as proxy for application benchmarks.

Existing research on benchmarking of FaaS environments has so far mostly focused on micro-benchmarks. Application-centric benchmarks that consider the overall performance of multiple functions, the interaction with external services, and the effects of different application load profiles are mostly still missing.

Beyond FaaS, there are a number of application-centric benchmarking frameworks in other domains, e.g., for database and storage systems [24, 52] or for virtual machines [29]. These can, however, not easily be adapted to FaaS platforms.

3.1 Benchmarking in Cloud Environments

Benchmarking is a well-established method in the IT domain to quantify and verify quality of service of hardware or software systems [16]. There are many benchmarks for different kinds of SUT, especially for database and storage systems, e.g., [129, 52, 24, 122, 95], but also for

virtual machines, e.g., [29, 144], web APIs [20, 21], or cloud-based queuing systems, e.g., [96]. Regardless of the type of SUT and kind of benchmark, all benchmarks should aim for design goals such as relevance, portability, or repeatability to provide reliable results [89, 22, 59, 83, 16].

A key requirement of benchmarks, the repeatability, is difficult to realize in variable cloud environments due to the many random factors that affect the benchmark [59, 152, 134, 36, 28, 32, 102, 101, 141, 97, 105, 1, 52, 86, 106, 158, 18]. These studies and approaches are relevant for the application of our presented contributions. If the variance in the test environment is known and or can be reduced to a minimum, application engineers are able to decide better and on a sound basis at what time and to which extent which benchmark type should be executed.

Despite the variability, to ensure that measurements are as accurate as possible and to derive correct conclusions, benchmarking experiments must be conducted several times to ensure their repeatability. One way to minimize the effects of this variability and the number of experiment repetitions is to benchmark multiple SUTs concurrently on the same VM(s) [32, 36]. This is, however, not always easy to implement or even possible. In more complex systems that include several components distributed on different instances, for example, it would be necessary to ensure that the individual components are also exposed to the same load concurrently to provide a valid application benchmark. To the best of our knowledge, we are the first who use and apply the Duet Benchmarking technique proposed by [32] in longer running application benchmarks to counteract random cloud variability and to provide repeatable results.

Other approaches aim to reduce the execution time for application benchmarks by stopping the benchmark run when the system reaches a repetitive performance state [6, 4, 5] or use a statistical approach based on kernel density estimation to stop once a benchmark is unlikely to produce a different result with more repetitions [78]. Such approaches can only be combined with our contributions and optimizations for workload generation or microbenchmark suites under certain conditions. The main aspect here are rarely called functions which might never be called if the benchmark run is terminated early. [41] use the functional tests of software projects and extract classifiers for predicting tests that will reveal performance changes. Overall, this reduces the testing time drastically and the approach is also able to detect real performance issues in production. Nevertheless, the authors use a partly automatic, partly manual performance analysis based on reported issues for their SUTs only and the number of application-relevant performance changes thus might be underestimated.

3.2 Continuous Benchmarking

The idea of using performance testing or benchmarking in CI/CD pipelines, also in cloud environments, has already been addressed in several related papers. [120] argue that application performance after new commits should be tracked and propose an automatic approach based on call trees. [60] manually inject performance issues in three study objects to verify their automatic performance regression detection approach [60, 61]. [160] include microbenchmarks in a CI/CD pipeline. Moreover, several studies also conduct performance case studies using a dedicated benchmarking step and real software projects [69, 48, 85, 47]. [88] propose a CI/CD tool chain considering performance tests. [152] and [76] propose frameworks supporting the automatic execution of benchmark experiments in the cloud as part of CI/CD pipelines. Our contribution on the optimization of microbenchmark suites continues this research by studying, through extensive experimentation, to which degree different benchmarking approaches can detect performance changes of open source systems as part of a CI/CD pipeline.

Continuous Benchmarking is a powerful mechanism for evaluating QoS of a new system version in a production-like environment. As such, it is relies on benchmarking approaches such as [28, 52, 20, 29, 24, 22, 43]. An alternative but also complementary approach to CB are live testing techniques such as canary releases [82] or dark launches [57]. In contrast to CB, live testing is characterized by the fact that a new version (of a software artifact) is directly deployed into the production environment in parallel with the older version.

For canary releases [82], this new version is initially rolled out for a very small subset of users and developers monitor its behavior in production. If there are errors or QoS issues in the new version, the impact only affects a few users and the version is reverted or shut down. Otherwise, more and more users are added to the set of test users until the new version has completely been rolled out. While canary releases aim to only affect a small subset of users in case of failures, dark (or shadow) launches [57, 155] eliminate potentially unsatisfied users completely by deploying a new version in the production environment without serving real user traffic – so called shadow instances. This way, no user is confronted with the new version and its potential issues.

Live testing techniques can be used to detect performance and other QoS issues in production. However, testing new versions in a production environment might be problematic for several reasons: First, a production system is usually in a normal state with usual load and regular traffic. Thus, a new version is never evaluated in production under extreme conditions or for rare corner cases. Second, a roll-out of several new versions of multiple software artifacts is administratively complex and error-prone, though tools like BiFrost [143] try to overcome these problems. Third, theoretical setups and architectures including new versions are hard to evaluate with live testing techniques. Finally, live testing does not necessarily create the right data to identify QoS degradation in the system release as varying workloads depending on user

traffic will lead to varying observable QoS behavior. All this can be done with Continuous Benchmarking, e.g., by creating benchmark setups for extreme load peak situations. As benchmarking, however, can never be identical to a production load, we propose to combine the strengths of both approaches, i.e., to use both live testing and Continuous Benchmarking in parallel.

3.3 Detecting and Quantifying Performance Changes

There are several studies that aim to identify (the root cause of) performance regressions [123, 61, 48, 69, 160] or examine the influence of environment factors on the system under test, such as the usage of Docker [71].

We are, to the best of our knowledge, the first who apply a dynamically adapted performance detection threshold which adjusts along with the analyzed code changes to the respective micro or application benchmark instability. Besides basic threshold metrics such as the ones we adapted from [69], there are more complex techniques for detecting and quantifying performance changes. [60] use performance signatures of past experiment runs and determine confidence measures. [48] also consider noise in their performance evaluation and cluster the time series experiment data to identify performance change points [117]. Moreover, even though the Iter8 framework proposed by [156] is designed for live testing, the proposed decision bayesian learning based algorithms can also be adapted to decide which version performs better. Each of these approaches could be used as alternatives for detecting performance changes and might, e.g., reduce the number of false alarms. On the other hand, however, each of these approaches also increases the complexity and implementation effort of the analysis.

Finally, there are approaches that focus on automatically identifying the respective root causes of performance changes [123, 79].

double check this: Another large body of research is performance regression testing, which utilizes microbenchmarks between two commits to decide whether and what to test for performance. [81] and [139, 140] utilize models to assess whether a code commit introduces a regression to select versions that should be tested for performance.

3.4 Benchmarking Microservices

There is, to the best of our knowledge, currently no approach for benchmarking microservices. We believe that this is largely due to the fact that microservices do not come with the common interface typical to other system domains such as CRUD interfaces for data management. Without such a common interface, it becomes quite hard to implement a benchmark that complies with standard benchmark requirements – especially portability [83, 59, 16, 22, 94]. Recent work in the microservice benchmarking domain presents general benchmark

requirements for microservices [3], focuses on the automation of performance tests of microservices [50], or implements a benchmark suite with six different microservice applications [64].

Nevertheless, there are some approaches and tools which can ease microservice benchmarking beyond building a complete benchmark from scratch: Load generators such as Artillery IO[1] or LoadUI[2] can run a defined and service-specific workload against a microservice. By manually defining scenarios which represent typical interactions, a service-specific workload can be created with parameters settings which include the amount of request or the distribution of scenarios. While it is possible to import service description files and external data items as "workload", this is always specific to a particular microservice and its respective version, i.e., there is no portability. Kao et al. [91] also generate requests based on (regularly updated) service descriptions, but focus on web services and manually define each test specification. With our workload generation approach, on the other side, arbitrary REST microservices can be benchmarked as long as the service supports the respective interaction patterns. Pattern definitions can be reused for benchmarking other microservices. Atlidakis et al. [11] analyze OpenAPI specifications and automatically generate functional tests to detect bugs and security vulnerabilities. Our approach, on the other side, focuses on non-functional requirements. Nevertheless, their algorithm to resolve dependencies by analyzing actual service responses can be used to identify dependencies between services automatically, which can ease our binding definition step.

Zheng et al. [167] also use interaction patterns comprised of basic operations (create, get, delete) for benchmarking but do so for object storage services. Their approach relies on the standard interface defined by CDMI and, hence, does not have to deal with interface heterogeneity. Beyond these, there are several systems which could be used as a load generator. Benchmarking systems such as YCSB [43] or NDBench [128] can be used to create synthetic workloads against a CRUD endpoint. While these tools are very powerful load generators – particularly when considering the broad range of configuration options – they completely disregard the mapping from the generic CRUD to a specific microservice. Although creating such a mapping will be possible for a large percentage of microservices, actually programming the mapping still remains a manual effort that needs to be repeated for every microservice and version that shall be benchmarked. Furthermore, we believe that benchmarking interaction with microservices should preferably be based on sequences of operations instead of isolated operations to get more realistic results (systems such as YCSB+T [51] or BenchFoundry [24] are probably a better fit).

Besides workload generation and invocation of REST endpoints, our approach generates synthetic data for the workload. For data generation, we rely on JSON schema and the faker.js library. Approaches such as [134] are more powerful options for data generation and also

[1] `https://artillery.io/`
[2] `https://www.soapui.org/professional/loadui-pro.html`

support parallel generation. Such parallelization could improve our prototype in which generating the workload trace prior to distributing it onto the Worker Nodes can be rather slow. Nevertheless, we do not see parallelization as a critical feature since the generated workload can be persisted and reused instead of being generated from scratch for every benchmark run.

Finally, an alternative for clients of a microservice can be to rely on SLAs while monitoring violations, e.g., [92, 109]. This approach, however, only shifts the responsibility for ensuring microservice performance to another organizational entity and does not actually solve the challenge of detecting performance changes of microservices early on, ideally as part of a CI/CD pipeline [69, 160, 48].

3.4.1 Optimization of Microbenchmarks

[105] and [154] empirically studied how microbenchmarks – sometimes also referred to as performance unit tests – are used in open-source Java projects and found that adoption is still limited. Others focused on creating performance-awareness through documentation [80] and removing the need for statistical knowledge through simple hypothesis-style, logical annotations [34, 35]. [40] characterize code changes that introduce performance regressions and show that microbenchmarks are sensitive to performance changes. [49] study bad practices and anti-patterns in microbenchmark implementations. All these studies are complementary to ours as they focus on different aspects of microbenchmarking that is neither related to time reduction nor recommending functions as benchmark targets.

In this thesis, we propose and follow an optimization strategy that combines application and microbenchmarks to detect redundancies and optimize microbenchmark suites. Similar to us, [101] apply a mutation-testing-inspired technique to dynamically assess redundant benchmarks and detect redundancies between microbenchmarks of the same suite. [124] and [7] decide based on source code indicators and information from prior benchmark runs which microbenchmarks to execute on every commit. These approaches, however, treat every code section equally and does not favor practically relevant code, i.e., functions that are actually used in production. [53] study the usability of functional unit tests for performance testing and build a machine learning model to classify whether a unit test lends itself to performance testing. Our redundancy removal approach could augment their approach by filtering out unit tests (for performance) that lie on the hot path of an application benchmark.

Our optimization strategy can be combined or replaced with other microbenchmark prioritization strategies to reduce the number of false alarms (e.g., [121, 100]) or to further shorten the execution duration by stopping microbenchmarks once the results are stable and or do not show significant performance changes (e.g., [6, 4, 78, 103]). One might also execute the microbenchmarks in parallel on cloud infrastructure to further shorten the benchmark dura-

tion. Recent work studied how and to which degree such an unreliable environment can be used [102, 36].

Finally, synthesizing microbenchmarks could be a way to increase coverage of important parts of an application. These could, for instance, be identified by an application benchmark. SpeedGun generates microbenchmarks for concurrent classes to expose concurrency-related performance bugs [133]; and AutoJMH randomly generates microbenchmark workloads based on forward slicing and control flow graphs [137]. Both approaches are highly related to our microbenchmark suite optimization strategy as they propose solutions for not yet existing benchmarks. However, both require as input a class or a segment that shall be performance tested. Our recommendation algorithm could provide this input.

3.5 FaaS Benchmarking

Existing work on benchmarking of FaaS platforms usually focuses on the execution of small, isolated microbenchmarks that deploy and call a single function, e.g., a matrix multiplication or a random number generator. These functions are often designed for a specific purpose, e.g., to stress the CPU of the test system or to evaluate the test system with a disk-intensive workload [12, 114, 161, 58, 115, 149, 104, 162, 116]. Besides scaling of functions, cold start latency, containerization overheads, and instance lifetimes, the studies also evaluate metrics such as CPU utilization, network throughput, and costs. Since the publication of the initial version of BeFaaS [73], recent research work has focused on benchmarking function triggers [147], studying tail latency [157], implications of used programming languages [46], hardware influence factors [45], concurrent function executions [13], take a closer look at the cost perspective [132], performance fluctuations over time [148], fine-grained tracing of requests [146, 145], and general FaaS characteristics [55]. Almost all experiments, however, focus on a single isolated aspects and do not create a holistic comparability of platforms performance for FaaS application developers.

Several studies also consider more complex applications and focus on specific FaaS related features, e.g., by deploying image processing pipelines [93], analyzing chained functions, or deploying real world applications on serverless platforms [162]. While the authors of these studies also use application-centric workloads for experiments, their goal was not to propose a comprehensive framework for the execution of application-centric FaaS benchmarks. Further, there are several studies and frameworks that share some of the features and goals of BeFaaS: *PanOpticon* [153] uses a deployment, workload, and metrics module to evaluate chained functions and a simple chat server on two different FaaS vendors. Although PanOpticon has similar goals as BeFaaS, it neither supports detailed drill-down analysis nor federated multi-provider setups. Van Eyk et al. [56] develop a high-level architecture and state requirements for serverless benchmarking. *FaaSdom* [113] shares our motivation for a full application de-

ployment. It supports multiple platforms, several languages (e.g., Node.js, Python, Go), and an automatic deployment of performance tests via a web frontend. *SeBS* [44] is a FaaS benchmarking framework that highlights the cost efficiency of executions and, similar to FaaSdom, also only considers single-function applications.

Besides a library which supports multi-cloud setups [166], to the best of our knowledge, BeFaaS is still the only FaaS benchmarking framework for evaluating federated cross-provider setups which can also be used to trace request in the edge to cloud continuum.

Part II

Continuous Benchmarking

Chapter 4

Jenkins Plugin

4.1 Introduction

Today's IT systems tend to be rather complex pieces of software so that even the smallest changes, either in the source code itself or in the application settings, can have a big (negative) impact on their performance, e. g., adding or configuring security features as shown in [122, 127, 126]. Besides increased latencies which result in a poor user experience, cloud based systems usually use autoscaling to automatically adapt the amount of resources to meet quality of service (QoS) goals. When a software change now leads to increased use of hardware resources, more resources will be provisioned leading to significantly higher cost. Finally, systems and services rarely operate in an isolated way. In interaction with other systems and services, however, small QoS changes will affect other services and may even start a butterfly effect in some scenarios. Regardless of the cause, these deficits are often coupled with reduced revenue or even fines if the current performance metrics do not meet the defined service level agreements (SLA) or user expectations, e.g., Google reports that the number of daily searches per user decreases if the latency of results increases [31].

In order to prevent undesired effects on performance or other quality metrics, we propose to add system benchmarking to the build pipeline of software systems. This way, developers can assert that a new release is at least as good as the previous release and that it complies with SLAs. In this regard, we make the following contributions:

1. We describe how QoS requirements can be integrated into the development process and how benchmarking can be used to enforce QoS goals.

2. We present a proof-of-concept prototype, including the corresponding Jenkins plug-in.

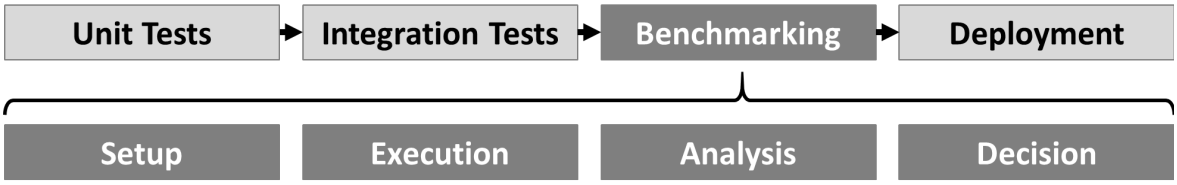


Figure 4.1: Main Steps of Continuous Benchmarking

4.2 Approach

In this section, we describe how benchmarking can be used as part of a CI or CD build process to ensure QoS requirements, give an overview of our system architecture, and describe how builds with QoS problems can be detected in benchmarking results.

4.2.1 Continuous Benchmarking

As already described, state of the art CI and CD solutions focus on functional tests and micro benchmarks such as single method performance. To complement this, we propose to regularly run system benchmarks as part of the build process. Comparable to CI and CD, we refer to this as Continuous Benchmarking (CB).

CB should only be done if the correct functionality of the software has already been ensured, otherwise the benchmark might run against a buggy software version and produce incorrect results, e.g., because data records are processed much faster due to an error. CB should, hence, take place once all functional tests and integration tests have already been passed. In the following, we will give an overview of the steps involved in CB; see also figure 4.1 which gives a high level overview of the CB process and how it integrates into a CI/CD process.

Setup: First, once all functional and integration tests have been passed, the SUT and the benchmarking client must be set up, which can be done either sequentially or in parallel. To get comparable results over several runs, it is important to deploy both systems in the same environment in each CB process (same hardware, operating system, supporting libraries, etc.). If, for example, a different hard disk would be used in each run, the different speed would have an influence on the results and it would not be possible to carry out a trend analysis of the key figures. Also, unless the benchmark explicitly targets a future use case, it is reasonable to choose a runtime environment as similar as possible to the production environment. Finally, the SUT and benchmarking system must be isolated from external factors which could affect the benchmark results, e.g., other processes running on the same machine, other services interacting with the SUT, or too much traffic on the network.

Depending on the system under test, it may also be necessary to deploy other external systems (e.g., BigTable [39] always relies on GFS [67] and Chubby [37]) or to run a preload phase which inserts an initial data set into the SUT [24].

Execution: In the second step, the benchmark is actually run. In fact, it may be run several times as benchmarks should usually be repeated and different benchmarks and benchmark configurations may be run in parallel. Here, monitoring should be used to assert that the machine(s) of the benchmarking client do(es) not become the performance bottleneck. Furthermore, benchmarks should be run for a sufficiently long time, typically, this means to keep a system benchmark running for at least 20-30 minutes [16].

Analysis: In the third step, the results from all benchmark runs need to be collected as they will typically be distributed across multiple machines. Next, these results need to be analyzed. Depending on the benchmark, this may mean simple unit conversions (e.g., ns to ms) and aggregations or more complex analysis steps. For instance, when data staleness is measured following the approach of [17], this may involve analysis of several GBs of raw text files.

Decision: Finally, the process needs to decide whether the current build is released to the deployment pipeline or whether the process is aborted because QoS goals were not met. Depending on the application and its benchmark, the measured values can either be compared with absolute thresholds, e.g., as defined in Service Level Agreements (SLA) or software specifications, or relative thresholds could be used (we discuss these metrics section 4.2.3).

4.2.2 Architecture and Components

As shown in figure 4.2, the components in our architecture are closely aligned with the steps described above. Typically, our CB process will be triggered by a CI server or some other build pipeline automation system. For this, the Benchmark Manager acts as the main entry point. Once it has been triggered, it installs and configures both the SUT and the Benchmarking Client before starting the execution of the benchmark run(s). Next, the Benchmark Manager collects all results and forwards them to the Analyzer which is responsible for all analysis steps. Of course, the Analyzer may also act as a proxy that forwards the raw results to an external analysis system – like the Benchmarking Client, the Analyzer is SUT-specific. Finally, the Analyzer forwards the aggregated analysis results to the CB Controller along with the raw data. The CB Controller then persists the data, decides on success or failure of the evaluated build, and reports the result back to the CI server. Beyond this, the Visual Interface visualizes the benchmarking results for human users and is also used to configure the CB Controller (e.g., to adjust relative and absolute thresholds).

4.2.3 Metrics

The final step of our approach requires some metrics to decide on the success or failure of a benchmark run. Depending on the system requirements, these decision can be based on fixed values given in SLAs, can reject a build because of a sudden and significant drop/jump in

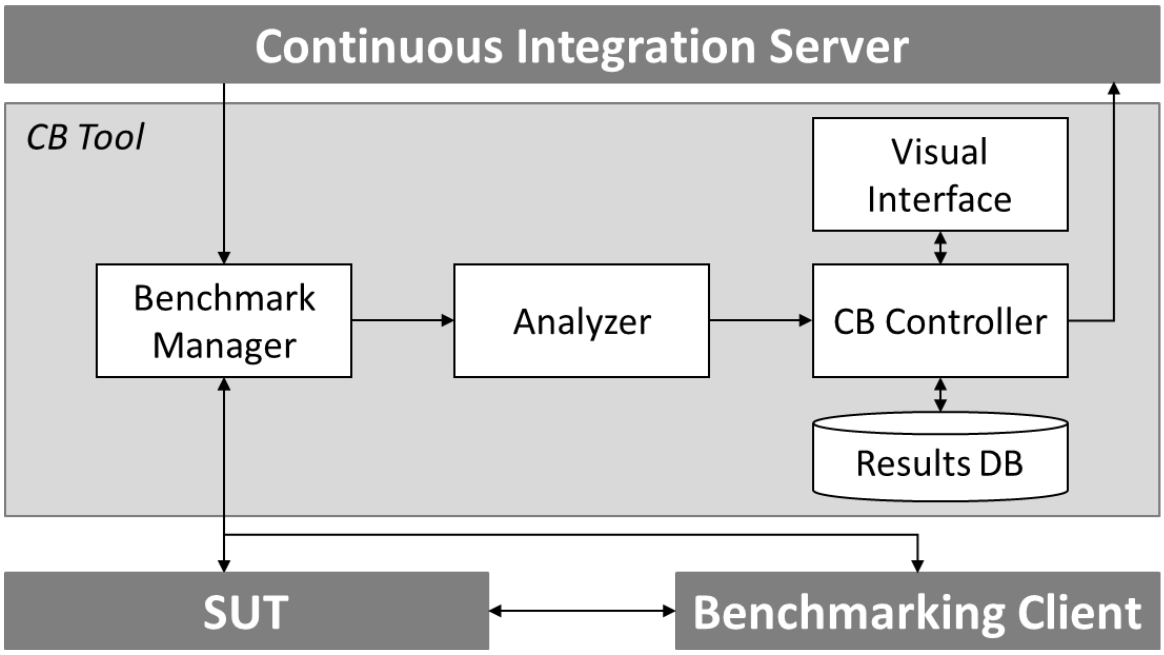


Figure 4.2: Architecture and Main Components of Continuous Benchmarking Setups

QoS compared to the last build, or detect a negative trend over multiple builds. Here, we present the decision algorithms which we will later use in our evaluation.

Fixed values (FV): The most simple method to detect undesired builds is to apply fixed thresholds, e.g., from SLAs. A build is rejected if the determined metric m_c, e.g. latency, is not in a specific interval.

$$f(m) = \begin{cases} succeed & \text{if } FV_{lower} < m_c < FV_{upper} \\ reject & \text{else} \end{cases} \tag{4.1}$$

Please, note that we always consider a lower and an upper value as thresholds. As an example, consider an application with a latency of 10ms. If this latency suddenly drops to 1ms, it could be the result of brilliant engineering. It is, however, much more likely to be the result of a bug where all requests terminate really quick with an error message.

Jump detection (JD): Especially if a software system offers much better quality than required by the FV thresholds, a sudden massive change in quality may still have a significant impact on the user experience. Thus, a relative comparison of the current run metric m_c to the predecessor run metric m_{c-1} can be used to reject builds in which QoS deviates from the last build by more than t percent. Concrete values for t obviously depend on the concrete application; however, we recommend values around 5%.

$$f(m_c, m_{c-1}) = \begin{cases} succeed & \text{if } t > 100 \left(\frac{m_c}{m_{c-1}} - 1\right) \\ reject & \text{else} \end{cases} \tag{4.2}$$

Trend detection (TD): Finally, a longer lasting trend can lead to the software deteriorating slightly in b builds and finally exceed a given relative threshold of t percent in total. Here, the metric of current build m_c must not exceed t percent more than the moving average of the previous b builds (m_{c-b} refers to the metric of the bth build before the current one).

$$f(m, b) = \begin{cases} succeed & \text{if } t > 100 * \left(\frac{m_c \cdot b}{\sum_{i=1}^{b} m_{c-i}} - 1\right) \\ reject & \text{else} \end{cases} \tag{4.3}$$

Besides these, there are other metrics which could be used. A valid approach might actually be to set the t value in JD and TD to zero to enforce continuously improving QoS. This, however, is likely to reject most builds unless the experiments are run on a completely isolated infrastructure.

4.3 Evaluation

In this section, we evaluate our approach through a proof-of-concept prototype and a number of experiments with our proposed Continuous Benchmarking process in a realistic setup. We decided to use the existing commit history of Apache Cassandra as it is one of the most popular NoSQL systems and benchmark coverage, e.g., through the well established YCSB benchmark, is good. For our experiments, we replayed the commit history of Cassandra over the last two years.

4.3.1 Proof-of-Concept Implementation

We have implemented our system design as a proof-of-concept prototype. Parts of it are generic enough to be useful for all use cases, other parts are very use case-specific. Specifically, the Benchmark Manager's code depends to some degree on the SUT and the benchmarking client used. Here, we implemented everything as needed for our evaluation (see next section) with Apache Cassandra and YCSB.

The Benchmark Manager is implemented in Java and uses a number of Unix shell scripts for installation of Git, Ant, etc. if not already installed. For a production-ready implementation, we would recommend to replace such shell scripts with "Infrastructure as Code" environments such as Ansible[1].

[1] https://www.ansible.com/

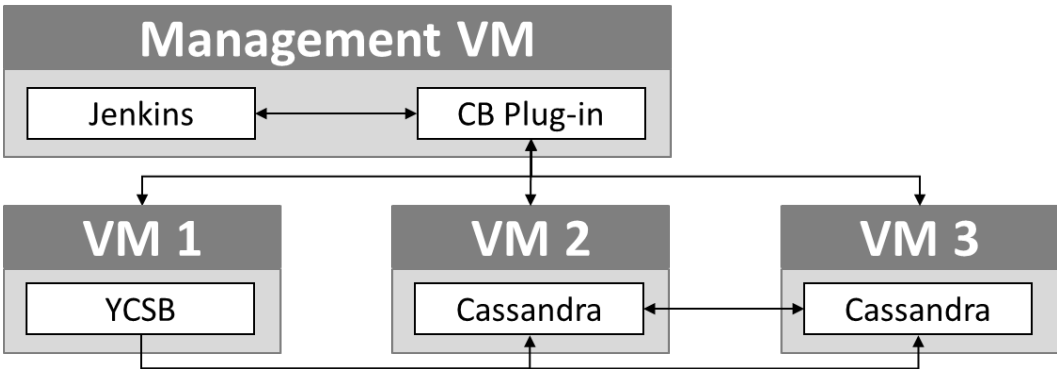


Figure 4.3: Setup in all Experiment Runs

As already indicated above, the Analyzer is SUT- and benchmark-specific. In our evaluation case, it is implemented as a very short script that converts the YCSB output files to a standard format that our CB Controller can understand.

To better integrate our prototype into existing build pipelines, we have implemented the CB Controller and the Visual Interface as a Jenkins plug-in[2]. The Visual Interface allows users to specify thresholds and illustrates line charts for trend analysis as well as functionality for detailed insights into single benchmark runs. In contrast to our Benchmark Manager which is aligned with our experiments, the plug-in is generally applicable to arbitrary metrics.

4.3.2 Experiment Setup

For our experiments, we deployed Jenkins and our CB plug-in on a single virtual machine (VM). We configured the plug-in to run Cassandra on two other machines and YCSB on a third machine (see figure 4.3).

In all experiments, Cassandra used the "SimpleStrategy" for replication as we only had two nodes in the cluster; the replication factor was two. YCSB used workload A with the following configuration: fieldcount=10, fieldlength=100, records=20,000, operations=1,000,000 and threads=100.

We ran our set of experiments on Amazon EC2 m3.medium instances (3.75GB RAM, one CPU core) in the eu-west region, all in the same availability zone. We used the Amazon Linux AMI and ran all experiments on the same three VMs.

As input for our experiments, we used 465 commits between Jan 3, 2017 and Oct 23, 2018 of Cassandra's commit history which merged changes into the main trunk. We tested this reduced commit history three times successively, thus, each of these commits was benchmarked three times at different points in time, i.e., we had almost 1400 benchmark runs.

[2]https://github.com/jenkinsci/benchmark-evaluator-plugin

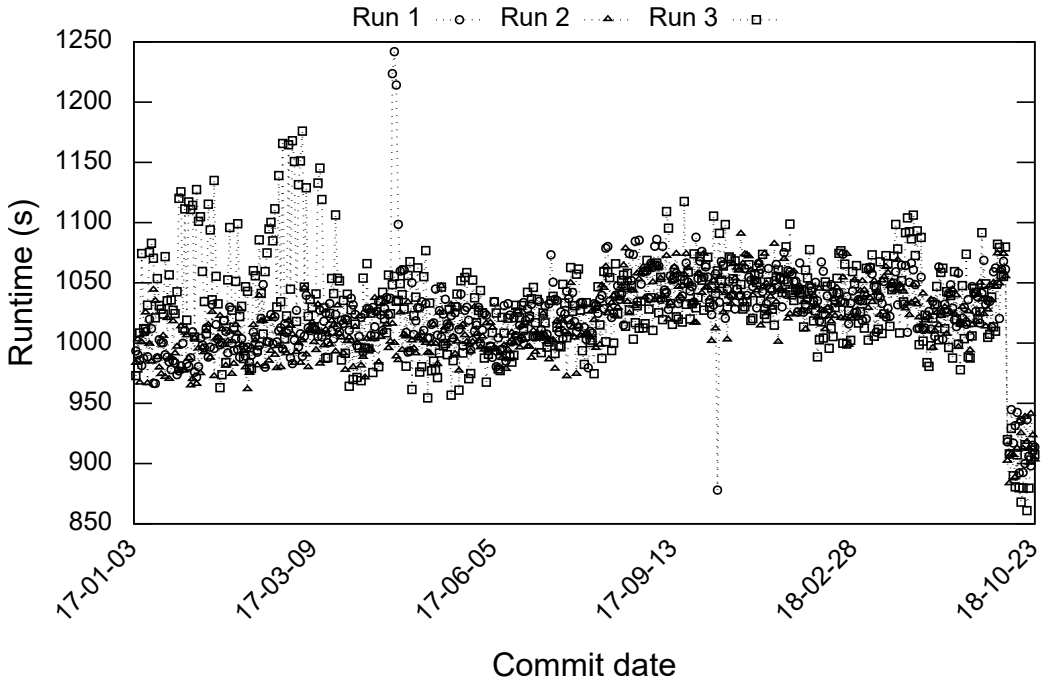


Figure 4.4: Total Benchmark Runtime

Please, note that a real build pipeline of course also involves steps such as testing. For our experiments, we decided to exclude these steps for reasons of simplicity.

4.3.3 Results

Figure 4.4 shows the results of our experiments as returned by YCSB. When ignoring outliers which can be expected when experimenting in the cloud [18], the values indicate the performance gradient as they are mostly densely packed.

At this stage, we did not specify any absolute or relative thresholds in our plug-in as we wanted the entire commit series to run through.

4.3.4 Application of Threshold Metrics

Following our approach and the metrics defined in 4.2.3, we applied these thresholds on the median benchmark measurement to exclude outliers but still evaluate with actual measurement values. For the total benchmark runtime, we set $950s$ and $1100s$ as FV thresholds. We chose $t = 5\%$ as relative threshold for JD and $t = 4\%$ for TD which includes $b = 20$ builds.

Figure 4.5 illustrates these graphs along with the results from our median experiment run. An intersection of the median line and one of the other lines means that the respective build will be rejected.

The fixed value boundaries trigger only once – for the sudden performance improvement (ca. 13.5%) towards the end of the timeseries. Here, the developers introduced two features: "Flush

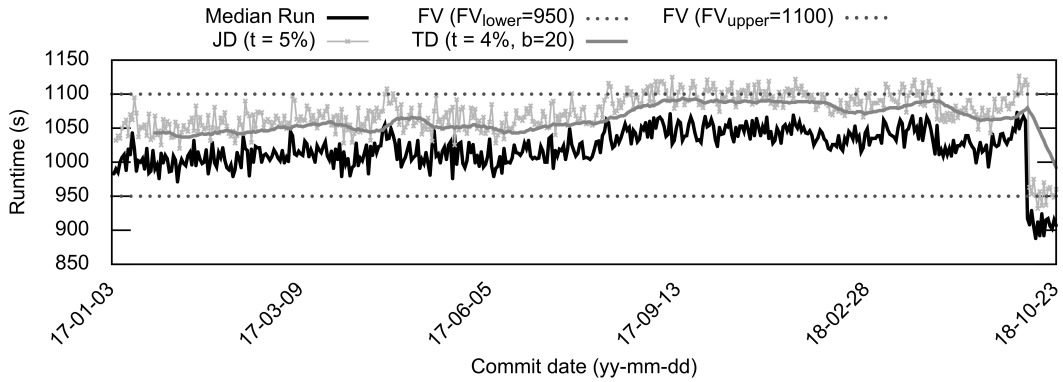


Figure 4.5: Total Benchmark Runtime: Median Run and Threshold Metrics

netty client messages immediately by default" and "Improve TokenMetaData cache populating performance avoid long locking" which indicates that our detection is a false positive. Our jump detection algorithm would reject one build on May 10, 2017 (jump around 6.5%) which can either be caused by, according to the Git commit messages, "Forbid unsupported creation of SASI indexes over partition key column" or "Avoid reading static row twice from legacy sstables"; the first one, however, seems more likely to be the cause. The trend detection, on the other side, would reject 4 builds. The most significant violating build was on Aug 2, 2018 (performance drop of almost 4.6%) which just moves some code comments without touching any functionality, thus, the main reason for this negative trend is caused in the builds before and further analysis is necessary.

We only applied our defined metrics to the total runtime. Of course, there would be more metrics in other, more complex scenarios.

4.4 Discussion

Based on our measurement results, we believe that CB is a very useful approach for keeping QoS of a system either constant or to continuously improve it while using the same amount of infrastructure resources. With our proof-of-concept prototype, we have also shown that the integration of CB into a build pipeline is indeed possible and does not involve a lot of effort – in fact, CB is simply integrated into the build process through our prototype which is automatically triggered for new versions. There is, however, also a number of open challenges and caveats.

Running CB will typically create additional costs for the development process; there is a tradeoff between how frequently CB is run, i.e., how early QoS problems can be detected, and the costs associated with that. We believe that this tradeoff is system-specific and cannot be solved in a general way. Developers also have to decide whether they plan to run CB on dedicated physical machines on-premises or whether they shift benchmark execution to the cloud. Depending on the frequency of CB runs, the on-premises option may be less expensive.

The cloud option, in contrast, allows to run several benchmarks (and benchmark runs) in parallel so that it will be the preferred option when CB uses a set of benchmarks instead of a single benchmark only.

This choice between on-premises non-virtualized hardware and the cloud option is also related to the variance of results: We believe that running the CB process on dedicated hardware will produce more stable results with less variance across experiment runs. When running experiments in the cloud, we would recommend to run an initial experiment with at least ten runs (more is better) to get a better understanding of the variance effects caused by the underlying infrastructure. This would then also determine the number of necessary repetitions during the actual CB execution. Based on our AWS results, we would recommend to run the experiment at least three times (preferably five times) and to use the median result for further analysis and decision making.

There is also the question of when to trigger a CB run. In fact, we do not believe that running one for every single Git commit is the ideal but too expensive scenario. In our opinion, this would simply create too much data so that the developers may no longer be able to visually comprehend results. Comparable to our evaluation approach, we would recommend to trigger CB whenever a new feature branch is merged into the main branch. This also allows developers to manually override QoS thresholds when there are external events which mandate feature or configuration updates. For instance, when a new vulnerability in a TLS cipher suite is detected, switching cipher suites may be necessary but might have a strong impact on system performance [122, 127, 126].

There is also the challenge of finding a benchmark in the first place. In our case, we were running Apache Cassandra for which a number of open source benchmarks and benchmark tools exist. For some custom microservice, this will typically not be the case. In such scenarios, developers have to build their own benchmark first – comparable to test-driven development – which causes additional costs for personnel.

Finally, to conclude all cost aspects: CB will directly cause additional costs for the CB infrastructure and personnel costs for management or development of benchmarks. These costs, however, will likely be offset by indirect costs caused by unhappy customers or direct costs from compensation payments for SLA violations. Balancing these costs is a non-trivial task that is application-specific and should probably be approached in an agile way with continuous readaptation.

4.5 Conclusion

Complex systems are very sensitive to change and even the smallest change can strongly affect QoS. Particularly, such changes occur frequently when releasing new software versions.

Existing CI/CD pipelines, however, focus on functional testing or single method performance measurements and can, hence, not detect changes in QoS.

In this paper, we proposed a new approach called Continuous Benchmarking in which one or more system benchmarks are run as additional step in the build pipeline. Measurement results from these benchmark runs are then compared to either absolute thresholds, e.g., as specified in an SLA, or to relative thresholds which compare the result to previous results to assert that QoS levels always improve or at least remain constant across releases. We have prototypically implemented CB using Apache Cassandra as SUT and YCSB as benchmarking client and evaluated our approach by replaying almost two years of Cassandra's commit history.

Chapter 5

OpenISBT

5.1 Introduction

The complexity of today's software systems and their requirements are growing continuously. Thus, modern applications often rely on a microservice architecture to ease the development process(es), the deployment, and the operation of complex software systems with many components [108]. Instead of one large monolithic system, the business logic of an application is distributed across many small services which execute their specific tasks according to the UNIX-philosophy "Make each program do one thing well" [119].

While the functional requirements of individual microservices can be checked by specifying unit and integration tests, there are some challenges in ensuring non-functional requirements. State of the art live testing techniques coupled with monitoring include canary releases [82] or dark launches [57], which deploy a new version of the service in the production environment and assess its functionality and non-functional characteristics on a (small) share of actual traffic. However, live testing is not possible (i) when there is no production system (e.g., in early development stages), (ii) when testing for non-current workloads (e.g., testing the Christmas traffic of the shopping cart service in July), or (iii) when exposing even a fraction of actual traffic to a new, untested service is too risky. Thus, an alternative but complementary approach to live testing or monitoring is to benchmark new microservice releases. In contrast to live testing, benchmarking evaluates a microservice in a well-defined and isolated testbed which can also include related services. Benchmarking, thus, allows developers to evaluate of non-functional requirements – such as performance – of microservices in specific environments, for specific workloads, and over time. For example, repeatedly running the same benchmark in a controlled environment while the microservice evolves over time can identify performance regressions (or improvements) [69, 160, 48]. Benchmarking, however, can be costly to perform. Besides the setup procedure for the testing environment, workloads have to be defined, the

benchmark run has to be monitored, and finally the results must be analyzed to decide whether the requirements have been met.

In other domains such as database benchmarking, there are a variety of tools, e.g., YCSB [43, 24, 52], which leverage the common interface of database systems to achieve repeatability and portability. For microservice applications, however, interfaces and supported operations vary with every service, making it impossible to find a general interface for microservice benchmarking. Furthermore, as microservices evolve over time, interface changes will break ad-hoc benchmarks that a developer might have implemented.

In previous work [72], we proposed an approach for REST-based microservices in which developers can specify their benchmark workload through abstract interaction patterns. At runtime, these patterns are then automatically resolved and bound to a concrete microservice leveraging its REST characteristics. This allows developers to reuse both the abstract workload description and the benchmark execution environment [24] across microservice releases and the latter also across microservices. This significantly reduces the effort for developers while still ensuring that benchmarks fulfill important characteristics, namely that they are relevant, repeatable, portable, verifiable, and economical [16]. In our previous approach, we focused on single microservices – full microservice applications or dependencies across microservices were neither considered nor supported.

In this paper, we extend this work to microservice applications with multiple services and cross-microservice dependencies through a significantly more complex application-wide pattern matching process. As in our earlier work, we do this based on the machine-readable interface descriptions of the respective microservices and resolve the actual workload at benchmark runtime. Our extended algorithm identifies links between microservice operations and resolves the specified abstract patterns into application-specific interaction sequences which can span multiple microservices.

In this regard, we make the following contributions:

1. We propose an extended approach for benchmarking of REST-based microservice applications based on abstract and reusable interaction patterns.
2. We present a proof-of-concept prototype implementing our pattern-based benchmarking approach.
3. We evaluate our approach by benchmarking an open-source microservice application to demonstrate how little manual effort is needed for this.

Please, note that providing a complete pattern catalog is beyond the scope of this paper, but applying our approach in practice obviously requires a comprehensive set of interaction patterns.

The remainder of this paper is structured as follows: After outlining relevant background in Section **??**, we present our pattern-based benchmarking approach in Section 6.2 and its evaluation in Section 8.5. We discuss our approach in Section 8.6 and present related work in Section **??** before concluding in Section 8.7.

5.2 Pattern-based Benchmarking

Our pattern-based benchmarking approach relies on the observation that there are sequences of interactions with resources in REST APIs which recur across APIs. One common example is to list resources of a specific type (e.g., by performing *GET .../customers*), to then retrieve information about one specific resource (e.g., by performing *GET .../customers/1*), and finally deleting that resource (e.g., by performing *DELETE .../customers/1*). Based on this observation, we argue that it is possible to automatically generate benchmarking workloads from

- an abstract description of such patterns and
- a description of how to interact with each of the microservices' APIs (e.g., in OpenAPI format).

In this section, we start by describing the challenges in generating such a pattern-based workload. Next, we introduce our pattern-based solution in detail and finally give an overview of our approach's system design.

5.2.1 Challenges

We have identified the following three major challenges facing our approach:

(A) The first challenge is to define patterns and workloads for arbitrary microservice applications, including the total number of requests and their distribution across patterns.

(B) Once defined, the individual patterns must be mapped to the individual services and their respective operations. Here, an abstract pattern composed of multiple operations (e.g., *listResources*) must be linked to service-specific resources (e.g., a list of *customers* or *catalog items*) and its operations (e.g., *GET ../customers* and *GET ../catalogue*).

(C) Finally, the abstract requests must be filled with concrete parameter values depending on the interface definition which is, especially for request sequences, hard because parameter values may depend on the outcome of previous requests (potentially to a different service).

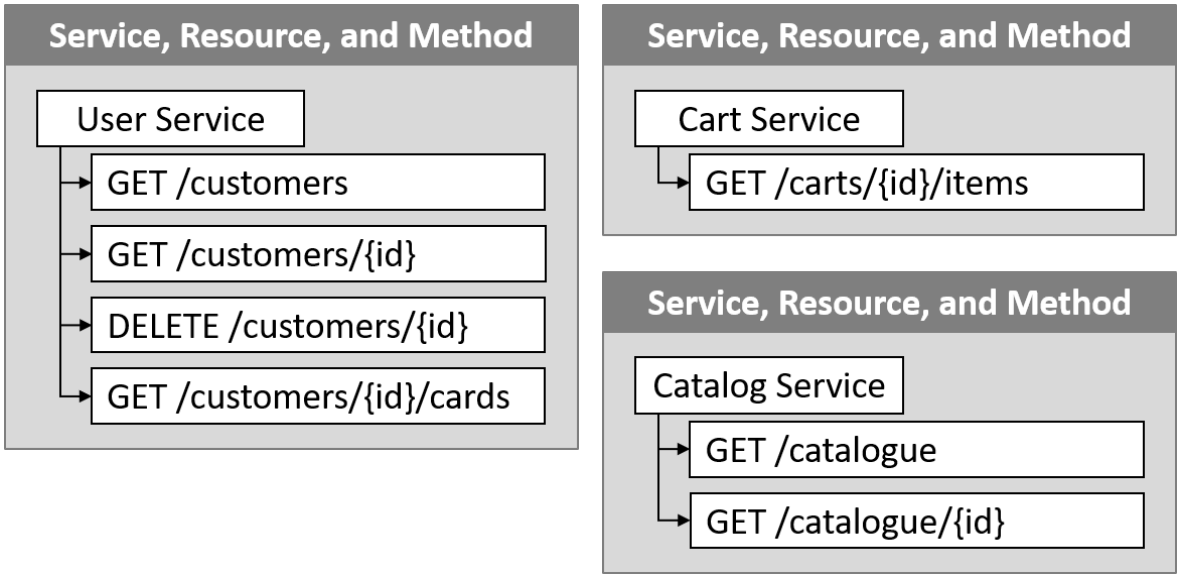


Figure 5.1: Example application used to explain our matching process.

5.2.2 From Abstract Interaction Patterns to Service-Specific Workloads

The key idea of our approach is to define an abstract workload separately from the services in an application itself and to resolve the actual service-specific workload at runtime. To address the challenges outlined above, which also have interdependencies, we divide this process into six steps (described in detail later). While challenge A is solved in the first two steps, steps 3 and 4 aim to cope with the difficulties described in Challenge B. Finally, challenge C, the actual workload generation, is covered in the steps 5 and 6.

1. **Pattern definition:** Define abstract interaction patterns.
2. **Workload definition:** Enhance pattern definition and specify frequency and ratio of requests.
3. **Binding definition:** Optionally, overwrite default binding behavior.
4. **Binding enactment:** Bind abstract interactions to concrete microservices and their operations.
5. **Workload generation:** Create application-specific workload.
6. **Benchmark execution:** Run the workload against the SUT and substitute values at runtime.

In the following, we will explain these steps using the example application listed in Figure 5.1. In the example, the customer and cart service use the same ID field.

Table 5.1: List of currently supported abstract operations which are combined to form abstract interaction pattern.

Operation	Description
CREATE	Creates and returns an item.
READ	Reads an item based on some filter (e.g., an ID) and returns the requested item.
SCAN	Reads multiple items based on some (optional) filter (e.g., a keyword) and returns the results.
UPDATE	Modifies an item based on some filter.
DELETE	Deletes an item based on some filter.

Table 5.2: Abstract interaction pattern which requests multiple resources, reads one random item from the resulting list, and finally deletes the selected item.

Step	Operation	Input	Selector	Output
1	SCAN	-	-	list
2	READ	list	RAND	item
3	DELETE	item	-	-

Step 1 – Pattern definition: The first step is to define abstract interaction patterns that are independent of the microservices but still applicable to them.

As defined by the second level of Richardson's maturity model, the typical REST CRUD operations can be mapped to HTTP methods: A new resource can be created at a resource endpoint by calling the *POST* HTTP method and accessed following a path structure at that endpoint. Individual resources can be read (HTTP *GET*), updated (HTTP *PATCH* or *PUT*), and deleted (HTTP *DELETE*). Finally, multiple resources can be listed by sending an HTTP *GET* to a list operation (e.g., *GET ../search*) which potentially may return multiple items. In the very first step of our approach, we use these basic interactions to define the abstract operations shown in Table 5.1 which we will later use to bind abstract patterns to concrete service resources.

While almost all of these interactions refer to a specific single resource, read operations can request either a single resource or multiple resources; we therefore distinguish the two read operations *READ* (single) and *SCAN* (multiple). Most operations require some filter information about the item to read, update, or delete. These do not only include an ID or key of the requested resource, but also further domain-specific values if multiple items should be read (SCAN). Furthermore, we introduce selectors as part of these filter information: If a list of items serves as input for an operation, the selector determines which item to pick from that list (e.g., first, last, or random item).

Table 5.3: Abstract interaction pattern which queries a list of resources and then requests all associated resources for one random item.

Step	Operation	Input	Selector	Output
1	SCAN	-	-	list
2	SCAN	list	RAND	sublist

Our abstract operations already cover the common CRUD interactions with REST services. If necessary, our approach can be extended with additional basic operations. Using the basic operations from Table 5.1, we can now compose an interaction pattern as a sequence of abstract interactions. Thereby, each interaction is linked to a microservice, an operation, an abstract resource, and optional filter information. Moreover, it must define where to store output values of an interaction and from where to obtain input values.

The complete interaction pattern for the abstract example from the beginning of this section is shown in Table 5.2: First, a SCAN operation determines all available resources on a service resource endpoint and stores the resulting values in a variable called *list*. Next, an individual value is selected by a random selector from this list, the corresponding resource is read (possibly from another microservice), and stored into a variable called *item*. Finally, the selected item is deleted.

Table 5.3 motivates a more complex example with sub-resources. The interaction starts again with a *SCAN* of available resources and the outcome is written to a variable called *list*. Applied to a microservice application, this could, e.g., request a list of users which each have a number of sub-resources. Next, a random item from that list serves as input for a subsequent *SCAN* operation which requests all (sub-)resources for the chosen item, e.g., a list of all credit cards for the selected user or a list of all items in the user's shopping cart (see Figure 5.1). Figure 5.2 shows how the binding of the pattern shown in Table 5.3 would later be resolved for the two example sub-resources: In the upper part, the selected customer ID is used to retrieve all credit cards of the selected user from the same microservice as sub-resources. In the lower part, the ID is used to list all items in the selected user's cart via a second microservice.

Step 2 – Workload Definition: The next step is to specify the actual workload which should be executed against the SUT. Similar to the business transactions in BenchFoundry [24], a pattern definition can include optional conditions for individual patterns (e.g., waiting times between operations to mimic realistic user behavior). Comparable to YCSB [43], which defines a workload based on a total number of operations as well as the respective share of each database operation, we define a workload based on three pieces of information: first, the list of all patterns which shall be used; second, the total number of pattern invocations; third,

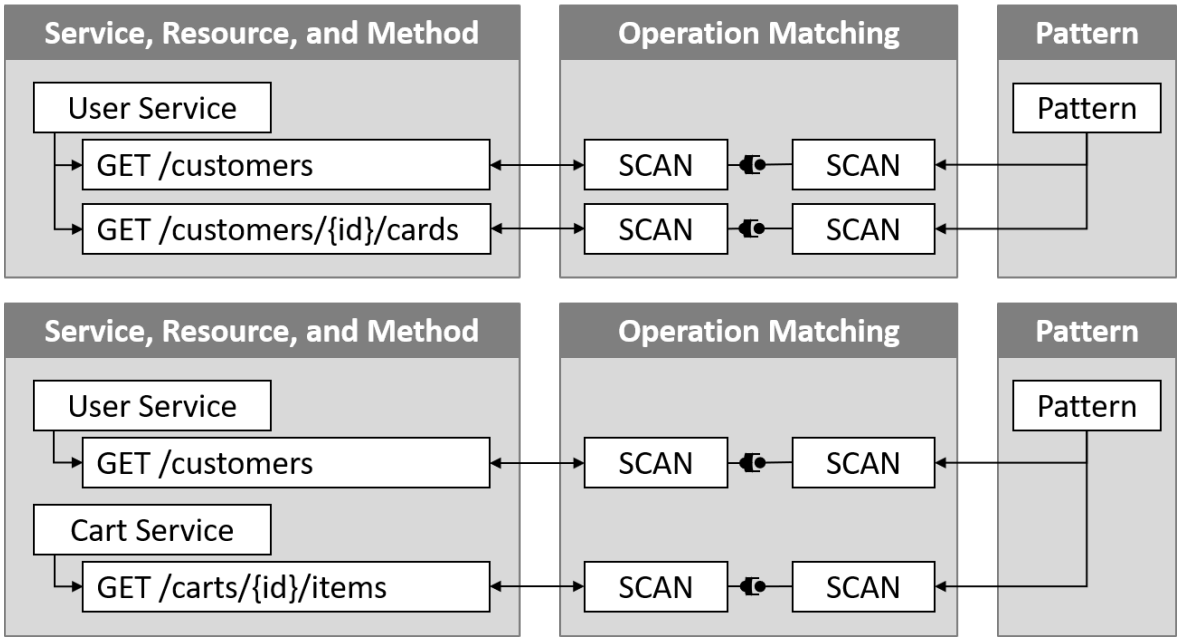


Figure 5.2: Matching the pattern from Table 5.3 to two different interaction sequences.

the share or weight of each pattern. At execution time, multiple such patterns are usually executed in parallel.

In the following, we will refer to the abstract interaction patterns defined in Step 1 and the workload definition as *pattern configuration*.

Step 3 – Binding Definition: While our matching algorithm automatically identifies links between requests, there are several cases in which these automatic bindings should be suppressed or require additional information. In this optional adjustment step, developers can manually exclude specific service operations from the matching algorithm or define links between microservices. If used, this provides additional information to the default binding process described in Step 4 below.

As one usage scenario, microservice applications sometimes provide multiple resource endpoints (e.g., */customers* and */catalogue*) which can be used by the benchmarking client for an interaction pattern. By default, all possible resource endpoints and operations are used by the benchmark. When, however, the automatic binding from pattern to resource and operation should be suppressed (e.g., in case that only the */customers* resource endpoint should be benchmarked), this can be achieved by excluding the respective service endpoint for a subset of the patterns.

As another usage scenario, a manual definition can also be used to link parameters of two services with the goal of enabling interaction sequences which span multiple microservices of the same application. For example, if the resources of the user service can be accessed through

a parameter called *id* and another service uses the same IDs to maintain the shopping carts for the respective user, then our binding algorithm needs this manual link definition to match an abstract pattern to these two services, i.e., to clarify that the second ID is indeed the user ID and not the cart ID. Here, semi-automatic approaches based on text similarity measures can support developers to define these links. For simplicity of presentation, however, we will focus only on manual links between services in this paper.

Step 4 – Binding Enactment: As already described above, our approach for automated binding enactment relies on a number of key ideas: First, REST operations can directly be mapped to the corresponding HTTP methods, e.g., a *CREATE* is mapped to an HTTP *POST*. Second, a microservice which complies with the second level of Richardson's maturity level exposes its operations in a way that is compliant with the REST operation semantics, e.g., creating a new user will always be exposed as a *CREATE* which can then be mapped to *POST*. Third, the input and output of these operations as well as the corresponding data schema are described in the interface description file, i.e., in our case, the OpenAPI file, so that we can link the output of one operation to the input of another. This allows us to create the cross-operation links in our interaction patterns. Finally, the interface description also provides information on where to find the microservice, hence, we can actually invoke it once we have completed all the mappings as described above. Based on the reasoning above, we can automatically generate a binding between an interaction pattern and the actual sequence of HTTP calls – subject to the conditions above, e.g., that creating a new user is not exposed as a *PUT*.

A pattern can be mapped to a microservice application if there is at least one supported interaction sequence for this pattern within the application. As shown in Figure 5.2, this can either affect a single service only or an operation result can be used to interact with another service, thus, enabling a cross-service benchmark run.

Algorithm 1 describes the algorithm we use to match and generate the application-specific interaction sequences; it has the three phases Initialization, Matching, and Sequence Extraction.

Step 4.1 – Initialization: First, we analyze the interface description of all services and determine the abstract operation for all resource paths and operations (line 3). While it is easy to determine the operation for creation, modification, and deletion as they directly map to an HTTP method by convention, this is more difficult for the SCAN and READ operation. Here, we have to inspect the resource path a bit further and check if it ends with an input parameter. If so, the parameter refers to a key or an ID and supports the READ operation; If the resource path does not end with a parameter and returns an array of items, it can be bound to the SCAN operation.

Algorithm 1: Generate Interaction Sequences

```
   Input: pattern - abstract interaction pattern
   Input: specs - list of interface descriptions
   Result: sequences - supported interaction sequences
 1
 2 /* Initialization                                                    */
 3 AO = IdentifyAbstractOperations(specs)
 4 root = CreateRootNode()
 5
 6 /* Matching                                                         */
 7 for i ← 0 to pattern.size do
 8 |   a = pattern[i]
 9 |   candidate = FindByAbstractOperation(a, AO)
10 |   foreach node ∈ GetNodesByLevel(i,root) do
11 |   |   foreach o ∈ candidate do
12 |   |   |   if AreDependenciesResolvable(node, o) then
13 |   |   |   |   AddChildNode(node, o)
14 |   |   |   end
15 |   |   end
16 |   end
17 end
18
19 /* Sequence Extraction                                              */
20 sequences = ExtractSequences(root)
```

Next, we initialize a virtual root node of a tree data structure which we will use to store intermediate results of pattern matching (line 4). The paths in the tree from root to leaf will later hold our interaction sequences.

Step 4.2 – Matching: In the Matching phase, we iterate over the abstract operations in the pattern and select all service operations of the same abstract type as a list of operation candidates (lines 7-9). For the abstract interaction pattern in Table 5.2 and our example application, the first operation can be mapped to three service-specific operation candidates: listing all users, listing all credit cards of a user, and listing all items of the catalog service (see Figure 5.3).

Next, we iterate over all leaf nodes *node* at the specified level in the tree (for the first pattern operation this is the root node) and check for every candidate operation *o* whether its dependencies can be fulfilled on the path from *root* to *node* (lines 10 - 16). Candidate operations that do not require input values (such as listing all users) have no dependencies and can, hence, always be used. In that case, we simply add this operation as a child node to *node*. If, however, there are dependencies, we verify that the required information can be obtained from previous requests on the path from *root* to *node*. In our example, this affects the sequence

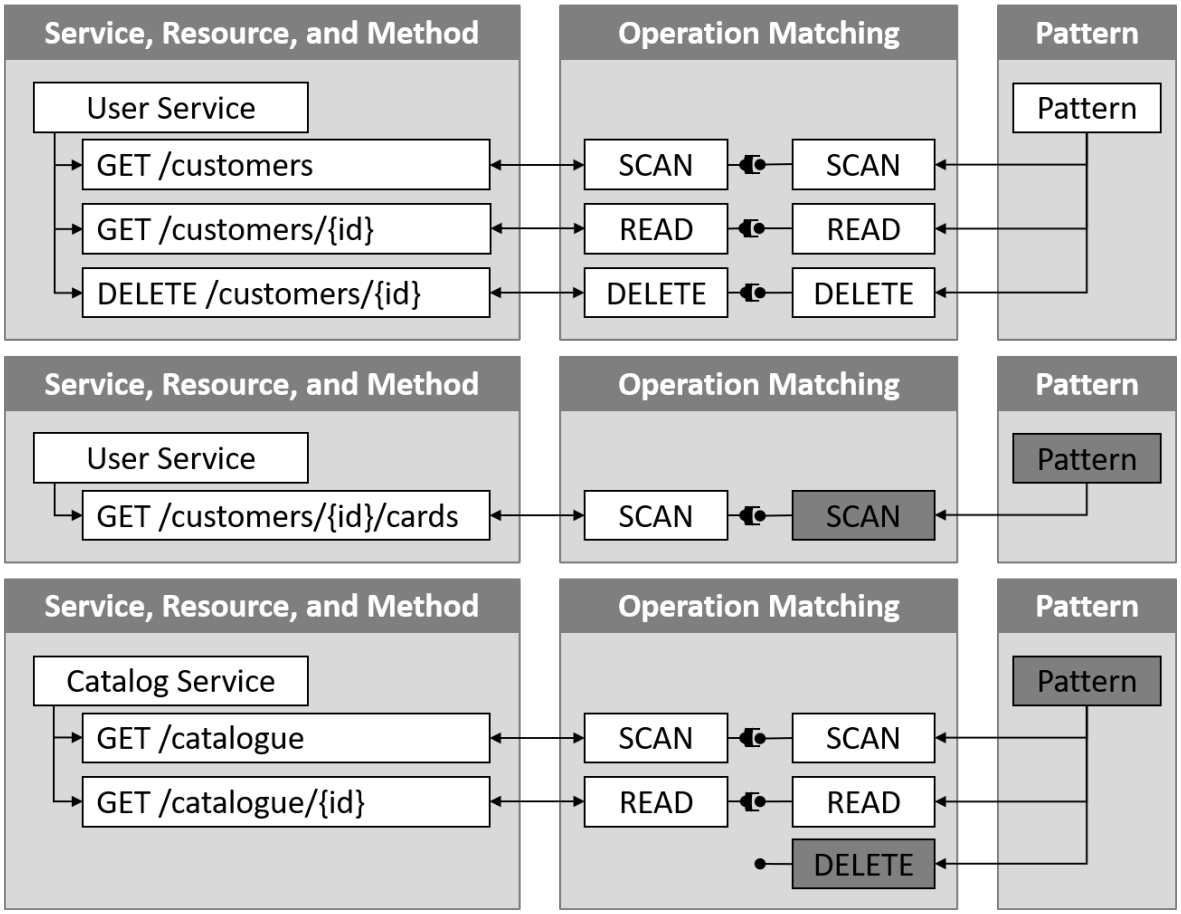


Figure 5.3: Mapping a pattern to interaction sequences, only one sequence (list all users, read one random user, and delete the selected user profile) supports the example pattern.

starting with listing the cards of a user: As it is a candidate for the first operation, there is no information available on the required user ID, see Figure 5.3. Any operation for which all dependencies can be resolved is added as a child node; all other operations are disregarded (lines 12 - 14). In this step, it is also possible to define additional conditions that an operation needs to fulfill before it can be added as a child node; in our prototype, for instance, we have implemented a filter that only adds operations that target microservices already used on the current path or microservices for which an explicit link has been defined in step 3. We do this to limit the number of interaction sequences, but it is not strictly necessary to do so.

We execute this for all operations in the pattern and, thus, gradually build our tree data structure. In the third example in Figure 5.3 for instance, we can see that the first two operations of the example pattern can be matched against the catalog service. The third operation (DELETE), however, cannot be matched since the only available service-specific DELETE requires a user ID as input which cannot be obtained from the two previous operations (listing all catalog entries and reading a specific catalog item).

Step 4.3 – Extract Sequences: Finally, we extract the interaction sequences from our tree. For this, every path from the root node to a leaf that – excluding the root node – contains the same number of nodes as the pattern has operations represents an interaction sequence. Shorter paths are incomplete interaction sequences where one or more pattern operations could not be matched; these can be discarded.

When at least one interaction sequence per pattern has been found, the binding is saved and represents the automatic binding for the combination of patterns, binding definition, and interface descriptions. In some cases, the binding algorithm may find sequences that are not relevant or not desired in the respective use case. These sequences can either be deleted or deactivated manually or can be used to create additional load on the SUT.

Step 5 – Workload Generation: With the previously created binding and the interface descriptions, we can finally generate the benchmark workload by building HTTP requests which follow the interaction patterns and the restrictions in the interface description files. First, each pattern operation can be directly resolved to an HTTP method based on the binding. Next, we can fill the required parameters and request body content of each request by inspecting the interface description of the respective microservice and generating random values for all interactions: In OpenAPI, complex parameter values and request bodies are described using JSON Schema.[1] We can use these descriptions to generate the required data items filled with random values. Moreover, we can generate use-case specific values such as product names or Bitcoin addresses by augmenting the service description with special keywords.

As stated above, some content of the individual requests may depend on the returned values of previous calls (e.g., identifier values). These values must be injected later in the benchmark execution phase (Step 6) for which we use special markers. Nevertheless, once the required number of pattern executions has been generated, the workload can be persisted and reused across several benchmark runs even if the generated workload is incomplete in that sense.

Step 6 – Benchmark Execution: As already stated above, some values of the workload must be replaced during the execution if there are dependencies between requests. For these values, there are many sources depending on the concrete operation: A *CREATE* operation could, for example, return the ID of the created item or respond with an HTTP 200 status code if the ID was part of the request and the item was created successfully. Depending on the implementation, the subsequent *READ* request must select the ID from the response or the request body of the preceding request; this, however, only if the response had an HTTP 200 status.

[1] `https://json-schema.org/`

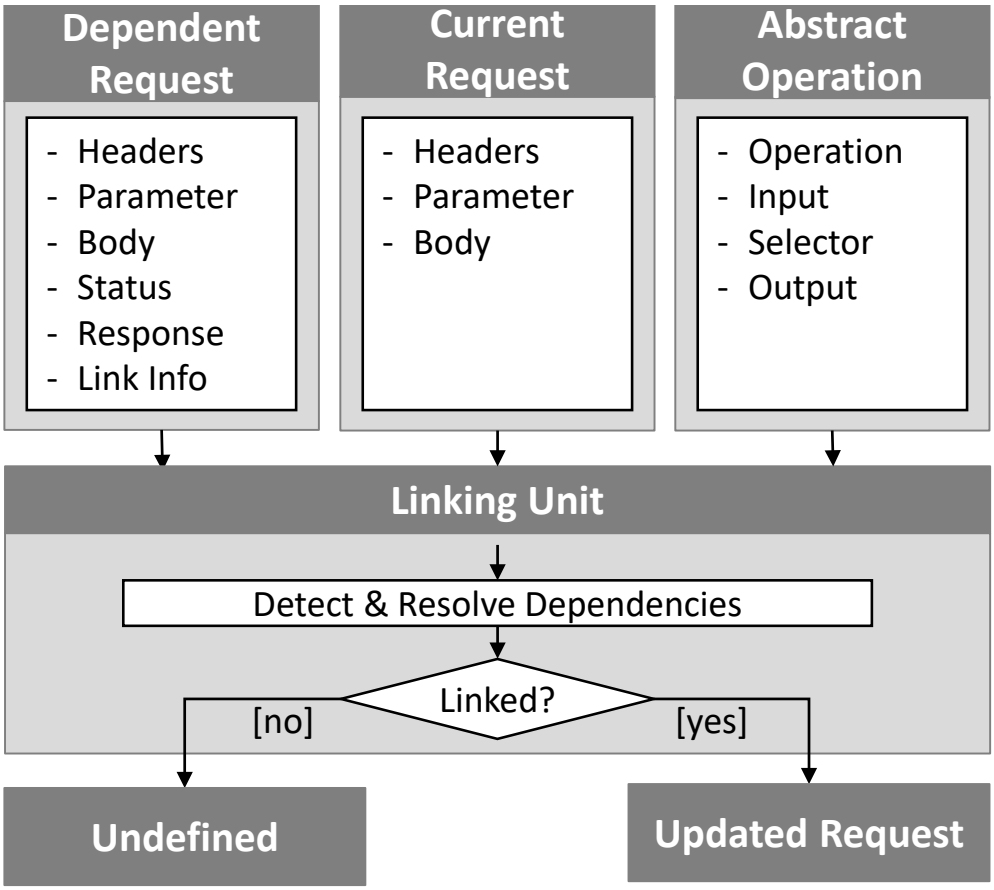


Figure 5.4: Linking two related requests based on the content. If a link is detected, the successive request is updated and returned.

For such purposes, we have designed the *linking unit* interface illustrated in Figure 5.4: A linking unit tries to find and resolve dependencies based on the preceding requests (including message body, response, etc.), the current request (including the values to be replaced, e.g., a parameter), and the abstract pattern operation. If a linking unit detects a link, it resolves the dependency, e.g., by replacing the placeholder value in the current request body with some value from the parameters of the previous one, and returns the updated request or *"undefined"* otherwise. This way, it is possible to apply multiple linking units sequentially until a dependency has been resolved; it also allows us to order the application of several units hierarchically (e.g., general or very service-specific units first).

Currently, we have identified four different types of linking units but additional, potentially service-specific, custom units can be added:

- **OpenAPI Link Linking Unit:** OpenAPI 3.0 documents can define links which describe further operations and their content after a query. This linking unit inspects the link definitions of the preceding request and replaces the values of the current request accordingly.

- **Binding Linking Unit:** This unit resolves the links manually defined in the binding definition (see step 3).
- **Parameter Name Linking Unit:** Individual resources can often be accessed by following a path structure in REST interfaces, e.g., */{username}*. These parameters were initially filled with placeholder values in Step 5 which have to be adjusted now to access actual resources. This linking unit searches for these values in the preceding request based on the parameter name. If exactly one element with this name as key is found in the request (either in parameter values, request body, or response), this value is used in the current request. If there are multiple values to choose from, which one is chosen is determined by the selector (e.g., select a random value).
- **ID Linking Unit:** In some cases, parameter names in the preceding and current requests do not match exactly. For example, a user is created with a field named "*id*" in the request body and individual users can be accessed via the path */{userID}*. This linking unit resolves dependencies by searching field names for the substring "id" and replacing values in the same fashion as the parameter name linking unit.
- **Custom Linking Unit:** Finally, as dependencies cannot always be detected and resolved with our default linking units, developers can also define custom and service-specific linking units which can be added to the application chain of units by implementing the linking unit interface.

5.2.3 System Design

Our system design comprises a number of components; these – along with the corresponding steps – are shown in Figure 5.5. The Pattern Binder creates service-specific interaction sequences based on API description files, a pattern configuration, and optional binding definitions (steps 1 to 3). Once the Pattern Binder has bound every pattern to at least one interaction sequence (Step 4), the binding can be stored and, if necessary, adjusted manually (e.g., if the automatic algorithm identified an unusual but possible interaction sequence). Next, the Workload Generator generates the service-specific but incomplete workload based on this pattern binding (Step 5). As a pattern execution is by definition independent of other pattern executions (otherwise, they should be merged into one pattern), we can parallelize pattern execution and also distribute this execution across a number of Worker Nodes. Similar to the method proposed in [24], the Benchmark Manager does this by partitioning the workload into worker packages to enable concurrent execution (the number of packages corresponds to the number of concurrent Worker Nodes), then distributes the worker packages to the available Worker Nodes, manages the (concurrent) benchmark execution, and collects the results. Finally, as our approach is intended for use in Continuous Integration and Deployment pipelines [69, 160], the Benchmark Manager compares the observed metrics to predefined

requirements and constraints such as service level agreements (SLAs) to ultimately decide on success or failure of the benchmark run.

Within a worker package, requests across patterns can be interleaved as long as requests within a pattern are not reordered. As some requests depend on the outcome of preceding ones (e.g., the UPDATE operation requires the resource ID which was part of the result from a previous CREATE call), the Worker Nodes cannot simply read the generated workload and run the requests against a service endpoint, but must adapt some values at runtime with the outcome from posted requests based on the linking units (Step 6).

5.3 Evaluation

In our original paper [72], we evaluated our benchmarking approach through a proof-of-concept prototype and a set of experiments with three different microservices to demonstrate the general applicability for benchmarking individual services. In this paper, we omit these single service experiments and focus on evaluating our extended approach by benchmarking an entire microservice-based application with multiple services using our improved proof-of-concept prototype. As, again, our focus is the applicability of our approach, the actual measurement *results* are irrelevant as long as results *can* be obtained. In this section, we first present our proof-of-concept implementation and describe the microservice application which we benchmarked. Next, we describe how we followed the individual steps of our approach to create and run a pattern-based benchmark workload. Finally, we summarize our evaluation findings.

5.3.1 Proof-of-concept Implementation

We implemented our approach and system design as an open-source proof-of-concept prototype[2] written in Kotlin. Our prototype implementation can be integrated in an existing Continuous Integration and Deployment pipeline and comprises four components:

1. A Pattern Binder which maps abstract interaction patterns to service-specific operations.
2. A Workload Generator which fills the requests with random values based on the interface description.
3. A Benchmark Manager which orchestrates the benchmark run and aggregates the results.
4. Worker Nodes which execute the workload, resolve dependencies between successive requests using linking units, and measure the runtime of each operation to report the results.

[2] `https://github.com/martingrambow/openISBT`

Our prototype uses the open-source library json-schema-faker[3] to generate data for the workload. This library allows us to generate the required JSON elements for the individual HTTP requests based on the schema information in the OpenAPI description file. Moreover, our prototype also supports special faker keywords (defined by the library Faker.js[4]) which can be added to the OpenAPI file. These additional keywords can be used to generate realistic and use-case-specific workload values (e.g., names, product IDs, or dates).

5.3.2 Sock Shop Microservice Application

We evaluate our approach on a microservice-based Webshop[5] for socks which implements an e-commerce application. In our experiments, we focus on the following three REST services and dependencies (see Figure 5.6):

User Service[6]: The user service maintains the customer information such as usernames, passwords, credit cards, and addresses. Each user has a unique customer ID which is used to access the respective data set as REST resource. Moreover, there are also resource paths for credit cards and addresses. For example, a new address can be created by sending an HTTP POST to */addresses* and all addresses of a customer can be queried by sending an HTTP GET to */customers/{id}/addresses*.

Cart Service[7]: Each user has a virtual shopping cart which is as REST resource identified by their customer ID. Items from the catalog service can be added, deleted, or modified.

Catalog Service[8]: The catalog service provides an interface for retrieving all products available in the shop. A data item comprises an item ID (which is also used for accessing the respective REST resource), a description, tags, and the price of the corresponding product. The service only offers operations for browsing existing products, items can neither be modified nor added or removed.

5.3.3 Experiment

In line with our proposed process, we run the following experiments to evaluate our pattern-based benchmarking approach:

Step 1 – Pattern Definition: We evaluate the REST microservices discussed above with four self-defined abstract interaction patterns as shown in Table 5.4: First, a simple list pattern which lists available resources and reads an item from that list (LST). Second, our

[3] https://github.com/json-schema-faker/json-schema-faker/
[4] https://github.com/marak/Faker.js/
[5] https://microservices-demo.github.io/
[6] https://github.com/microservices-demo/user
[7] https://github.com/microservices-demo/carts
[8] https://github.com/microservices-demo/catalogue

Table 5.4: An overview of the four interaction patterns which we use in our experiments.

Pattern	Step	Operation	Input	Selector	Output
LST	1	SCAN	-	-	list
	2	READ	list	RAND	-
DEL	1	SCAN	-	-	list
	2	READ	list	RAND	item
	3	DELETE	item	-	-
SUBLST	1	SCAN	-	-	list
	3	SCAN	list	RAND	sublist
TWOIN	1	SCAN	-	-	list1
	2	SCAN	-	-	list2
	3	CREATE	list1, list2	RAND, RAND	item

introductory example pattern from Table 5.2 which identifies and deletes a resource (DEL). Third, our more complex example pattern from Table 5.3 which lists sub-resources for a randomly selected root resource (SUBLST). Fourth, another complex pattern which creates a new resource based on two input values (TWOIN).

In all steps where an item needs to be selected from a list, we always use a random selector for reasons of simplicity but there will of course be services for which another selector makes more sense, e.g., picking the latest item. Moreover, we want to emphasize again that these patterns are examples only, as our goal is not to identify a comprehensive pattern catalog.

Step 2 – Workload Definition: In this evaluation, we want to demonstrate the general functionality and applicability of our approach and not to rate the performance of individual microservices. Thus, we run rather "small" workloads and use one benchmark run only. In practice, however, these parameters must be adapted to fulfill usual benchmark best practices, e.g., regarding execution duration [16]. For our experiment, we run 1,000 pattern requests in total, 250 per pattern. We also run an initial preload phase which inserts 1,000 customers including one credit card, one address, and one cart item in advance for each user. The catalog service already offers nine items out of the box.

Step 3 – Binding Definition: For our experiment, we do not have to manually exclude operations (this may be different in other scenarios). We, however, define five manual links for the ID fields of the evaluated microservices as shown in Figure 5.6.

The customer ID, which is the key for accessing the REST resource, is used in four different contexts under different names: Within the user service, the field *id* is used for the */customers*

and */register* paths. Moreover, the same value is referenced as *userId* in the */cards* and */addresses* paths. Besides these links, we define two links which connect the user service to the cart service via the customer ID and the cart service to the catalog service via the item ID.

Step 4 – Binding Enactment: Here, we bind our example patterns to the target microservices and their resources. Figure 5.7 outlines the resulting binding for each pattern, including our manual definitions from Step 3.

For the *LST* pattern, our binding algorithm identifies five possible interaction sequences. Three of them are limited to the user service (e.g., query a list of customers and get the details of a random customer afterwards), one interacts with the catalog service (get all items and select a random one), and one sequence combines the user and cart service (query a list of customers and get the virtual cart of one random customer from the resulting list).

The *DEL* pattern can be found in six interactions. The first four interactions start with listing registered customers. Next, one randomly selected customer ID can be used to either get the details for this customer or to get the customer's shopping cart. Third, the customer ID serves as input to delete either the customer or the cart. The remaining two sequences start with listing all credit cards or addresses. In the first binding iteration, the search is restricted to the respective endpoint and the */customer* endpoint as no other link has been defined (we discussed this additional condition in Step 4 of Section 6.2). Thus, the *id* fields are not mixed up and the *READ* operation continues to use either the */cards* or */addresses* endpoint. The */customer* endpoint is disregarded because the *customer.id* is not linked to the *cards.id* (but to *cards.userId*). The same holds for the second binding iteration and the final *DELETE* operation.

Our binding for the *SUBLST* pattern identifies three interaction sequences, all starting with listing customers. Next, the randomly selected customer ID can be used to list the user's addresses, credit cards, or shopping cart items.

The *TWOIN* pattern combines two different inputs to create a new resource. In our evaluated application, this pattern can be used to add items from the catalog to a user's shopping cart. For the first two operations, it does not matter whether customers or items are listed first as neither operation has a dependency on the respective other one. Thus, our binding algorithm identifies two interaction sequences, one starting with listing customers, the other starting with listing catalog items. Finally, the *CREATE* operation requires one input as parameter and another input inside the request body. Here, other operations (such as registering a new customer) are not matched because they either expect only one input value or the manually defined service links do not match (e.g., the *card.id* is not linked to the *cart.customerId*).

Step 5 – Workload Generation: To generate the workload for our experiments with our prototype, we adjust the OpenAPI files slightly for the following reasons: First, we convert[9] the service descriptions to the current OpenAPI version 3.0 for implementation reasons. Second, we align the description with the actual implementation, as the OpenAPI file is missing a few properties offered by the corresponding implementation. Finally, we add faker.js keywords to generate realistic random values, e.g., for names, numbers, and dates.

Step 6 – Benchmark Execution: We run our benchmark on AWS EC2[10] instances, all in the same availability zone. For our experiment, one instance runs the Sock Shop microservice application including the three evaluated microservices and two instances each run a Worker Node with five threads each to demonstrate the parallelization and scalability features of our prototype. Finally, a fourth instance hosts the Pattern Binder, Workload Generator, and Benchmark Manager.

Our experiment benchmarks three different REST microservices of an example application. As a result, we get pattern execution measurements which include the total pattern duration and the duration of individual requests. These measurements can, e.g., be used to generate box plots for each pattern and interaction sequence. Figure 5.8 shows the latency at pattern level for our evaluated patterns using box plots[11]. Again, since the actual results are irrelevant and only show the applicability of our approach, we do not discuss the measured results.

5.3.4 Summary

As described above, our prototype can benchmark typical microservice-based applications with minimal adaptation effort. After initializing all application services, our proof-of-concept implementation benchmarks the evaluated and linked microservices (almost) out of the box and only a few minor changes in the description file are needed. Moreover, the abstract workload definition – which, in our evaluation example, is a JSON file with less than 100 lines – can be reused across a variety of services in different versions. The five manual links between the microservices which we had to define are about 30 lines. Unless major breaking interface changes are made, these manual links can be reused across benchmark runs against different versions of the microservice application. Overall, we managed to design and setup the benchmark for the entire microservice applications in less than an hour in total which significantly reduces the effort necessary to benchmark a microservice: there is simply no need anymore to manually implement a benchmark (tool) from scratch which can also be quite challenging [22].

[9] `https://mermade.org.uk/openapi-converter`
[10] `https://aws.amazon.com/ec2/`
[11] Boxes represent 25%, 50%, and 75% quartiles; whiskers show min and max duration for each sequence

5.4 Discussion

As the evaluation shows, our approach can be used to benchmark REST microservices based on their service description but, nevertheless, there are some points to consider when applying our pattern-based approach and which we want to discuss in more detail here. Moreover, we also present current limitations and propose possible solutions for them.

Our design generates a synthetic and traced-based workload based on a pattern and workload definition. Defining a proper workload comes with its own challenges (which, however, is not specific to this approach). If a workload definition creates more resources than deleting existing ones, the list of resources grows with every iteration and may produce unrealistic workloads. E.g, the workload definition from our evaluation may fit the carts service because the number of items is usually constant (about the same number of additions and deletions) but, on the other hand, it may not fit the customer service well, where the number of users usually increases during the service lifetime because there are more new users registering than existing ones leaving. Thus, the actual patterns and a realistic distribution of these patterns has to be considered when defining the workload (e.g., by inspecting the log files to identify common interactions and their frequency [87]). This also implies that an existing workload based on a common pattern catalog yet to be defined cannot be applied blindly to other microservices.

Next, our approach relies on the semantics of REST-based interfaces and assumes HTTP-based microservices. Microservices using other communication protocols can be used as well but essentially require manual bindings for every operation. Since the basic CRUD semantics exist independent of the protocol used, it will be interesting to see whether there are common ways in which these are exposed in non-REST-based microservices and whether these could be leveraged by our approach. E.g., the abstract operations could be mapped via the function name instead of the HTTP verb. Nonetheless, our approach can already be used for a large variety of microservices for which there are no benchmarking alternatives yet.

While not every pattern and workload definition can be blindly applied to every REST microservice, our approach allows developers to run a benchmark against REST services which share the same characteristics in general, which is helpful in several situations: First, every new service version can be compared to older versions as long as the individual changes do not alter the basic characteristics of the microservice. This is particularly important for use in Continuous Integration pipelines [69, 160, 48]. Second, if the API changes (e.g., a new parameter is introduced or a schema is adjusted), nothing but the interface description file must be replaced (ideally, this description is generated from the microservices' source code with every build) and the benchmark adapts automatically, there is no need to adjust workload files or source code. Third, our approach can be used to evaluate different services which share the same purpose (e.g., user management). This is particularly useful when replacing a

microservice with a new one as both can be benchmarked and compared extensively prior to switching in production.

In contrast to our initial prototype which was limited to single microservices and did not consider cross-service dependencies and sub-resources, our extended approach addresses this issue. There are, however, still some limitations: First, the dependencies between microservices must be defined manually in advance. This requires domain knowledge about the target application and can be hard for applications with many services. Here, our prototype could be enhanced with an automated analysis based on text similarities which detects the service dependencies automatically to support application developers, e.g., that identifies that `userID` and `user/{id}` refer to the same property. Furthermore, our binding algorithm identifies all possible interaction sequences for a pattern configuration (and an optional binding definition). This should be handled with caution as the binding might also identify unrealistic sequences which should be disabled before running the benchmark. E.g., two of the *DEL* interaction sequences which we found in our evaluation might be unrealistic (in another scenario). Here, the third operation deletes a shopping cart after getting the details for a selected customer or deletes a customer after querying its cart. Thus, we recommend verifying the identified bindings manually before running benchmark experiments, especially for applications which involve multiple microservices. Nevertheless, in the worst case, such unrealistic matchings simply add additional load on the SUT – it is only important to disregards their respective results.

Overall, we believe that our approach and its prototypical implementation are useful for a large percentage of microservice deployments as they significantly reduce the implementation effort for microservice developers. Some restrictions such as the REST requirement apply but could also serve as an incentive to switch to REST in some cases where other communication solutions are used for legacy reasons.

5.5 Conclusion

Benchmarking microservices serves to understand and check their non-functional properties for relevant workloads and over time. Performing benchmarks, however, can be costly: each microservice requires the design and implementation of a benchmark from scratch, possibly repeatedly as the service evolves. As microservice APIs differ widely, benchmarking tools, which typically assume common interfaces of the system under test, do not exist yet.

In this work, we proposed a pattern-based approach to reduce the efforts for defining microservice benchmarks, while still being able to measure qualities of complex interactions. Our approach assumes that microservices expose a REST API, described in a machine-understandable way, and allows developers to model interaction patterns from abstract operations that can be mapped to that API. Required parameter values are provided at run-

time and possible data-dependencies between operations are resolved. We implemented our approach in a prototype, which we used to demonstrate the low effort applicability of our pattern-based benchmarking approach to an open-source microservice-based application with multiple services and dependencies. With this, we could show that pattern-based benchmarking of microservices is indeed feasible which opens up opportunities for microservice providers and tooling developers.

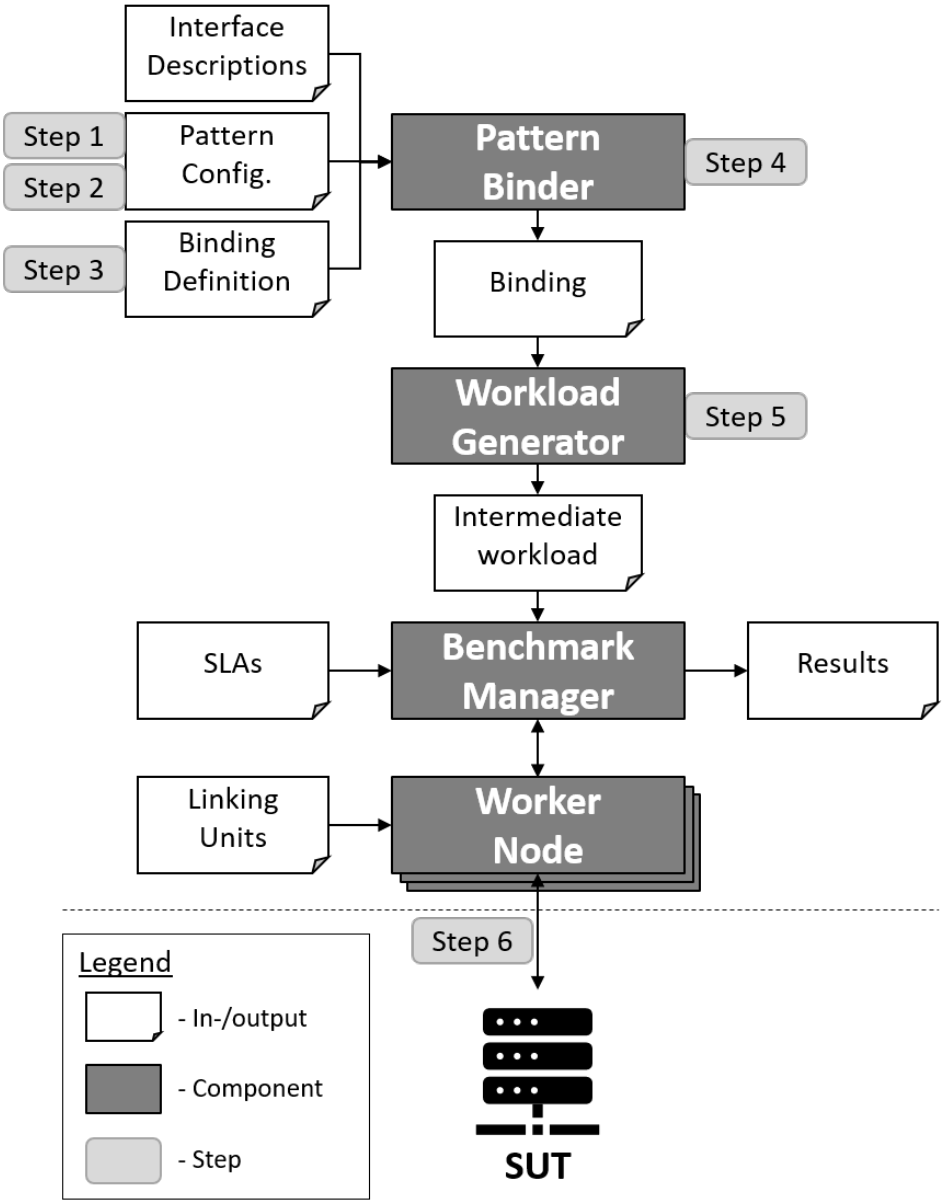


Figure 5.5: Overview of our system design and setup including input and output documents.

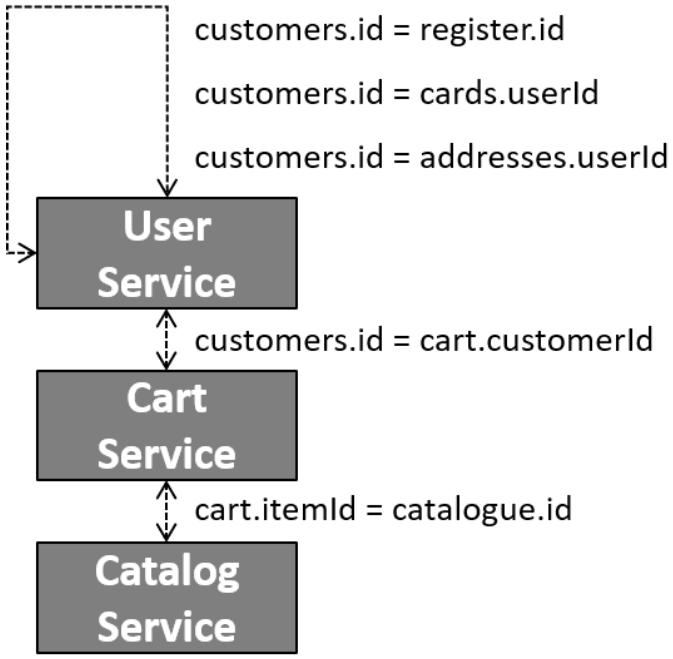


Figure 5.6: Our evaluation focuses on three microservices of the Sock Shop application and their interdependencies.

Interaction Sequences for LST pattern

1	GET /customers
2	GET /customers/{id}

1	GET /cards
2	GET /cards/{id}

1	GET /addresses
2	GET /addresses /{id}

1	GET /customers
2	GET /carts/{customerId}

1	GET /catalogue
2	GET /catalogue/{id}

Interaction Sequences for DEL pattern

1	GET /customers
2	GET /customers/{id}
3	DELETE /customers/{id}

1	GET /customers
2	GET /carts/{customerId}
3	DELETE /carts/{customerId}

1	GET /customers
2	GET /customers/{id}
3	DELETE /carts/{customerId}

1	GET /customers
2	GET /carts/{customerId}
3	DELETE /customers/{id}

1	GET /cards
2	GET /cards/{id}
3	DELETE /cards/{id}

1	GET /addresses
2	GET /addresses/{id}
3	DELETE /addresses/{id}

Interaction Sequences for SUBLST pattern

1	GET /customers
2	GET /customers/{id}/addresses

1	GET /customers
2	GET /customers/{id}/cards

1	GET /customers
2	GET /carts/{customerId}/items

Interaction Sequences for TWOIN pattern

1	GET /customers
2	GET /catalogue
3	POST /carts/{customerId}/items

1	GET /catalogue
2	GET /customers
3	POST /carts/{customerId}/items

Figure 5.7: All our example patterns can be matched to at least one interaction sequence each.

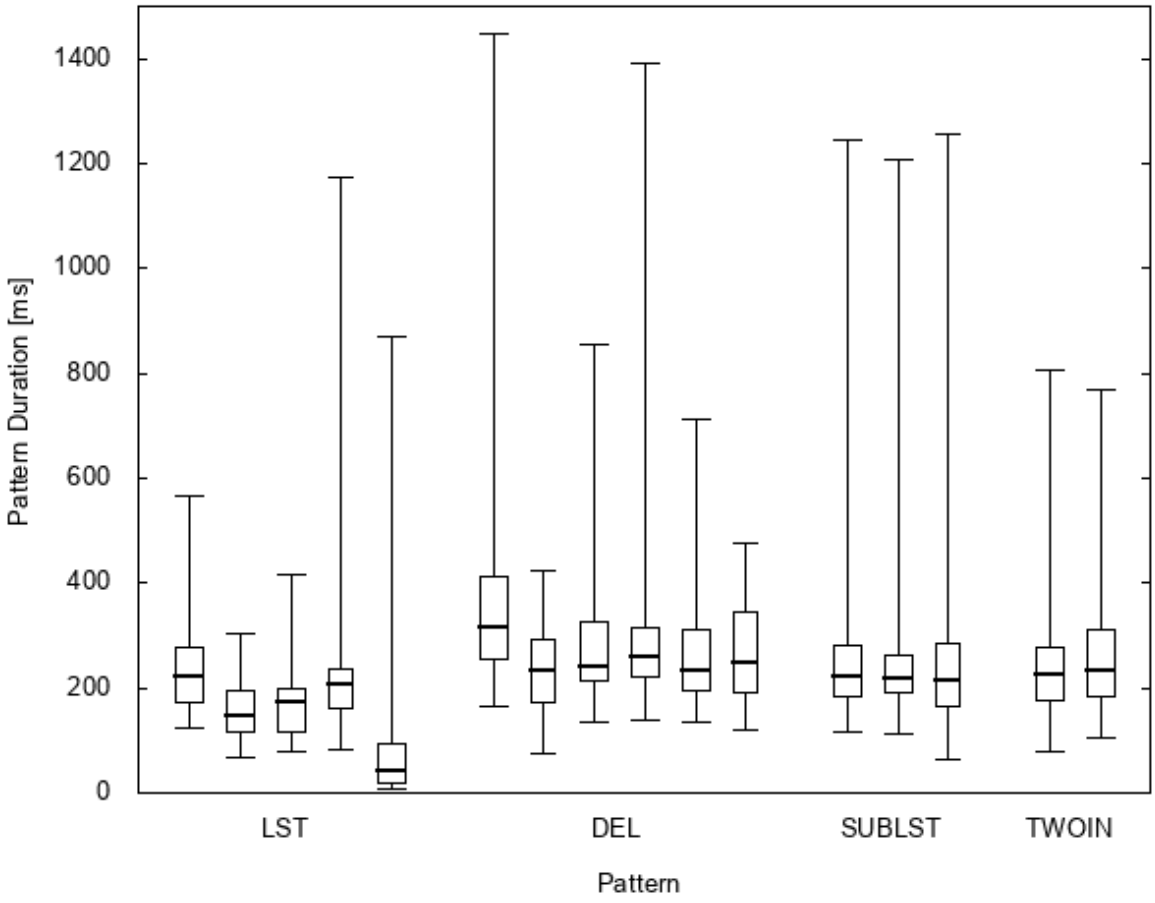


Figure 5.8: Example results for total sequence duration with n=250 measurements per pattern. The box plots use the same order as the interaction sequences in Figure 5.7.

Part III

Optimizing Microbenchmark Suites

Chapter 6

PeerJ

In this paper, we use time series database systems (TSDBs) as study objects. TSDBs are designed and optimized to receive, store, manage, and analyze time series data [54]. Time series data usually comprises sequences of timestamped data – often numeric values – such as measurement values. As these values tend to arrive in-order, TSDB storage layers are optimized for append-only writes because only a few straggler values arrive late, e.g., due to network delays. Moreover, the stored values are rarely updated as the main purpose of TSDBs is to identify trends or anomalies in incoming data, e.g., for identifying failure situations. Due to this, TSDBs are optimized for fast aggregation queries over variable-length time frames. Furthermore, most TSDBs support tagging which is needed for grouping values by dimension in queries. Taken together, these features and performance-critical operations make TSDBs a suitable study object for the evaluation of our approach. Examples of TSDBs include InfluxDB[1], VictoriaMetrics[2], Prometheus[3], and OpenTSDB[4].

6.1 Introduction

With the continuously increasing complexity of software systems, the interest in reliable and easy-to-use test and evaluation mechanisms has grown as well. While a variety of techniques, such as unit and integration testing, already exists for the validation of functional requirements of an application, mechanisms for ensuring non-functional requirements, e.g., performance, are used more sparingly in practice [8, 38]. Besides live testing techniques such as canary releases [142], developers and researchers usually resort to benchmarking, i.e., the execution of an artificially generated workload against the system under test (SUT), to study and analyze non-functional requirements in artificial production(-near) conditions.

[1] `https://www.influxdata.com`
[2] `https://victoriametrics.com`
[3] `https://prometheus.io`
[4] `http://opentsdb.net`

While application benchmarks are the gold standard and very powerful as they benchmark complete systems, they are hardly suitable for regular use in continuous integration pipelines due to their long execution time and high costs [16, 15]. Alternatively, less complex and therefore less costly microbenchmarks could be used, which are also easier to integrate into build pipelines due to their simpler setup [101]. However, a simple substitution can be dangerous: on one hand, it is not clear to what extent a microbenchmark suite covers the functions used in production; on the other hand, often only a complex application benchmark is suitable to evaluate complex aspects of a system. To link both benchmark types, we introduce the term *practical relevance* which refers to the extent to which a microbenchmark suite targets code segments that are also invoked by application benchmarks.

In this paper, we aim to determine, quantify, and improve the practical relevance of a microbenchmark suite by using application benchmarks as a baseline. In real setups, developers often do not have access to a (representative) live system, e.g., generally-available software such as database systems are used by many companies which install and deploy their own instances and, consequently, the software's developers usually do not have access to the custom installations and their production traces and logs. In addition, software is used differently by each customer which results in different load profiles as well as varying configurations. Thus, it is often reasonable to use one or more application benchmarks as the next accurate proxy to simulate and evaluate a representative artificial production system. The execution of these benchmarks for each code change is very expensive and time-consuming, but a light-weight microbenchmark suite that has proven to be practically relevant could replace them to some degree.

To this end, we analyze the called functions of a reference run, which can be (an excerpt from) a production system or an application benchmark, and compare them with the functions invoked by microbenchmarks to determine and quantify a microbenchmark suite's practical relevance. If every called function of the reference run is also invoked by at least one microbenchmark, we consider the respective microbenchmark suite as 100% practically relevant as the suite covers all functions used in the baseline execution. Based on this information, we devise two optimization strategies to improve the practical relevance of microbenchmark suites according to a reference run: (i) a heuristic for removing redundancies in an existing microbenchmark suite and (ii) a recommendation algorithm which identifies uncovered but relevant functions.

In this regard, we formulate the following research questions:

RQ 1 How to determine and quantify the practical relevance of microbenchmark suites?

Software source code in an object-oriented system is organized in classes and functions. At runtime, executed functions call other classes and functions, which leads to a program flow that can be depicted as a call graph. This graph represents which functions call which other

functions and adds additional meta information such as the duration of the executed function. If these graphs are available for a reference run and the respective microbenchmark suite, it is possible to compare the flow of both graphs and quantify to which degree the current microbenchmark suite reflects the use in the reference run, or rather the real usage in production. Our evaluation with two well-known time series databases shows that their microbenchmark suites cover about 40% of the functions called during application benchmarks. The majority of the functionality used by an application benchmark, our proxy for a production application, is therefore uncovered by the microbenchmark suites of our study objects.

RQ 2 How to reduce the execution runtime of microbenchmark suites without affecting their practical relevance?

If there are many microbenchmarks in a suite, they are likely to have redundancies and some functions will be benchmarked by multiple microbenchmarks. By creating a new subset of the respective microbenchmark suite without these redundancies, it is possible to achieve the same coverage level with fewer microbenchmarks, which significantly reduces the overall runtime of the microbenchmark suite. Applying this optimization as part of our evaluation shows that this can reduce the number of microbenchmarks by 77% to 90%, depending on the application and benchmark scenario.

RQ 3 How to increase the practical relevance within cost efficiency constraints?

If the microbenchmark suite's coverage is not sufficient, the uncovered graph of the application benchmark can be used to locate functions which are highly relevant for practical usage. We present a recommendation algorithm which provides a fast and automated way to identify these functions that should be covered by microbenchmarks. Our evaluation shows that an increase in coverage from the original 40% to up to 90% with only three additional microbenchmarks is theoretically possible. An optimized microbenchmark suite could, e.g., serve as initial and fast performance smoke test in continuous integration or deployment (CI/CD) pipelines or for developers who need a quick performance feedback for their recent changes.

After applying both optimizations, it is possible to cover a maximum portion of an application benchmark with a minimum suite of microbenchmarks which has several advantages. First of all, this helps to identify important functions that are relevant in practice and ensures that their performance is regularly evaluated via microbenchmarks. Instead of a suite that checks rarely used functions, code sections that are relevant for practical use are evaluated frequently. Second, microbenchmarks evaluating functions that are already implicitly covered by other microbenchmarks are selectively removed, achieving the same practical relevance with as few microbenchmarks as possible while reducing the runtime of the total suite. Furthermore, the effort for the creation of microbenchmarks is minimized because the microbenchmarks of the proposed functions will cover a large part of the application benchmark call graph

and fewer microbenchmarks are necessary. Developers will still have to design and implement performance tests, but the identification of highly relevant functions for actual operation is facilitated and functions that implicitly benchmark many further relevant functions are pointed out, thus covering a broad call graph. Ultimately, the optimized microbenchmark suite can be used in CI/CD pipelines more effectively: It is possible to establish a CI/CD pipeline which, e.g., executes the comparatively simple and short but representative microbenchmark suite after each change in the code. The complex and cost-intensive application benchmark can then be executed more sparsely, e.g., for each major release. In this sense, the application benchmark remains as the gold standard revealing all performance problems, while the optimized microbenchmark suite is an easy-to-use and fast heuristic which offers a quick insight into performance yet with obviously lower accuracy.

It is our hope that this study contributes to the problem of performance testing as part of CI/CD pipelines and enables a more frequent validation of performance metrics to detect regressions sooner. Our approach can give targeted advice to developers to improve the effectiveness and relevance of their microbenchmark suite. Throughout the rest of the paper, we will always use an application benchmark as the reference run but our approach can, of course, also use other sources as a baseline.

Contributions:

- An approach to determine and quantify the practical relevance of a microbenchmark suite.
- An adaptation of the Greedy-based algorithm proposed by [42] to remove redundancies in a microbenchmark suite.
- A recommendation strategy inspired by [138] for new microbenchmarks which aims to cover large parts of the application benchmark's function call graph.
- An empirical evaluation which analyzes and applies the two optimizations to the microbenchmark suites of two large open-source time series databases.

Paper Structure: After summarizing relevant background information in **??**, we present our approach to determine, quantify, and improve microbenchmark suites in Section 6.2. Next, we evaluate our approach by applying the proposed algorithms to two open-source time series databases in Section 6.3 and discuss its strength and limitations in Section 8.6. Finally, we outline related work in **??** and conclude in Section 6.5.

6.2 Approach

We aim to determine and quantify the practical relevance of microbenchmark suites, i.e., to what extent the functions invoked by application benchmarks are also covered by microbench-

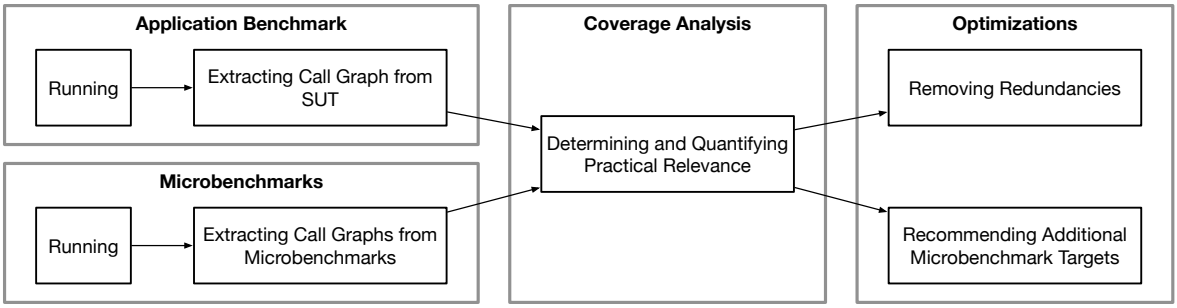


Figure 6.1: A study subject (system) is evaluated via application benchmark and its microbenchmark suite, the generated call graphs during the benchmark runs are compared to determine and quantify the practical relevance, and two use cases to optimize the microbenchmark suite are proposed.

marks. Moreover, we want to improve microbenchmark suites by identifying and removing redundancies as well as recommending important functions which are not covered yet.

Our basic idea is based on the intuition that, regardless of whether software is evaluated by an application benchmark or microbenchmark, both types evaluate the same source code and algorithms. Since an application benchmark is designed to simulate realistic operations in a production-near environment, it can reasonably be assumed that it can serve as a baseline or reference execution to quantify relevance in the absence of a real production trace. On the other hand, microbenchmarks are written to check the performance of individual functions and multiple microbenchmarks are bundled as a microbenchmark suite to analyze the performance of a software system. Both benchmark types run against the same source code and generate a program flow (call graph) with functions[5] as nodes and function calls as edges. We propose to analyze these graphs to (i) determine the coverage of both types to quantify the practical relevance of a microbenchmark suite, (ii) remove redundancies by identifying functions (call graph nodes) which are covered by multiple microbenchmarks, and (iii) recommend functions which should be covered by microbenchmarks because of their usage in the application benchmark. In a perfectly benchmarked software project, the ideal situation in terms of our approach would be that all practically relevant functions are covered by exactly one microbenchmark. To check and quantify this fact for a given project and to improve it subsequently, we propose the approach illustrated in Figure 8.2.

To use our approach, we assume that the software project complies with best practices for both benchmarking domains, e.g., [16, 49]. It is necessary that there is both a suite of microbenchmarks and at least one application benchmark for the respective SUT. The application benchmark must rely on realistic scenarios to generate a relevant program flow and must run against an instrumented SUT which can create a call graph during the benchmark execution.

[5]In the following, we exclusively refer to functions but our approach can similarly be used for methods and procedures depending on the SUT's programming language.

During that tracing run, actual measurements of the application benchmark do not matter. The same applies for the execution of the microbenchmarks, where it also must be possible to reliably create the call graphs for the duration of the benchmark run. These call graphs can subsequently be analyzed structurally to quantify and improve the microbenchmark suite's relevance. We will discuss this in more detail in the Section 6.2.1.

We propose two concrete methods for optimizing a microbenchmark suite: (1) An algorithm to remove redundancies in the suite by creating a minimal sub-set of microbenchmarks which structurally covers the application benchmark graph to the same extent (Section 6.2.2). (2) A recommendation strategy to suggest individual functions which are currently not covered by microbenchmarks but which are relevant for the application benchmark and will cover a large part of its call graph (Section 6.2.3).

6.2.1 Determining and Quantifying Relevance

After executing an application benchmark and the microbenchmark suite on an instrumented SUT, we retrieve one (potentially large) call graph from the application benchmark run and many (potentially small) graphs from the microbenchmark runs, one for each microbenchmark. In these graphs, each function represents a node and each edge represents a function call. Furthermore, we differentiate in the graphs between so-called project nodes, which refer directly to functions of the SUT, and non-project nodes, which represent functions from libraries or the operating system. After all graphs have been generated, the next step is to determine the function coverage, i.e., which functions are called by both the application benchmark and at least one microbenchmark.

Figure 6.2 shows an example: The application benchmark graph covers all nodes from node 1 to node 19 and has two entry points, node 1 and node 8. These entry points, when invoked, call other functions, which again call other functions (cycles are possible, e.g., in the case of recursion). Nodes in the graph can be both project functions of the SUT or functions of external libraries. There are also two microbenchmarks in this simple example, benchmark 1 and benchmark 2. While benchmark 1 only covers two nodes, benchmark 2 covers four nodes and seems to be more practically relevant (we will discuss this later in more detail).

To determine the function coverage, we iterate through all application benchmark nodes and identify all microbenchmarks which cover this function. As a result, we get a list of coverage sets, one for each microbenchmark, where each entry describes the overlap of nodes (functions) between the application benchmark call graph and the respective microbenchmark graph. Next, we count (i) all project-only functions and (ii) all functions which are called during the application benchmark and in at least one microbenchmark. Finally, we calculate two different coverage metrics: First, the *project-only* coverage of all executed functions in

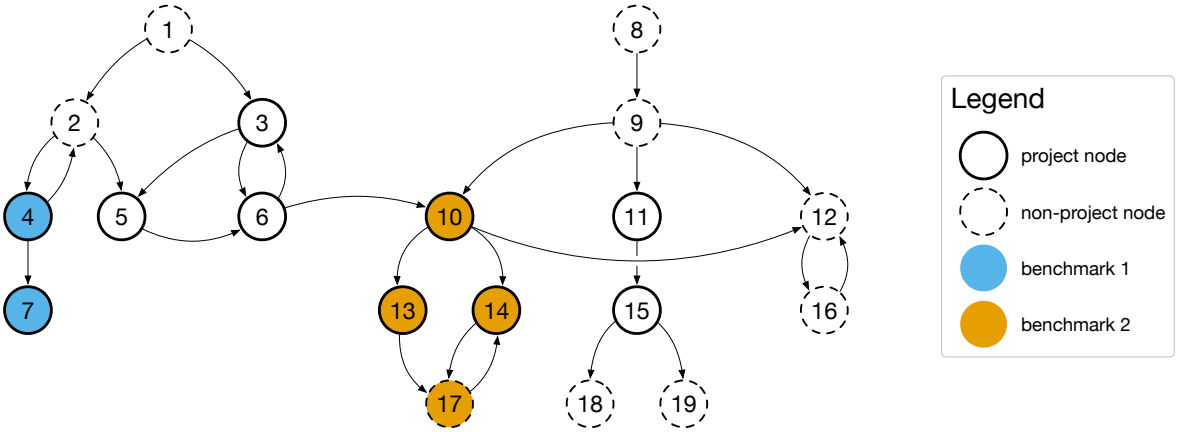


Figure 6.2: The practical relevance of a microbenchmark suite can be quantified by by relating the number of covered functions and the total number of called functions during an application benchmark to each other.

comparison to the total number of project functions in the application benchmark. Second, the *overall* coverage, including external functions.

For our example application benchmark call graph in Figure 6.2: $coverage_{project-only} = \frac{5}{10} = 0.5$ and $coverage_{overall} = \frac{6}{19} \approx 0.316$. Note that these metrics would not change if there would be a third microbenchmark covering a subset of already covered nodes, e.g., node 14 and node 17.

6.2.2 Removing Redundancies

Our first proposed optimization removes redundancies in the microbenchmark suite and achieves the same coverage level with fewer microbenchmarks. For example, the imaginary third benchmark mentioned above (covering nodes 14 and 17 in Figure 6.2) would be redundant, as all nodes are already covered by other microbenchmarks. To identify a minimal set of microbenchmarks, we adapt the Greedy algorithm proposed by [42] and rank the microbenchmarks based on the number of reachable function nodes that overlap with the application benchmark (instead of *all* reachable nodes as proposed in [42]), as defined in Algorithm 2.

After analyzing the graphs, we get coverage sets of overlaps between the application benchmark and the microbenchmark call graphs (input C). First, we sort them based on the number of covered nodes in descending order, i.e., microbenchmarks which cover many functions of the application benchmark are moved to the top (line 3). Next, we pick the first coverage set as it covers the most functions and add the respective microbenchmark to the minimal set (lines 4 to 8). Afterwards, we have to remove the covered set of the selected microbenchmark from all coverage sets (lines 9 to 11) and sort the coverage set again to pick the next microbenchmark. We repeat this until there are no more microbenchmarks to add (i.e., all

Algorithm 2: Removing redundancies in the microbenchmark suite.

```
   Input: C - coverage sets
   Result: minimalSet - Minimal set of microbenchmarks
 1 minimalSet ← ∅
 2 while |C| > 0 do
 3 |   C ← SortSets(C)
 4 |   largestCoverage ← RemoveFirst(C)
 5 |   if |largestCoverage| == 0 then
 6 |   |   return minimalSet
 7 |   end
 8 |   minimalSet ← minimalSet ∪ largestCoverage
 9 |   foreach set ∈ C do
10 |   |   set ← set \ largestCoverage
11 |   end
12 end
```

microbenchmarks are part of the minimal set and there is no redundancy) or until the picked coverage set would not add any covered functions to the minimal set (line 6).

In this work, we sort the coverage sets by their number of covered nodes and do not include any additional criteria to break ties. This could, however, result in an undefined outcome if there are multiple coverage sets with the same number of covered additional functions, but this case is a rare event and did not occur in our study. Still, including other secondary sorting criteria such as the distance to the graph's root node or the total number of nodes in the coverage set might improve this optimization further.

6.2.3 Recommending Additional Microbenchmark Targets

A well-designed application benchmark will trigger the same function calls in an SUT as a production use would. A well-designed microbenchmark for an individual function will also implicitly call the same functions as in production or during the application benchmark. In this second optimization of the microbenchmark suite, we rely on these assumptions to selectively recommend uncovered functions for further microbenchmarking. This allows developers to directly implement new microbenchmarks that will cover a large part of the uncovered application benchmark call graph and thus increase the coverage levels (see Section 6.2.1).

Similar to the removal of redundancies, we build on the idea of a well-known, greedy test case prioritization algorithm proposed by [138] to recommend functions for benchmarking that are not covered yet. In particular, we adapt Rothermel's *additional algorithm*, which iteratively prioritizes tests whose coverage of new parts of the program (that have not been covered by previously prioritized tests) is maximal. Instead of using the set of all covered methods by a microbenchmark suite, our adaptation uses the function nodes from the application benchmark that are not covered yet as greedy criteria to optimize for.

Algorithm 3: Recommending functions which are not covered by microbenchmarks yet.

Input: $\langle A, M, C \rangle$ - application benchmark CG (cg), microbenchmark cgs, coverage sets
Input: n - number of microbenchmarks to recommend
Result: R - Set of recommended functions to microbenchmark

```
1  R ← ∅
2  notCovered ← {a|a ∈ A ∧ IsProjectNode(a)} \ C^total
3  N ← ∅
4  while n > 0 do
5  |  foreach function f_a ∈ notCovered do
6  |  |  additionalNodes ← DetermineReachableNodes(f_a) ∩ notCovered
   |  |  N ← N ∪ {additionalNodes}
7  |  end
8  |  SortByNumberOfNodes(N)
9  |  largestAdditionalSet ← RemoveFirst(N)
10 |  if |largestAdditionalSet| == 0 then
11 |  |  return R
12 |  end
13 |  R ← R ∪ largestAdditionalSet[0]
14 |  n = n − 1
15 |  notCovered ← notCovered \ largestAdditionalSet
16 end
```

Algorithm 3 defines the recommendation algorithm. The algorithm requires as input the call graph from the application benchmark, the graphs from the microbenchmark suite, and the coverage sets determined in Section 6.2.1, as well as the upper limit n of recommended functions.

First, we determine the set of nodes (functions) in the application benchmark call graph which are not covered by any microbenchmark (line 2). Next, we determine the reachable nodes for each function in this set, only considering project nodes, and store the results in another set N (lines 5 to 7). To link back to our example graph in Figure 6.2, the resulting set for function 3 (neither covered by benchmark 1 nor 2) would be nodes 3, 5, and 6 (node 12 is not a project node and not part of the reachable nodes). Third, we sort the set N by the number of nodes in each element, starting with the set with the most nodes in it (line 8). If two functions cover the same number of project nodes, we determine the distance to the closest root node and select functions that have a shorter distance. If the functions are still equivalent, we include the number of covered non-project nodes as a third factor and favor the function with higher coverage. Finally, we pick the first element and add the respective function to the recommendation set R (lines 9 to 13), update the not covered functions (line 15), and run the algorithm again to find the next function which adds the most additional nodes to the covered set. Our algorithm ends if there are n functions in R (i.e., upper limit for

Table 6.1: Our Evaluation uses two open-source TSDBs written in Go as study objects.

Project	*InfluxDB*	*VictoriaMetrics*
GitHub URL	influxdata/influxdb	VictoriaMetrics/VictoriaMetrics
Branch / Release	`1.7`	`v1.29.4`
Commit	`ff383cd`	`2ab4cea`
Go Files	646	1284
Lines Of Code (Go)	193 225	462 232
Contributers	407	32
Stars	ca. 19 100	2500
Forks	ca. 2700	185
Microbenchmarks in Project	347	65
Extracted Call Graphs	288	62

recommendations reached) or if the function which would be added to the recommendation set R does not add additional functions to the covered set (line 11).

6.3 Empirical Evaluation

We empirically evaluate our approach on two open-source TSDBs written in Go, namely *InfluxDB* and *VictoriaMetrics*, which both have extensive developer-written microbenchmark suites. As application benchmark and, therefore, baseline, we encode three application scenarios in YCSB-TS[6]. On the other side, we run the custom microbenchmark suites of the respective systems.

We start by giving an overview of YCSB-TS and both evaluated systems (Section 6.3.1). Next, we describe how we run the application benchmark using three different scenarios (Section 6.3.2) and the microbenchmark suite (Section 6.3.3) to collect the respective set of call graphs. Finally, we use the call graphs to determine the coverage for each application scenario and quantify the practical relevance (Section 6.3.4) before removing redundancies in the benchmark suites (Section 6.3.5) and recommending functions which should be covered by microbenchmarks for every investigated project (Section 6.3.6).

6.3.1 Study Objects

To evaluate our approach, we need an SUT which comes with a developer-written microbenchmark suite *and* which is compatible with an application benchmark. For this, we particularly looked at TSDBs written in Go as they are compatible with the YCSB-TS application benchmark, and since Go contains a microbenchmark framework as part of its standard library. Furthermore, Go provides a tool called pprof[7] which allows us to extract the call graphs of

[6] `https://github.com/TSDBBench/YCSB-TS`
[7] `https://golang.org/pkg/runtime/pprof`

an application using instrumentation. Based on these considerations, we decided to evaluate our approach with the TSDBs *InfluxDB*[8] and *VictoriaMetrics*[9] (see Table 6.1).

YCSB-TS[10] is a specialized fork of YCSB [43] (an extensible benchmarking framework for data serving systems) for time series databases. Usually every experiment with YCSB is divided into a load phase which preloads the SUT with initial data, and a run phase which executes the actual experiment queries.

InfluxDB is a popular TSDB with more than 400 contributors and more than 19 000 stars on GitHub. *VictoriaMetrics* is an emerging TSDB (the first version was released in 2018) which has already collected more than 2000 stars on GitHub. Both systems are written in Go, offer a microbenchmark suite, and can be benchmarked using the YCSB-TS tool. However, there was no suitable connector for *VictoriaMetrics* in the official YCSB-TS repository, which we implemented based on the existing connectors for *InfluxDB* and Prometheus. Moreover, we also fixed some small issues in the YCSB-TS implementation. A fork with all necessary changes, including the new connector and all fixes, is available on GitHub[11].

6.3.2 Application Benchmark

Systems such as our studied TSDBs are used in different domains and contexts, resulting in different load profiles depending on the specific use case. We evaluated each TSDB in three different scenarios which are motivated in Section 6.3.2. The actual benchmark experiment is described in Section 6.3.2.

Scenarios

Depending on the workload, the call graphs within an SUT may vary. To consider this effect in our evaluation, we generate three different workloads based on the following three scenarios for TSDBs, see Table 6.2. All workload files are available on GitHub[12].

Medical Monitoring: An intensive care unit monitors its patients through several sensors which forward the tagged and timestamped measurements to the TSDB. These values are requested and processed by an analyzer, which averages relevant values for each patient once per minute and scans for irregularities once per hour. In our workload configuration, we assume a new data item for every patient every two seconds and deal with 10 patients.

We convert this abstract scenario description into the following YCSB-TS workload: With an evaluation period of seven days, there are approximately three million values in the range

[8] `https://www.influxdata.com`
[9] `https://victoriametrics.com`
[10] `https://github.com/TSDBBench/YCSB-TS`
[11] `https://github.com/martingrambow/YCSB-TS`
[12] `https://github.com/martingrambow/YCSB-TS/tree/master/workloads`

Table 6.2: We configured an application benchmark to use three different workload profiles.

Scenario	Medical Monitoring	Smart Factory	Wind Parks
Load			
Records	1 512 000	1 339 200	2 190 000
Run			
Insert	1 512 000	1 339 200	2 190 000
Scan	1680	1860	35 040
Avg	100 800	744	35 040
Count	0	744	0
Sum	0	2976	8760
Total	1 614 480	1 345 524	2 268 840
Other			
Duration	7 days	31 days	365 days
Tags	10	10	5

of 0 to 300 that are inserted into the database in total. Half of them, about 1.5 million, are initially inserted during the load phase. Next, in the run phase, the remaining records are inserted and the queries are made. In this scenario, there are about 100 000 queries which contain mostly AVG as well as 1.680 SCAN operations. Furthermore, the workload uses ten different tags to simulate different patients.

Smart Factory: In this scenario, a smart factory produces several goods with multiple machines. Whenever an item is finished, the machine controller submits the idle time during the manufacturing process as a timestamped entry to the TSDB tagged with the kind of produced item. Furthermore, a monitoring tool queries the average and the total amount of produced items once per hour and the accumulated idle time at each quarter of an hour. Finally, there are several manual SCAN queries for produced items over a given period. Our evaluation scenario deals with five different products and ten machines which on average each assemble a new item every 10 seconds. Moreover, there are 60 SCAN queries on average per day.

The corresponding YCSB-TS workload covers a 31-day evaluation period during which approximately 2.6 million data records are inserted. Again, we split the records in half and insert the first part in the load phase and insert the second part in parallel to all other queries in the run phase. In this scenario, we execute about 6000 queries in the run phase which include SUM, SCAN, AVG, and COUNT operations (frequency in descending order). Furthermore, the workload uses five predefined tags to simulate the different products.

Wind Parks: Wind wheels in a wind park send information about their generated energy as timestamped and tagged items to the TSDB once per hour. At each quarter of an hour,

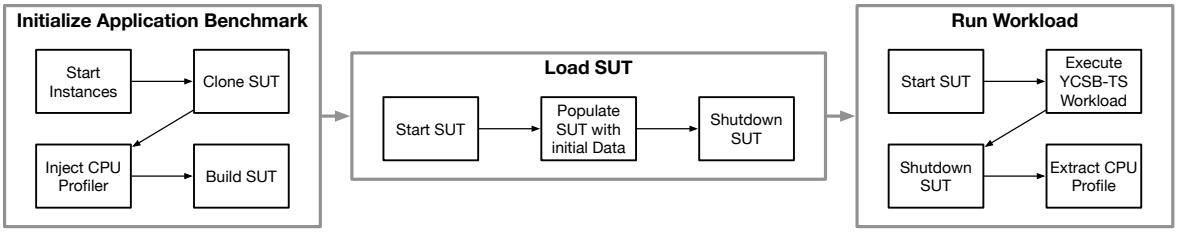


Figure 6.3: After initialization, the SUT is filled with initial data and restarted for the actual experiment run to clearly separate the program flow.

a control center scans and counts the incoming data from 500 wind wheels in five different geographic regions. Moreover, it totals the produced energy for every hour.

Translated into a YCSB-TS workload with 365 days evaluation time, this means about 4.4 million records to be inserted and five predefined tags for the respective regions. Again, we have also split the records equally between the load and run phase. In addition, we run about 80 000 queries, split between SCAN, AVG, and SUM (frequency in descending order).

Experiments

Similar to [24], each experiment is divided into three phases: initialize, load, and run (see Figure 6.3). During the initialization phase, we create two AWS t2.medium EC2 instances (2 vCPUs, 4 GiB RAM), one for the SUT and one for the benchmarking client in the eu-west-1 region. The setup of the client is identical for all experiments: YCSB-TS is installed and configured on the benchmarking client instance. The initialization of the SUT starts with the installation of required software, e.g., Git, Go, and Docker. Next, we clone the SUT, revert to a fixed Git commit (see Table 6.1) and instrument the source code to start the CPU profiling when running. Finally, we build the SUT and create an executable file.

During the load phase, we start the SUT and execute the load workload of the respective scenario using the benchmarking client and preload the database. Then, we stop the SUT and keep the inserted data. This way we can clearly separate the call graphs of the following run phase from the rest of the experiment.

Afterwards, we restart the SUT for the run phase. Since the source code has been instrumented, a CPU profile is now created and function calls are recorded in it by sampling while the SUT runs. Next, we run the actual workload against the SUT using the benchmarking client and subsequently stop the SUT. This run phase of the experiments took between 40 minutes and 18 hours, depending on the workload and TSDB. Note that the actual benchmark runtime is in this case irrelevant (as long as it is sufficiently long) since we are only interested in the call graph. Finally, we export the generated CPU profile which we use to build the call graphs.

After running the application benchmark for all scenarios and TSDBs, we have six application benchmark call graphs, one for each combination of scenario and TSDB.

6.3.3 Microbenchmarks

To generate the call graphs for all microbenchmarks, we execute all microbenchmarks in both projects one after the other and extract the CPU profile for each microbenchmark separately. Moreover, we set the benchmark execution time to 10 seconds to reduce the likelihood that the profiler misses nodes (functions), due to statistical sampling of stack frames. This means that each microbenchmark is executed multiple times until the total runtime for this microbenchmark reaches 10 seconds and that the runtime is usually slightly higher than 10 seconds (the last execution starts before the 10-second deadline and ends afterwards). Finally, we transform the profile files of each microbenchmark into call graphs, which we use in our further analysis.

6.3.4 Determining and Quantifying Relevance

Based on the call graphs for all scenario workloads and microbenchmarks, we analyze the coverage of both to determine and quantify the practical relevance following Section 6.2.1. Figure 6.4 shows the microbenchmark suite's coverage for each study object and scenario. For *InfluxDB*, the overall coverage ranges from 62.90% to 66.29% and the project-only coverage ranges from 40.43% to 41.25%, depending on the application scenario. For *VictoriaMetrics*, the overall coverage ranges from 43.5% to 46.74% and the project-only coverage from 35.62% to 40.43%. Table 6.3 shows the detailed coverage levels.

As a next step, we also analyze the coverage sets of all application benchmark call graphs to evaluate to which degree the scenarios vary and generate different call graphs. Figures 6.5a and 6.5b show the application scenario coverage as Venn diagrams for *InfluxDB* and *VictoriaMetrics* using project nodes only. Both diagrams show the same characteristics in general. All scenarios trigger unique functions which are not covered by other scenarios, see Table 6.2. For both SUTs, the Smart Factory scenario generates the smallest unique set of project-only nodes (29 unique functions for *InfluxDB* and 4 unique ones for *VictoriaMetrics*) and the Wind Parks scenario the largest one (134 functions for *InfluxDB* and 77 for *VictoriaMetrics*). Furthermore, all scenarios also generate a set of common functions which are invoked in every scenario. For *InfluxDB*, there are 464 functions of 920 in total (50.43%) and for *VictoriaMetrics* there are 341 functions of 603 in total (56.55%) which are called in every application scenario. Table 6.4 shows the overlap details.

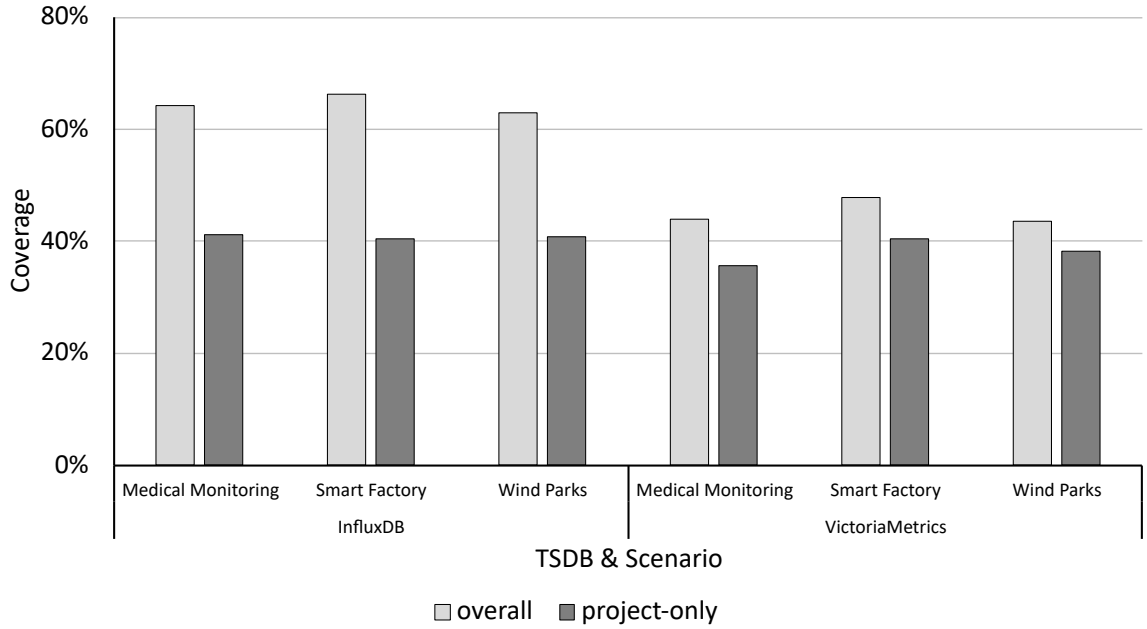
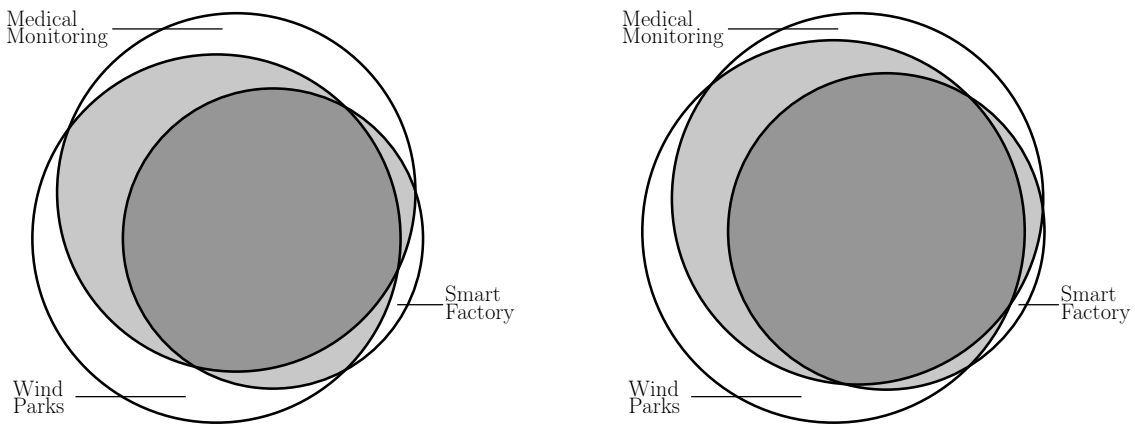


Figure 6.4: The project-only coverage is about 40% for both microbenchmark suites, leaving a lot potential room for improvement.

Table 6.3: All microbenchmarks together form a significantly larger call graph than the application benchmark (number of nodes) but, however, these by far do not cover all functions called during the application benchmarks (coverage).

Project	Scenario	Node Type	Number of Nodes		Coverage	
			App	Micro	Abs.	Rel.
InfluxDB	*Medical Monitoring*	*overall*	1838	3069	1180	64.20%
		project-only	737	1621	304	41.25%
	Smart Factory	*overall*	1504	3069	997	66.29%
		project-only	517	1621	209	40.43%
	Wind Parks	*overall*	1895	3069	1192	62.90%
		project-only	778	1621	318	40.87%
VictoriaMetrics	*Medical Monitoring*	*overall*	1573	1125	691	43.93%
		project-only	511	454	182	35.62%
	Smart Factory	*overall*	1238	1125	591	47.74%
		project-only	371	454	150	40.43%
	Wind Parks	*overall*	1600	1125	696	43.50%
		project-only	542	454	207	38.19%



(a) *InfluxDB* (b) *VictoriaMetrics*

Figure 6.5: All scenarios generate an individual call graph. Some functions are exclusively called in one scenario, many are called in two or all three scenarios.

Table 6.4: Pair-wise Overlap Between Different Scenarios

Project	Node Type	Scenario	*Medical Monitoring*	*Smart Factory*	*Wind Parks*
InfluxDB	*overall*	*Medical Monitoring*	same	1411 (76.77%)	1662 (90.42%)
		Smart Factory	1411 (93.82%)	same	1445 (96.08%)
		Wind Parks	1662 (87.70%)	1445 (76.25%)	same
	project-only	*Medical Monitoring*	same	468 (63.50%)	624 (84.67%)
		Smart Factory	468 (90.52%)	same	484 (93.62%)
		Wind Parks	624 (80.21%)	484 (62.21%)	same
VictoriaMetrics	*overall*	*Medical Monitoring*	same	1171 (74.44%)	1391 (88.43%)
		Smart Factory	1171 (94.59%)	same	1158 (93.54%)
		Wind Parks	1391 (86.94%)	1158 (72.37%)	same
	project-only	*Medical Monitoring*	same	356 (69.67%)	454 (88.84%)
		Smart Factory	356 (95.96%)	same	352 (94.88%)
		Wind Parks	454 (83.76%)	352 (64.94%)	same

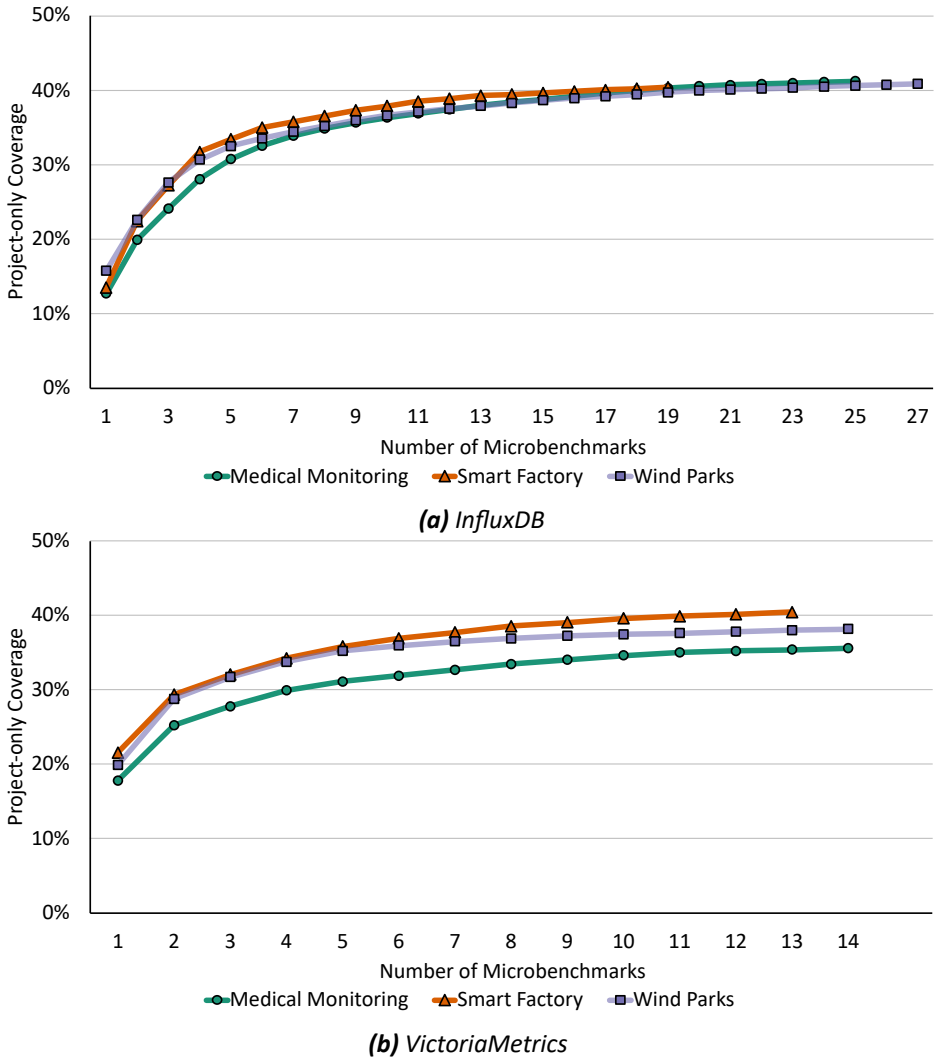


Figure 6.6: Already the first four microbenchmarks selecting by Algorithm 2 cover 28% to 31% for *InfluxDB* and 29% to 34% for *VictoriaMetrics* of the respective application benchmark's call graph.

6.3.5 Removing Redundancies

Our first optimization, as defined in Algorithm 2, analyzes the existing coverage sets and removes redundancy from both microbenchmark suites by greedily adding microbenchmarks to a minimal suite which fulfills the same coverage criteria. Figure 6.6 shows the step-by-step construction of this minimal set of microbenchmarks up to the maximum possible coverage.

For *InfluxDB* (Figure 6.6a), the first selected microbenchmark already covers more than 12% of each application benchmark scenario graph. Furthermore, the first four selected microbenchmarks are identical in all scenarios. Depending on the scenario, these already cover a total of 28% to 31% (with a maximum coverage of about 40% when using all microbenchmarks, see Table 6.3). These four microbenchmarks are therefore very important when covering a

large practically relevant area in the source code. However, even if all microbenchmarks selected during minimization are chosen and the maximum possible coverage is achieved, the removal of redundancies remains very effective. Depending on the application scenario, the initial suite with 288 microbenchmarks from which we extracted call graphs were reduced to a suite with either 19, 25, or 27 microbenchmarks.

In general, we find similar results for *VictoriaMetrics* (Figure 6.6b). Already the first microbenchmark selected by our algorithm covers at least 17% of the application benchmark call graph in each scenario. For *VictoriaMetrics*, the first four selected microbenchmarks also have similar coverage sets, there is only a small difference in the parametrization of one chosen microbenchmark. In total, these first four microbenchmarks cover 29% to 34% of the application benchmark call graph, depending on the scenario, and there is a maximum possible coverage between 35% and 40% when using the full existing microbenchmark suite. Again, the first four microbenchmarks are therefore particularly effective and already cover a large part of the application benchmark call graph. Moreover, even with the complete minimization and the maximum possible coverage, our algorithm significantly reduces the number of microbenchmarks: from 62 microbenchmarks down to 13 or 14 microbenchmarks, depending on the concrete application scenario.

Since each microbenchmark takes on average about the same amount of time (see Section 6.3.3), our minimal suite results in a significant time saving when running the microbenchmark suite. For *InfluxDB* it would take only about 10% of the original time and for *VictoriaMetrics* about 23% respectively. On the other hand, these drastic reductions also mean that many microbenchmarks in both projects evaluate the same functions. This can be useful under certain circumstances, e.g., if there is a performance degradation detected using the minimal benchmark suite and developers need to find the exact cause. However, given our goal of finding a minimal set of microbenchmarks to use as smoke test in a CI/CD pipeline, these redundant microbenchmarks present an opportunity to drastically reduce the execution time without much loss of information.

6.3.6 Recommending Additional Microbenchmark Targets

Our second optimization, the recommendation, starts with the minimal microbenchmark suite from above and subsequently recommends functions to increase the coverage of the microbenchmark suite and application benchmark following Algorithm 3. Figure 6.7 shows this step-by-step recommendation of functions starting with the current coverage up to a 100% relevant microbenchmark suite.

For *InfluxDB* (Figure 6.7a), a microbenchmark for the first recommended function would increase the coverage by 28% to 31% depending on the application scenario and the first three recommended functions are identical for all scenarios: (i) *executeQuery* runs a query

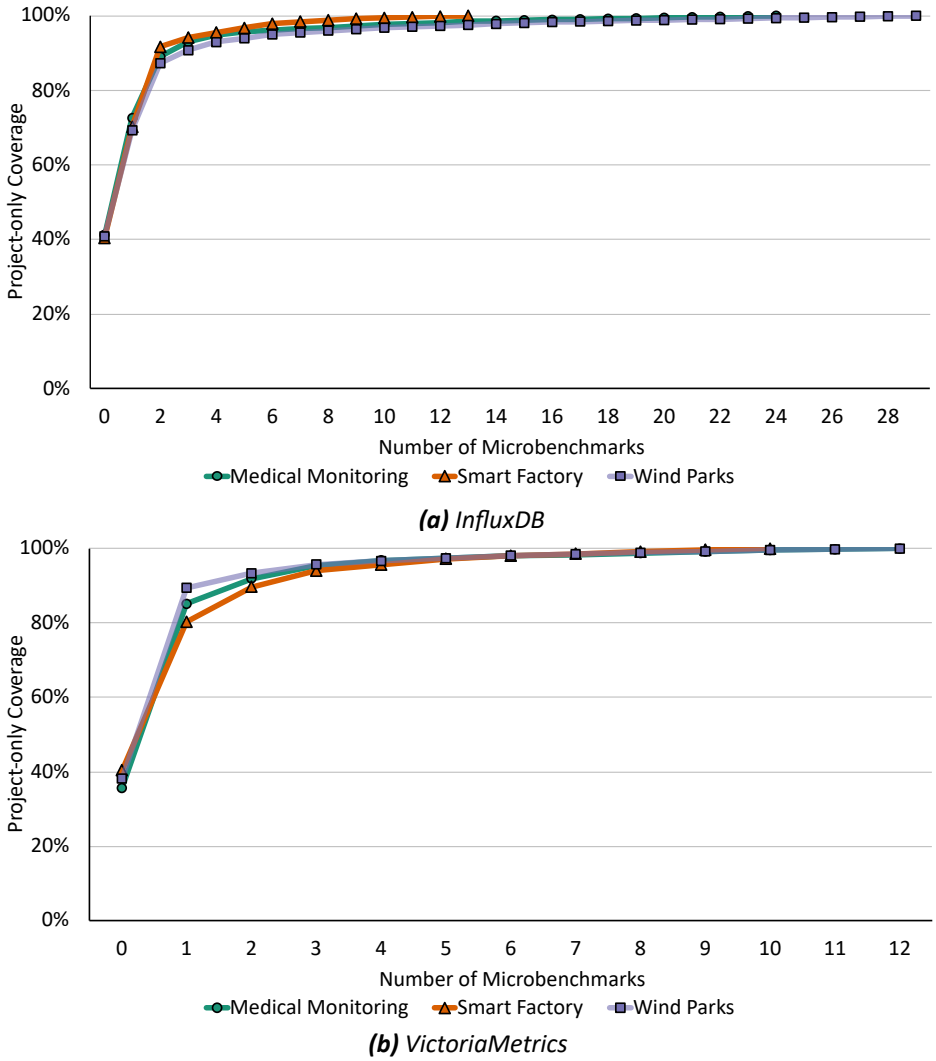


Figure 6.7: Already microbenchmarks of the first three recommended functions could increase the project-only coverage up to 90% to 94% for *InfluxDB* and 94% to 95% for *VictoriaMetrics*.

against the database and returns the results, (ii) *ServeHTTP* responds to HTTP requests, and (iii) *storeStatistics* writes statistics into the database. If each of these functions were evaluated by a microbenchmark in the same way as the application benchmark, i.e., resulting in the same calls of downstream functions and the same call graph, there would already be a total coverage of 90% to 94%. To achieve a 100% match, additional 10 to 26 functions must be microbenchmarked, depending on the application scenario and always under the assumption that the microbenchmark will call the function in the same way as the application benchmark does.

In general, we find similar results for *VictoriaMetrics* (Figure 6.7b). Already a microbenchmark for the first recommended function would increase the coverage by 39% to 51% and microbenchmarking the first three recommended functions would increase the coverage up

to a total of 94% to 95%. Again, these three functions are recommended in all scenarios, only the ordering is different. All first recommended functions are anonymous functions, respectively (i) an HTTP handler function, (ii) a merging function, and (iii) a result-related function. To achieve 100% project-only coverage, 10 to 14 additional functions would have to be microbenchmarked depending on the application scenario.

In summary, our results show that the microbenchmark suite can be made much more relevant to actual practice and usage with only a few additional microbenchmarks for key functions. In most cases, however, it will not be possible to convert the recommendations directly into suitable microbenchmarks (we discuss this point in Section 8.6). Nevertheless, we see these recommendations as a valid starting point for more thorough analysis.

6.4 Discussion

We propose an automated approach to analyze and improve microbenchmark suites. It can be applied to all application systems that allow the profiling of function calls and the subsequent creation of a call graph. This is particularly easy for projects written in the Go programming language as this functionality is part of the Go environment. Furthermore, our approach is beneficial for projects with a large code base where manual analysis would be too complex and costly. In total, we propose three methods for analyzing and optimizing existing microbenchmark suites but can also provide guidance for creating new ones. Nevertheless, every method has its limits and should not be applied blindly.

Assuming that the application benchmark reflects a real production system or simulates a realistic situation, the resulting call graphs will reflect this perfectly. Unfortunately, this is not always the case, because the design and implementation of a sound and relevant application benchmark has its own challenges and obstacles which we will not address here [16]. Nevertheless, a well-designed application benchmark is capable of simulating different scenarios in realistic environments in order to identify weak points and to highlight strengths. Ultimately, however, for the discussion that follows, we must always be aware that the application benchmark will never be a perfect representation of real workloads. Trace-based workloads [24] can help to introduce more realism.

Considering only function calls is imperfect but sufficient: Our approach relies on identifying the coverage of nodes in call graphs and thus on the coverage of function calls. Additional criteria such as path coverage, block coverage, line coverage, or the frequency of function executions are not considered and subject to future research. We deliberately chose this simple yet effective method of coverage measurement: (1) Applying detailed coverage metrics such as line coverage would deepen the analysis and check that every code line called by the application benchmark is at least once called by a microbenchmark. However, if the

different paths in a function source code are relevant for production and do not only catch corner cases, they should be also considered in the application benchmark and microbenchmark workload (e.g., if the internal function calls in the Medical Monitoring scenario would differ for female and male patients, the respective benchmark workload should represent female and male patients with the same frequency as in production). (2) As our current implementation relies on sampling, the probability that a function that is called only once or twice during the entire application benchmark or microbenchmark is called at the exact time a sample is taken is extremely low. Thus, the respective call graphs will usually only include practically relevant functions. (3) We assume that all benchmarks adhere to benchmarking best practices. This includes both the application benchmark which covers all relevant aspects and the individual microbenchmarks which each focus on individual aspects. This implies that if there is an important function, this function will usually be covered by multiple microbenchmarks which each generate a unique call graph with individual function calls and which therefore will all be included into the optimized microbenchmark suite. Thus, there will still usually be multiple microbenchmarks which evaluate important functions. (4) Both base algorithms ([42] and [138]) are standard algorithms and have recently been shown to work well with modern software systems, e.g., [111]. We therefore assume that a relevant benchmark workload will generate a representative call graph and argue that a more detailed analysis of the call graph would not improve our approach significantly. The same applies to the microbenchmarks and their coverage sets with the application benchmark where our approach will only work if the microbenchmark suite generates representative function invocations. Overall, the optimized suite serves as simple and fast heuristic for detecting performance issues in a pre-production stage but it is – by definition – not capable of detecting all problems: there will be false positives and negatives. In practice, we would therefore suggest to use the microbenchmark-based heuristic with every commit whereas the application benchmark will be run periodically; how often is subject to future research.

The sampling rate affects the accuracy of the call graphs: The generation of the call graphs in our evaluation is based on statistical sampling of stack frames at specified intervals. Afterwards, the collected data is combined into the call graph. However, this carries the risk that, if the experiment is not run long enough, important calls might not be registered and thus will not appear in the call graph. The required duration depends on the software project and on the sampling rate, i.e., at which frequency samples are taken. To account for this, we chose frequent sampling combined with a long benchmark duration in our experiments which makes it unlikely that we have missed relevant function calls.

Our approach is transferable to other applications and domains: We have currently evaluated our approach with two TSDBs written in the Go programming language, but we see

no major barriers to implementing our approach for applications written in other programming languages. There are several profiling tools for other programming languages, e.g., for Java or Python, so this approach is not limited to the Go programming language and is applicable to almost all software projects. Moreover, there are various other application domains where benchmarking can be applied which we also discuss in **??**. In this work, we primarily intend to present the approach and its resulting opportunities, e.g., for CI/CD pipelines. The transfer to other application domains and programming languages is subject to future research.

The practical relevance of a microbenchmark suite can be quantified quickly and accurately: Our approach can be used to determine and quantify the practical relevance of a microbenchmark suite based on a large baseline call graph (e.g., an application benchmark) and many smaller call graphs from the execution of the microbenchmark suite. On one hand, this allows us to determine and quantify the practical relevance of the current microbenchmark suite with respect to the actual usage: in our evaluation of two different TSDBs, we found that this is ~40% for both databases. On the other hand, this means that ~60% of the required code parts for the daily business are not covered by any microbenchmark, which highlights the need for additional microbenchmarks to detect and ultimately prevent performance problems in both study objects. It is important to note that the algorithm only includes identical nodes in the respective graphs; edges, i.e., which function calls which other function, are not considered here. This might lead to an effectively lower coverage if our algorithms selects a microbenchmark that only measures corner cases. To address this, it may be necessary to manually remove all microbenchmarks that do not adhere to benchmarking best practices before running our algorithm. In summary, we offer a quick way to approximate coverage and practical relevance of a microbenchmark suite in and for realistic scenarios.

A minimal microbenchmark suite with reduced redundancies can be used as performance smoke test: Our first optimization to an existing microbenchmark suite, Algorithm 2, aims to find a minimal set of microbenchmarks which already cover a large part of an application benchmark, again based on the nodes in existing call graphs. Our evaluation has shown that a very small number of microbenchmarks is sufficient to cover a large part of the potential maximum coverage for both study objects. Furthermore, it has also shown that the number of microbenchmarks in a suite can still be significantly reduced, even if we want to achieve the maximum possible coverage. Translated into execution time, this removal of redundancies corresponds to savings of up to 90% in our scenarios, which offers a number of benefits for benchmarking in CI/CD pipelines. A minimal microbenchmark suite could show developers a rough performance impact of their current changes. This enables developers to run a quick performance test on each commit, or to quickly evaluate a new version before starting a more complex and cost-intensive application benchmark. In this setup, the appli-

cation benchmark remains the gold standard to detect all performance problems while the less accurate optimized microbenchmark suite is a fast and easy-to-use performance check. Finally, it is important to note that the intention of our approach is not to remove "unnecessary" microbenchmarks entirely but rather to define a new microbenchmark suite as a subset of the existing one which serves as a proxy to benchmarking the performance of the SUT. Although our evaluation also revealed that many microbenchmarks benchmark the same code and are therefore redundant, this redundancy is frequently desirable in other contexts (e.g., for detailed error analysis).

The recommendations can not always be directly used: Our second optimization, Algorithm 3, recommends functions which should be microbenchmarked in order to cover a large additional part of realistic application flow in the SUT. Our evaluation with two open-source TSDBs has shown that this is indeed possible and that already with a small number of additional microbenchmarks a large part of the application benchmark call graph could be covered. However, our evaluation also suggests that these microbenchmarks are not always easy to implement, as the recommended functions are often very generic and abstract. Our recommendation should therefore mostly be seen as an initial point for further manual investigation by expert application developers. Using their domain knowledge, they can estimate which (sub)functions are called and what their distribution/ratio actually is. Furthermore, the application benchmark's call graph can also support this analysis as it offers insights into the frequency of invocation for all covered functions.

6.5 Conclusion

Performance problems of an application should ideally be detected as soon as they occur. Unfortunately, it is often not possible to verify the performance of every source code modification by a complete application benchmark for time and cost reasons. Alternatively, much faster and less complex microbenchmarks of individual functions can be used to evaluate the performance of an application. However, their results are often less meaningful because they do not cover all parts of the source code that are relevant in production.

In this paper, we determine, quantify, and improve this practical relevance of microbenchmark suites based on the call graphs generated in the application during the two benchmark types and suggest how the microbenchmark suite can be designed and used more effectively and efficiently. The central idea of our approach is that all functions of the source code that are called during an application benchmark are relevant for production use and should therefore be covered by the faster and more lightweight microbenchmarks as well. To this end, we determine and quantify the coverage of common function calls between both benchmark types, suggest two methods of optimization, and illustrate how these can be leveraged to improve

build pipelines: (1) by removing redundancies in the microbenchmark suite, which reduces the total runtime of the suite significantly; and (2) by recommending relevant target functions which are not covered by microbenchmarks yet to increase the practical relevance.

Our evaluation on two time series database systems shows that the number of microbenchmarks can be significantly reduced (up to 90%) while maintaining the same coverage level and that the practical relevance of a microbenchmark suite can be increased from around 40% to 100% with only a few additional microbenchmarks for both investigated software projects. This opens up a variety of application scenarios for CI/CD pipelines, e.g., the optimized microbenchmark suite might scan the application for performance problems after every code modification or commit while running the more complex application benchmark only for major releases.

In future work, we plan to investigate whether such a build pipeline is capable of detecting and catching performance problems at an early stage. Furthermore, we want to examine if a more detailed analysis of our coverage criteria on path or line level of the source code is feasible and beneficial. Even though there are still some limitations, we think that our automated approach is very useful to support larger software projects in detecting performance problems effectively, in a cost-efficient way, and at an early stage.

Chapter 7

TCC

7.1 Introduction

Performance issues in software systems should be identified and dealt with as early as possible. Besides a poor user experience, performance issues can also occupy additional resources and result in major fixing efforts which all imply unpredictable additional costs [164, 163, 40]. Thus, performance changes should ideally be detected by a Continuous Integration and Deployment (CI/CD) pipeline immediately after a code change is checked in [33, 60, 69, 160, 120, 85, 47].

For validating performance properties or adherence to Service Level Agreements (SLAs) such as a specific maximum latency or processing duration, software engineers often use benchmarking. Here, a system under test (SUT) is stressed with an artificial load, the requested values are measured, and these are then compared with the specification values, with the results of another alternative system, or with a previous version [16]. For detecting a performance change using benchmarking, there are two alternatives with different levels of granularity: first, using application benchmarks, where the SUT is set up including all related components and stressed in an environment that mimics the production conditions (e.g., a database system running on a virtual instance is stressed by a client software which mimics the requests of thousands of users for half an hour) [33, 69, 28, 24]; second, using microbenchmarks, which analyze individual functions at source code level and execute them repeatedly (e.g., a date conversion method is called a million times) [102, 101]. While the former method provides reliable results regarding application runtime implications, it is complex due to the setup and execution of the application benchmark. Microbenchmarks, on the other hand, are simpler and less complicated to execute. Nevertheless, microbenchmarks are unable to reliably detect all problems, as they do not take the integration of the respective functions or modules into the overall (production) system into account. For a single (standalone) component, however, they might be used as a proxy for a complex application benchmark.

Nevertheless, neither benchmarking technique is currently suited to be executed on every code change due to the extensive execution durations of several hours as well as the resulting costs [101, 152, 69, 160, 48]. Applying one of these two approaches to a large project with many application developers, hundreds of source code files, and multiple code changes per day would soon create a stack of benchmark tasks that would prevent fast-paced software development and integration of individual changes. Optimized microbenchmark suites containing only a small number of microbenchmarks can potentially solve this problem, because their execution is orders of magnitude faster, yet they also have to reliably detect application-relevant performance changes. A recent approach by [74] proposes to optimize microbenchmark suites by removing redundancies within the suite and only executing practically relevant microbenchmarks, i.e., microbenchmarks which cover source code parts that are frequently used in production (represented by an application benchmark as baseline).

In this paper, we investigate to which extent application benchmarks and microbenchmarks detect the same performance changes, and if we can use a smaller, optimized microbenchmark suite as a proxy for application benchmarks. To this end, we apply the optimization approach of [74] in real world examples using the open source systems *InfluxDB* and *VictoriaMetrics* as case studies and execute application benchmarks, optimized microbenchmark suites, and complete microbenchmark suites against 70 and 110 commits, respectively.

This paper makes the following main contributions:

- A comprehensive benchmarking dataset of application benchmarks and microbenchmarks for series of successive code changes which is available openly.[1]
- A performance change detection approach for the resulting time series data which differentiates between performance jumps and trends as well as potential and definite performance changes.
- An impact metric for quantifying the implications of individual microbenchmark results on application performance.
- Empirical evidence showing that less expensive and faster optimized microbenchmark suites can be used as a proxy for application benchmarks in certain situations.

For the two evaluated systems, the results show that performance changes can be reliably detected by running an application benchmark for less than one hour and that a reduced and optimized microbenchmark suite can detect the same changes with less than ten microbenchmarks. Our experiments identify nine true positive detections for optimized microbenchmark suites. Nevertheless, our study also shows the limitations of the optimization approach and

[1] `http://dx.doi.org/10.14279/depositonce-15532`

which type of performance issues cannot be identified with an optimized suite. First, the optimized suite does not detect the application performance changes if the microbenchmarks do not cover the practically relevant code sections. Second, performance changes related to the concrete runtime environment may not be detected. Third, if a performance change is detected by a microbenchmark, its impact on application performance is hard to predict. The optimized suites hence often identify false positives.

Derived from our findings, we envision that a good continuous benchmarking strategy should, e.g., combine a fast and relevant optimized microbenchmark suite and a well-designed application benchmark. While the optimized suite provides an early performance feedback for almost every code change, e.g., as part of a local build process or routine action which is triggered for each submitted code change, the regular runs of a well-designed application benchmark, e.g., once per day, ensures that the desired performance metrics are met. This allows developers to get quick performance feedback for each change, which allows them to adjust their changed code sections if necessary. Because optimized suites may not or even cannot find all problems, a daily application benchmark run serves as backup to reliably detect the remaining ones and report them the next day.

Removing Redundancies in Microbenchmark Suites

Especially in large software projects with hundreds or thousands of microbenchmarks, a complete suite execution can take several hours, making the evaluation for every code change impractical. Thus, there are various approaches to optimize microbenchmark suites [124] and also to detect performance problems using call graphs [120, 88]. Application call graphs represent the individual methods and functions of an application as nodes and their respective calls to each other as edges. For this study, we use an approach that removes redundancies in a microbenchmark suite based on an application benchmark call graph and includes only practically relevant microbenchmarks, i.e., microbenchmarks evaluating functions that are actually used in production [74] (see Figure 7.1). Here, a pre-recorded call graph from an application benchmark, which mimics the production behavior, is used as a baseline and compared with the call graphs of the respective microbenchmarks. The approach leverages a greedy heuristic and iteratively appends the microbenchmarks with the respective largest common overlap within the graphs to the optimized microbenchmark suite until no more new nodes are introduced. This excludes microbenchmarks which evaluate the same code sections and microbenchmarks which do not evaluate practically relevant code sections from the suite, thus reducing the number of benchmarks and significantly shortening the overall suite execution duration. The final optimized suite then solely consists of microbenchmarks that actually evaluate functions that are relevant in a production environment (which is represented by the application benchmark call graph).

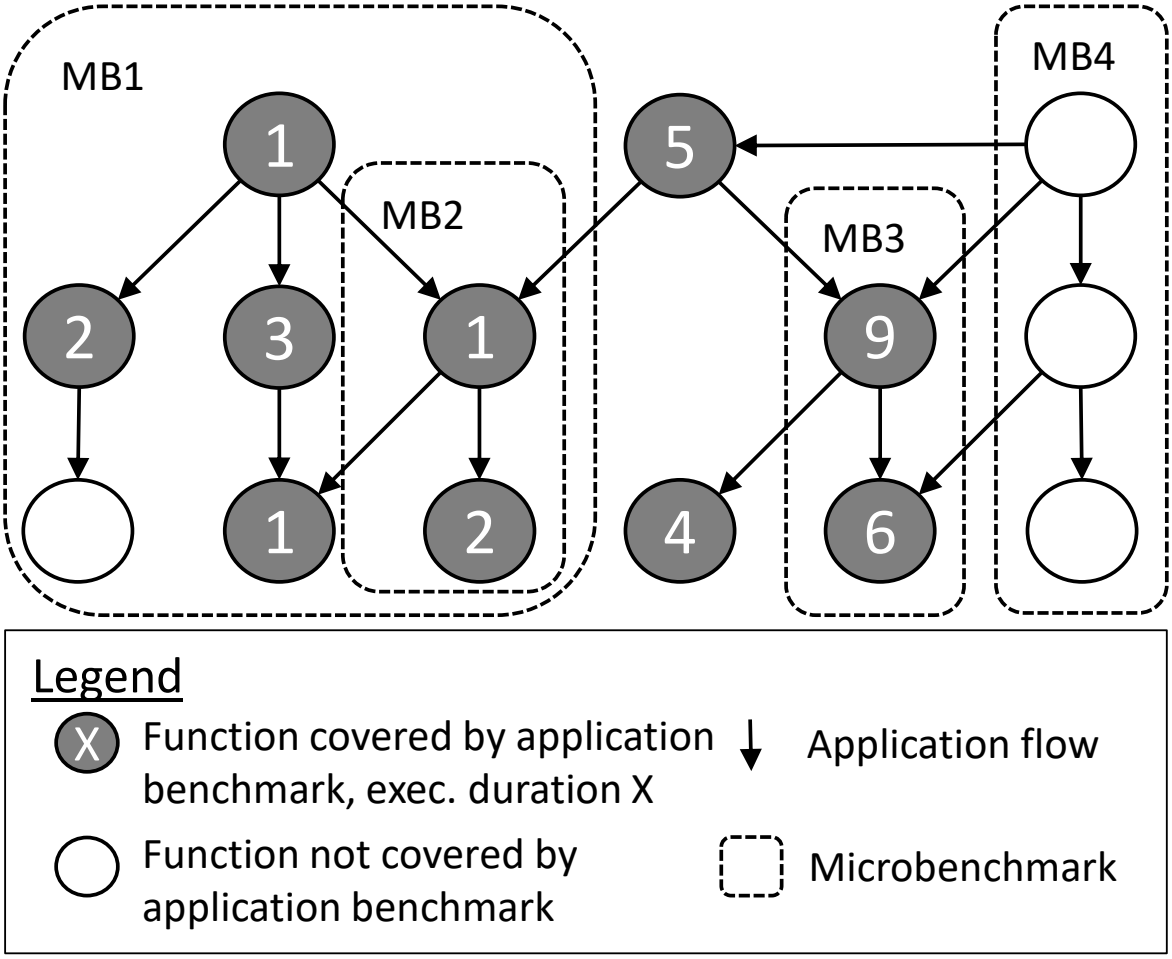


Figure 7.1: **Strategy for optimizing microbenchmark suites:** A suite containing microbenchmarks (MB) 1 and 3 would cover 80% of the application call graph. MB2 would not be included as all functions are already evaluated by MB1. MB4 does not evaluate any practically relevant functions.

7.2 Study Design

Applying and analyzing both benchmark types in realistic setups requires large software projects as study objects and a long commit history. Moreover, it must be possible to run a standardized application benchmark and there must be an extensive microbenchmark suite. Finally, to apply the removal of redundancies within the suite, it must be possible to trace the call graphs during the respective benchmark.

In the following sections and experiments, we therefore use two open source time-series database systems (TSDB) as study objects, namely *VictoriaMetrics* and *InfluxDB*. Both database systems are written in Go (which allows us to trace the call graphs without major modifications), come with extensive histories of code changes, and have comprehensive microbenchmark suites. Moreover, there are application benchmarks for both TSDBs. Because it is

infeasible to examine the entire development cycle over a period of several years, we examine a smaller sample of successive code changes, spanning several months, to simulate a realistic long-term use of all benchmarking techniques.

To study to which degree performance changes can be detected with an optimized microbenchmark suite, we initially run application benchmarks for both TSDBs to detect all application performance changes for the production environment. Next, we optimize the respective microbenchmark suites using recorded application benchmark call graphs as reference and run the optimized suite for every code change as well to check whether the optimized suites are capable of detecting the same performance changes. Finally, we also run the complete microbenchmark suites to quantify the degradation of detection quality caused by relying only on the optimized microbenchmark suite.

An optimized suite capable of detecting relevant performance changes can be embedded in a cloud-based CI/CD pipeline. To mimic this realistic setup as closely as possible, we therefore use cloud-based virtual machines (VMs) that are created and configured for every experiment run from scratch. To minimize performance variation between different instances and due to random effects such as noisy neighbors in the cloud environment, we adapt and apply recent best practices in each benchmarking discipline to acquire reliable measurement results: we use the Duet Benchmarking technique [32, 36] for application benchmarks and Randomized Multiple Interleaved Trials (RMIT) [2, 1] for execution of microbenchmarks. For all experiments, we use a hardware setup that is similar to the cloud experiment setups in related studies [101, 69, 71, 72, 29]. We run all experiments on e2-standard-2 Google Cloud instances in the europe-west3 region with 2 virtual CPUs and 8 GB RAM, local SSD storage, running Ubuntu 20.04 LTS.

7.2.1 Study Objects

TSDBs are optimized for storing sequences of time-stamped data, analyzing these sequences for specified time frames, and thus detecting trends and anomalies. Usually, values arrive in order and are appended to an existing time series whereas delayed values are inserted less frequently, e.g., due to network fluctuations. Furthermore, existing values are updated rarely and many TSDBs support grouping queries based on tagging [54].

Since the first version of *VictoriaMetrics* was released in 2018, more than 80 contributors have created more than $2,000$ files and made more than $2,800$ commits as of May 31, 2021. In our experiments, we study the most recent 70 commits at the time of running the experiments, i.e., between March 1 and May 31, 2021, which merge a pull request into the master branch,

Table 7.1: **Study objects and meta information.** Both TSDBs can be evaluated using application and microbenchmarks.

Project	*VictoriaMetrics*	*InfluxDB*
Go files	2,088	1,653
Lines of Go Code	742,191	520,716
Branch / Release	master	influx2.0
Start of Evaluation Period	Mar 1, 2021	Jan 1, 2021
End of Evaluation Period	May 31, 2021	May 14, 2021
Number of Commits	70	110
Number of Microbenchmarks	177	426 (109)

in more detail. In this period, the complete microbenchmark suite consists of 177 executable microbenchmarks in total, including all parameterized factors.

InfluxDB squashes individual fixes and features in single commits. *InfluxDB* version 2.0, which we investigate further, accumulated over 34,000 commits in more than 1,600 files from 422 contributors up to May 14, 2021. For our detailed study, however, we examine the most recent 110 commits at the experimentation phase in the time frame between Jan 1 and May 14, 2021. In contrast to *VictoriaMetrics*, the complete microbenchmark suite of *InfluxDB* does not remain constant for our evaluation period, but decreases from 426 microbenchmarks in the beginning to 109 at the end (we address and discuss this in Sections 7.3 and 8.6). Table 7.1 gives an overview of both study objects and the respective commits that we studied.

7.2.2 Application Benchmarks

For the application benchmarks, we use two cloud VMs in each experiment, one for the client sending the load and one for the respective SUT. Moreover, we adapt the Duet Benchmarking technique [32, 36] and set up two versions of the respective TSDB as Docker containers on the same VM: the base version and the variation (see Figure 7.2). We then use a pre-generated workload and start two benchmark clients simultaneously, one targeting the base version's port and one targeting the variation's port. Thus, the performance of the two variants can already be compared at three experiment repetitions, because both SUT versions are exposed to the same random factors at the same time.

The application benchmark workload is based on the *DevOps* use case in the Time Series Benchmark Suite (*TSBS*[2]), which in turn is based on the client *influxdb-comparisons*.[3] We use the *TSBS* client to benchmark *VictoriaMetrics* and *influxdb-comparisons* to benchmark

[2] `https://github.com/timescale/tsbs`
[3] `https://github.com/influxdata/influxdb-comparisons`

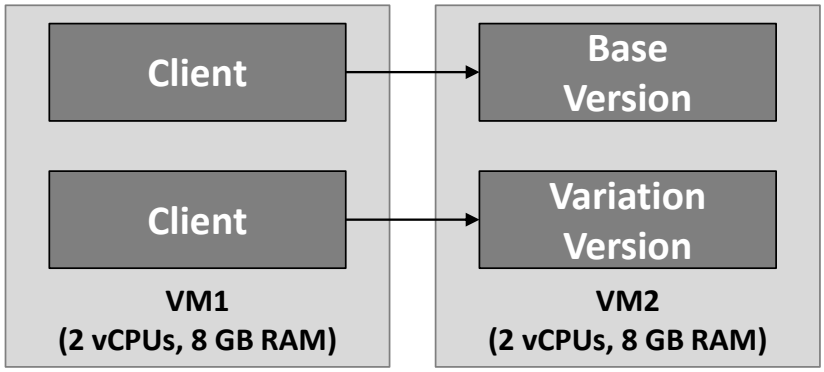


Figure 7.2: **Application benchmark setup.** To compare the performance of the first version (base) with the current version (variation) of the SUT in the respective evaluation period, we deploy both variants on the same VM and benchmark both simultaneously.

InfluxDB, but have extended both clients to report latency values of inserts and queries separately.[4] The *DevOps* use case simulates a server farm in which a specified number of servers sends utilization data (e.g., CPU and RAM) to the TSDB in a specified interval. After a first phase in which the data is inserted into the database, a second phase continues with simple queries, and a final phase with more complex group-by queries completes the experiment. We adjusted the number of simulated servers, the sending interval, the total simulated duration, and the number of respective queries to the specific SUT in a way that the client instance is below 50% utilization and the SUT instance is almost always fully utilized (see Table 7.2 for all workload details). Furthermore, we repeat each experiment at least three times using fresh VM instances to ensure reproducibility.

For our interpretation of results, we have to consider two aspects: first, as in almost all application benchmark experiments, the first few measurements must be considered as part of a warm-up phase and should be discarded [16]; second, due to the duet benchmarking, the last measurements should be removed as well. If one of the two evaluated versions has better performance, then the respective benchmark run will also finish earlier than the other one, which will then release resources on the experiment VM. The other container running the slower version then has access to additional resources and speeds up, which leads to wrong measurements. After some initial experiments comparing the first and last commit state of our study periods to determine the expected overall performance change, we chose to disregard the first 5% and the last 20% in the measurement series of each application benchmark run.

7.2.3 Microbenchmarks

To remove redundancies in the microbenchmark suite, we initially execute and trace an application benchmark against the first commit of the evaluation period and create an application

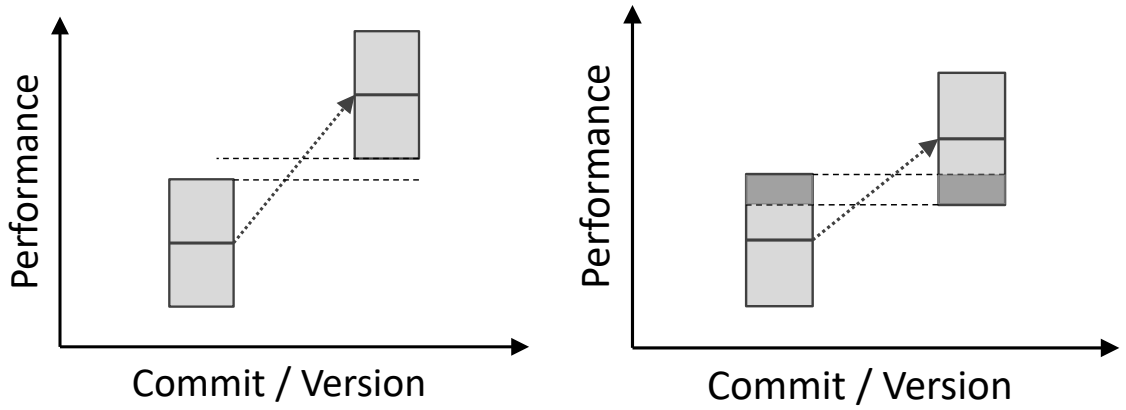
[4] `https://github.com/martingrambow/benchmarkStrategy`

Table 7.2: **Workload parameters.** The workload for each TSDB differs to ensure full utilization of the respective SUT.

Project	*VictoriaMetrics*	*InfluxDB*
Number of Simulated Servers	800	100
Sending Interval	$60s$	$60s$
Simulated Duration	$72h$	$168h$
Number of Insert Clients	4	10
Batch Size	400	60
Number of Batches	259,200	113,400
Number of Simple Queries	8,640	1,008
Number of Group-By Queries	1,440	168
Number of Query Clients	10	10

call graph, which serves as the reference for the optimization algorithm. Next, we execute and trace the full microbenchmark suite for the same commit and generate the call graph for each microbenchmark. With both inputs, the reference application graph and the microbenchmark graphs, we then determine the practical relevance of the microbenchmark suites, remove redundancies, decide which microbenchmarks to include in the optimized suite, and run the optimized microbenchmark suites for every commit in the respective evaluation period [74]. Finally, to rate the improvements and back-test the optimization, we also run the full microbenchmark suites for every 5th commit in the evaluation period. Running all microbenchmarks for every commit in practice is unrealistic due to the high costs of execution. An execution for every 5th commit is a trade-off between a very detailed analysis and a long execution time as well as the corresponding monetary costs. We believe that this is fine-grained enough to detect relevant changes and, in case there are anomalies, to further evaluate the relevant benchmark for the intermediate changes.

To mimic the usage of the microbenchmark suites in CI/CD pipelines, which compare a new version with an older commit state, we benchmark both versions on the same VM using RMIT time-shared execution [2, 1]. Here, to counteract infrastructure variation, we randomize the execution order of each suite and run each microbenchmark for both versions successively. To reduce the influence of the microbenchmarks on the performance of the following ones and to make sure that these effects are not systematic, we also randomly vary which microbenchmark version (base or variation) is executed first. Adapted from the configurations used by [99] and [40], we repeat each of our microbenchmark experiments three times on fresh VMs (instance run), run each suite three times (suite run), and call each benchmark five times (iteration) for one second each (duration). In total, thus, there are 45 measurements per microbenchmark per commit, each comprising many benchmark function calls.



(a) Definite Change (b) Potential Change

Figure 7.3: **Intensity classification.** Experiments in cloud environments can show a large variance. We therefore classify detected changes based on the 99% CI as definite or potential.

7.2.4 Analysis

We analyze the results of the respective benchmarks as follows:

Compared Versions

For all our experiments, we fix the base version to the first commit in the evaluation period and iterate over the commits as variation version. Thus, we always compare the current variation with the initial one. Nevertheless, because the results are transitive, the performance changes can be visualized as a pseudo-continuous graph.

Median Performance Change

To actually compare the versions, i.e., to decide which version performs better, we use the median value of all measurements. For the application benchmark, we use the median latency of all measured latency values for the respective query type. For microbenchmarks, we use the median execution duration of each microbenchmark of the 45 measurements (3 instance runs $*$ 3 suite runs $*$ 5 iterations). Finally, we calculate the relative change by comparing the median value of the base version with the median of the variation (e.g., if the median latency increases from $100ms$ to $110ms$, a query takes 10% longer).

Confidence Intervals

Moreover, to determine the confidence interval (CI) of a performance change, we use a bootstrapping methodology that implements hierarchical random re-sampling with replacement [90]. For the microbenchmarks, we draw $10,000$ random samples of 45 values each from

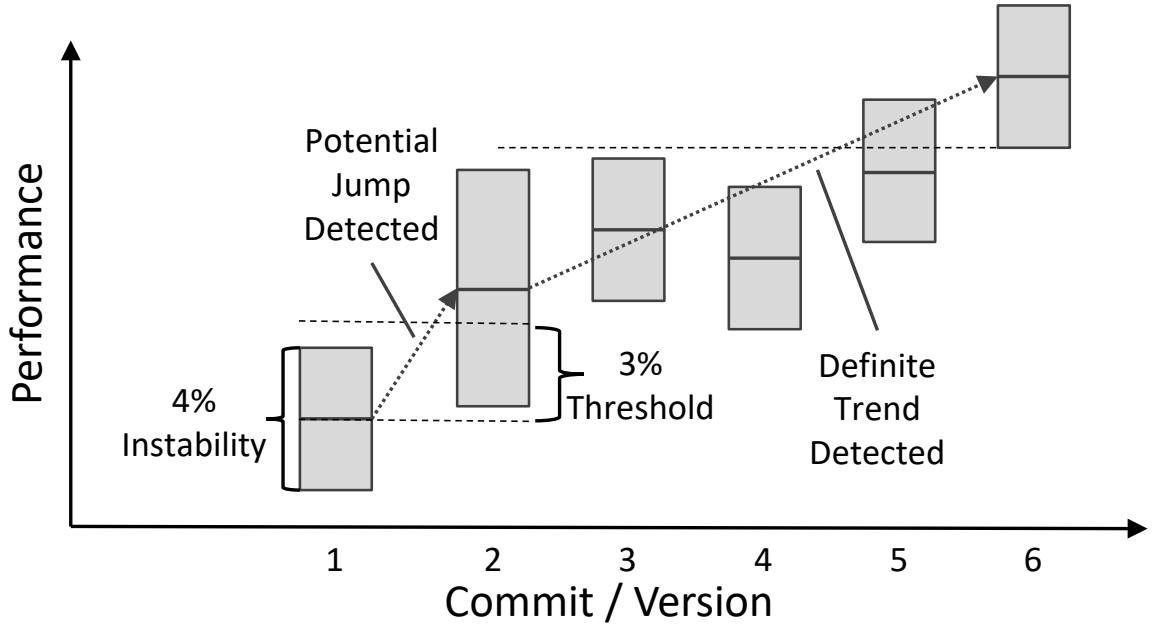


Figure 7.4: **Type classification.** While the jump detection identifies performance changes in two successive code changes, the trend detection considers a series of commits. The detection threshold adapts dynamically to the previous instability measurements. Thus, the detection threshold for the 3rd commit would increase because of the larger instability in the second one.

the measurements[5], determine the median value in each sample, and use the top and bottom $\alpha = 0.5\%$ of the resulting ordered set of the medians as the 99% CI. For the application benchmarks, we adapt the sample size to the number of requests for the respective request type, draw $10,000$ samples, and determine the CI in the same way.

Definite and Potential Performance Changes

A wide CI of an experiment implies that the concrete performance change of a (micro-)benchmark cannot be clearly quantified and that the individual benchmark is unstable. Thus, we refer to the width of a confidence interval as **instability**. The smaller this instability is, the better and more precisely it is possible to detect performance changes. On the other hand, a wide CI implies that it is only possible to detect large performance changes, because overlapping CIs of the respective experiments do not allow us to draw precise conclusions. Thus, we classify the detected performance changes as (99%) **definite** (no overlap) and **potential** (overlapping CIs) performance changes (see Figure 7.3).

Jump and Trend Detection

[5]Due to the replacement, values can be drawn multiple times and this precisely maps the actual distribution of the measurements.

We adapt two basic threshold-based algorithms by [69] to decide if we found a relevant performance change. For this study, we use (i) a **jump** detection algorithm to identify individual commits that introduce performance changes, and (ii) a **trend** detection algorithm to detect performance trends in a series of ten commits. In our study, however, we extend the static threshold and use a dynamic one that constantly adjusts to 75% of the instability of previous measurements (see Figure 7.4). Using 75% of the CI width is a trade-off between many false positives (50%) and potentially many false negatives (100%). Taking half the CI width could create false positive alarms in the change point detection, as the median performance change might just randomly fluctuate into the respective CI. For example, in Figure 7.4, using 50% of the instability in commit 1 corresponds to a 2% threshold, which leads to the median change in commit 2 to be exactly on the CI boundary of commit 1. Using the full CI width (100%) would only detect changes larger than the referenced instability, e.g., 4% for commit 2 in Figure 7.4. For both of our projects studied, using a 75% dynamic threshold provides a good balance between false-alarms and (potentially) undetected performance changes. Nevertheless, this threshold parameter is project-specific, especially if the median performance value is not centered in the respective CIs but is shifted to either side.

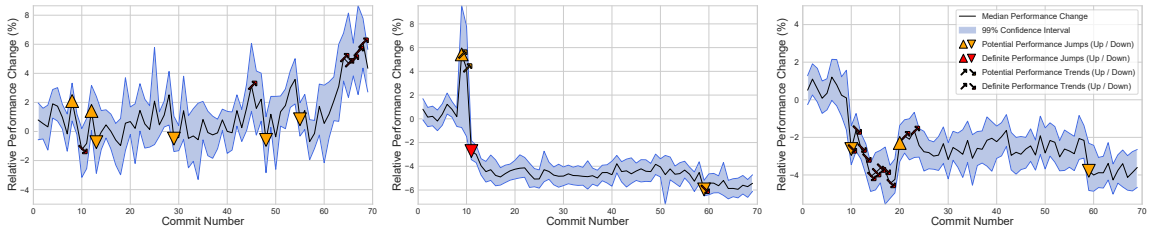
Code changes that stabilize the measurements will thus narrow the CI automatically (or the other way around) and random cloud fluctuations during the complete experiment series will automatically be considered in the analysis. Moreover, because we do not consider small performance changes as relevant in our evaluated projects, we also set a minimum threshold of 1%, similar to what best practice suggests [66]. In other projects, however, even smaller changes may also be relevant and this value would have to be adjusted. Finally, our dynamic detection algorithms require an initial threshold that is close to the expected value. If the difference is too high at first, either many false alarms would be triggered (small initial threshold) or relevant changes would not be detected (large initial threshold). Nevertheless, after the threshold has been continuously adjusted across several code changes (in our case ten), the detection mechanisms are adjusted to the respective instability.

Reference Impact

Running optimized microbenchmark suites using bootstrapping analysis and dynamic thresholds will detect multiple potential and definite performance changes for several microbenchmarks. These results, however, cannot be directly linked to a request type in the application benchmark or allow other direct conclusions. For example, if there is a definite performance drop of 5% in a microbenchmark, this drop cannot be directly linked to application performance (e.g., slower queries). To link a respective microbenchmark that detected a performance change to the application benchmark, we therefore use a **reference impact** value which is the sum of the execution durations of the overlapping functions in the reference application benchmark (see Figure 7.1). The key idea behind this is that a microbenchmark whose cov-

Table 7.3: **Result instability in A/A benchmarks.** All CIs are close to the 0% value but insert requests to *VictoriaMetrics* and queries to *InfluxDB* show a larger instability.

	Instability (99% CI)	
Project	*VictoriaMetrics*	*InfluxDB*
Inserts	6.27% [−2.95; 3.32]	0.88% [−0.71; 0.17]
Simple Queries	1.66% [−1.25; 0.41]	3.37% [−1.36; 2.01]
Group-By Queries	1.71% [−0.79; 0.92]	2.11% [−0.82; 1.29]



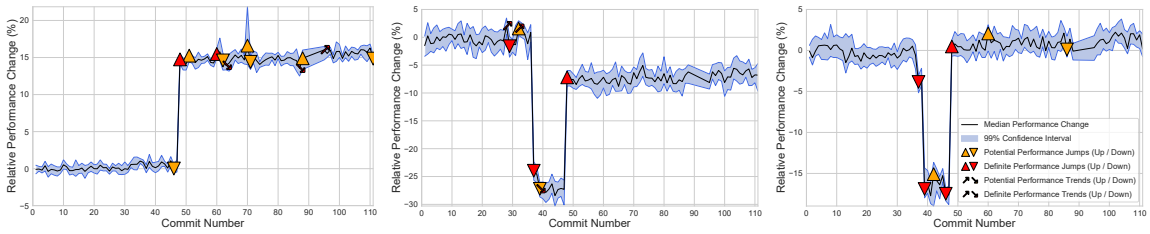
(a) Inserts (b) Simple Queries (c) Group-By Queries

Figure 7.5: **Application benchmark results for *VictoriaMetrics*, negative values show an improvement.** There is (i) a definite negative performance trend in the last commits of our evaluation period for inserts, (ii) a definite positive trend for both query types from commit 10 to 20 which is followed by (iii) a negative trend for group-by queries. Finally, there is (iv) a positive trend for both query types around commit 60.

ered functions in the application benchmark have a smaller total execution duration (e.g., 10 for MB1) will have less impact on overall application performance than another microbenchmark covering functions with a larger total application benchmark execution duration (e.g., 15 for MB3). We call this metric reference impact because it refers to the recorded application benchmark call graph and not to the performed microbenchmark experiment.

7.3 Results

We first report the results of the application benchmarks and use them as "ground truth" for the optimization algorithm (Section 7.3.1). Next, we investigate whether the optimized microbenchmark suite can detect the same performance changes with less effort (Section 7.3.2). To quantify the improvements and to verify that the complete microbenchmark suite is not a better proxy for detecting application performance changes, we also execute the complete suite for every 5th commit in our evaluation period (Section 7.3.3). Finally, we derive implications combining all information (Section 7.3.4).



(a) Inserts (b) Simple Queries (c) Group-By Queries

Figure 7.6: **Application benchmark results for *InfluxDB*, negative values show an improvement.** There are two large definite performance jumps at (i) commit 48 for all request types and at (ii) commit 37 for both query types. Moreover, there are several (potential) jumps and trends for all request types.

7.3.1 Application Benchmarks

First, to verify that our results are reliable and correct, we use five repeated A/A benchmarks which compare the first commit as the base version with itself as the variation version. Ideally there should not be any performance change, the CIs should be narrow, and around the 0% value. Table 7.3 reports the respective CIs derived from the bootstrapping method for each query type and SUT. All CIs straddle 0%, which implies no detected performance change. Nevertheless, especially the wide CI for inserts in *VictoriaMetrics* also implies that we can not reliably detect definite performance changes smaller than 6%. Based on the respective CI (reported in Table 7.3), we set the initial detection thresholds for both change point detection algorithms to $\approx$ 75% of the instability value or 1% (see Section 7.2.4, Jump and Trend Detection): except for the inserts in the case of *VictoriaMetrics* (5%) and the two query types in the case of *InfluxDB* (3% and 2%), we thus set all the initial detection threshold values to 1%.

Figures 7.5a to 7.5c show the relative performance history and the detected performance changes for insertions, simple queries, and group-by queries against *VictoriaMetrics*. A positive percentage value indicates that the respective request latency has increased.

Due to the non-deterministic setup of the internal data structure in *VictoriaMetrics*, there is a large instability for inserts. To overcome this obstacle, we split the initial insertion phase in half, copied the data from the base version container after the first half of insertions, and replaced the data in the variation container with this copy. The second part of the inserts is thus based on the same data structure and the non-determinism has a smaller effect on the result. To ensure that the queries are also based on the same underlying data structure, we repeat this step after the second half of inserts. Despite this instability, Figure 7.5a clearly shows that the insertions become significantly slower in the overall sequence of 70 commits and the change detection algorithm also detects this definite trend in the last ten commits. While simple queries improve during our study period by almost 6%, the performance history

of group-by queries shows more change points. Starting with commit 10, the performance of complex group-by queries improves initially, then degrades from commit 20 to 23, and improves again with commit 59.

Figures 7.6a to 7.6c show the detected performance jumps and trends for *InfluxDB* along with the measured relative performance history. Besides several detected (potential) jumps and trends, both query types are significantly improved through commit 37 and there is one major drop for all request types introduced with commit 48.

The corresponding commit message for commit 37, "feat(query/stdlib) [84]: promote schema and fill optimizations from feature flags", signals that a new feature successfully speeds up simple queries by around 25% and group-by queries by around 15%. This improvement, however, is reversed in commit 48 through the activation of profiling. The corresponding commit message for commit 48, "feat(http): allow for disabling pprof" [84], and the code changes indicate that this commit activates the costly profiling of Go by default. Looking at the total study period, simple queries are improved by about 5% by the end of the evaluation and group-by queries show a slight regression. Overall, we detect performance changes for all request types in both study objects.

7.3.2 Optimized Microbenchmark Suite

After generating the call graphs for both application benchmark and microbenchmark suite, we determine the practical relevance for both SUTs, and find an optimized microbenchmark suite based on the approach described in **??**, Removing Redundancies in Microbenchmark Suites.

Computing Optimized Suites

For *VictoriaMetrics*, the call graph analysis shows that 634 project functions are called during the application benchmark of which 314 are covered by microbenchmarks, thus indicating a practical relevance of $\approx 49\%$. In the next step, by removing redundancies in the suite, the same relevance is already achieved with only 17 microbenchmarks. Many of these microbenchmarks, however, cover only a few additional nodes (three or less) of the application benchmark. Thus, we use only the eight most relevant microbenchmarks for our further analysis, which corresponds to a 47% practically relevant microbenchmark suite.

Initial experiments with the microbenchmark suite of *InfluxDB* showed that there are major changes in the microbenchmark suite in the first 15 commits, which also affects our potentially optimized microbenchmark suite: a performance comparison of two versions of the microbenchmark is only possible if this microbenchmark is also present in both versions and has not been changed. Due to the fact that some of the most relevant microbenchmarks in

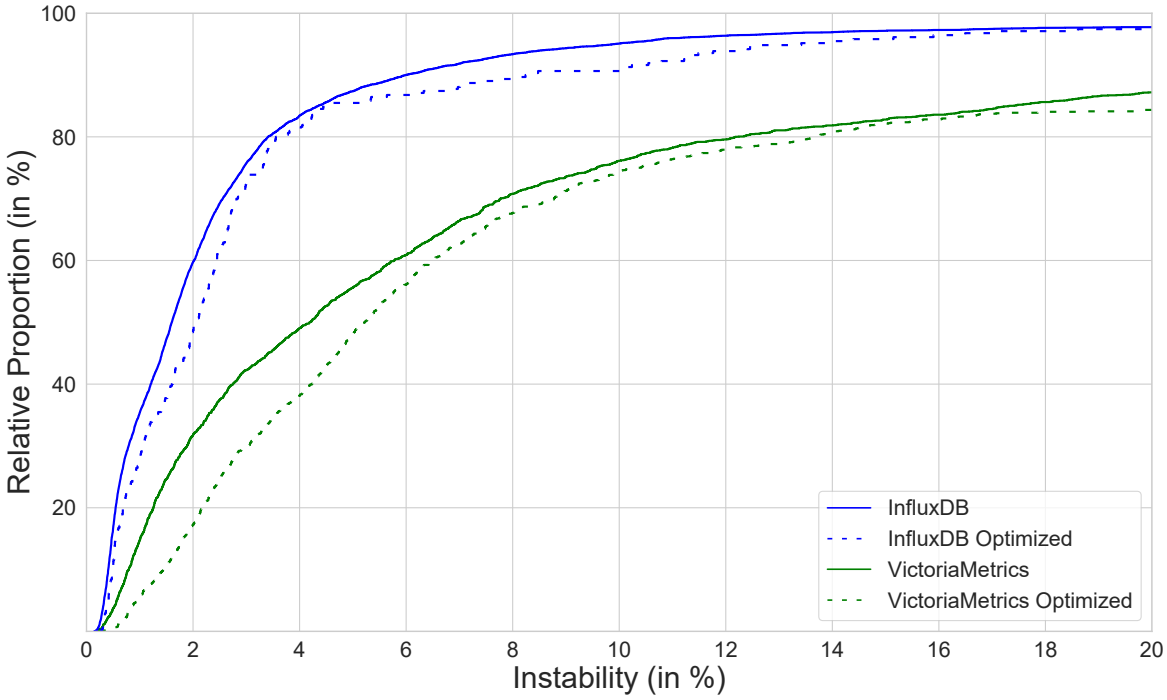


Figure 7.7: **Microbenchmark instability.** Approx. 80% of the microbenchmarks in the respective suites of *InfluxDB* show an instability of less than 4%. For *VictoriaMetrics*, however, only approx. 50% of the microbenchmarks show an instability of less than 4%.

the suite optimized for commit 1 are missing in commit 15, we set the base version to commit 15 and shorten the evaluation period for the microbenchmarks.

The practical relevance of *InfluxDB*'s complete microbenchmark suite is around 40% at commit 15 (269 of 660 nodes overlap). After computing the optimized suites without redundancies, many of the 26 proposed microbenchmarks add only a few additional nodes to the overlap (three or less). Thus, similar to *VictoriaMetrics*, we continue with only the 10 most relevant microbenchmarks ($\approx$ 36% practical relevance).

Determining Initial Detection Thresholds

Figure 7.7 shows cumulative distribution functions for the instabilities of all microbenchmark suites in A/A experiments. While the microbenchmarks of *InfluxDB* are very stable and $\approx$ 80% of the measurements have an instability below 4%, the performance of *VictoriaMetrics*'s microbenchmark suite(s) fluctuates more. Here, only $\approx$ 50% show an instability less than 4%. Thus, to cover $\approx$ 80% of the microbenchmarks instabilities in the respective suites with the initial detection threshold (see Section 7.2.4, Jump and Trend Detection), we choose a general starting threshold of 12% for *VictoriaMetrics* and 6% for *InfluxDB* for our change detection, i.e., only experiments exceeding these thresholds in the first code changes are classified as

performance changes. In the subsequent code changes, this threshold adapts to the respective microbenchmark's instability and the algorithm will detect changes more reliably.

Detected Changes for *VictoriaMetrics*

Running both optimized microbenchmark suites detects multiple potential and definite performance changes for several microbenchmarks. Figures 7.8 and 7.9 combine these detections with their corresponding reference impacts and evaluated performance metrics from the application benchmark. Ideally, any relevant performance change in the application benchmark (line chart in the upper part of the figure) should also be detected by a microbenchmark with a large reference impact (lower part of the figure). Nevertheless, because the microbenchmark suites in general only cover 47% and 36% of the application benchmark, we cannot expect to detect all changes.

Most of the detections in *VictoriaMetrics*' optimized microbenchmark suite originate from microbenchmarks with a reference impact of about 200 seconds in the application benchmark. With a total execution duration of the application benchmark of around 30 minutes (without setup), their covered functions are responsible for approximately 10% of the execution duration of the application benchmark.

The first three potential jumps are false positives due to the moving dynamic threshold. At the beginning of the evaluation period, the dynamic threshold is not yet adjusted to the observed instability. Therefore, we do not consider them further.

The first definite change in commit 10 and the next potential jumps and definite trend until commit 36 originate from the microbenchmark *BenchmarkAddMulti*, evaluating "a fast set for uint64" [159] using buckets, which indicates a relevance for inserts. The change detection of the application benchmark, on the other hand, also identifies a definite trend and faster inserts at commit 10. Visual analysis shows that all further detections of the microbenchmark are not clearly reflected in the performance of the inserts at *VictoriaMetrics*.

The detected changes from commit 41 to 46 refer to the microbenchmark *BenchmarkRowsUnmarshal*, which evaluates the unmarshalling of the Influx line protocol (which we use in our benchmarking client). Similarly to the detections before, these are not reflected in the application benchmark's detected changes.

The next definitive change correlates with another potential trend and jump at commit 59. The corresponding microbenchmarks *BenchmarkMergeBlockStreamsFourSourcesBestCase* and *BenchmarkMergeBlockStreamsFourSourcesWorstCase* merge multiple block streams and are related to queries. Although the correlated benchmarks have a longer average execution

time, both query types improve in the application benchmark at commit 59 (we discuss this in Section 8.6).

The microbenchmark change detection then raises signals at commit 64 and 65 for micro-benchmarks related to insert requests. These significant signals with a reference impact of 981 seconds (*BenchmarkStorageAddRows*) and 726 seconds (*BenchmarkIndexDBAddTSIDs*) are also significantly noticeable in the application benchmark.

Finally, the microbenchmarks *BenchmarkRowsUnmarshal* and *BenchmarkStorageAddRows* detect a definite change at commit 69. This change, however, is not visible in the application benchmark.

In total, the optimized suite detects four true positives (commits 10, 59, 64, and 65), but also raises false alarms for 17 commits. On the other hand, the optimized suite does not detect the negative performance trend of group-by queries starting with commit 20 (we discuss false negatives in more detail later).

Detected Changes for *InfluxDB*

The optimized suite of *InfluxDB* detects several potential and definite performance changes in five different microbenchmarks (see Figure 7.9).

The microbenchmark with the largest reference impact, *BenchmarkCreateIterator* (1013s impact), accounts for around 40% of the application benchmark's execution duration and benchmarks the creation of iterators for shard data items. This query-related benchmark identifies five potential performance changes for the commits 23, 30, 39, 52, and 84. While the commits 23, 30, and 52 are configuration-related code changes, which are unlikely to have an impact on application performance, commit 39 and 84 introduce larger changes. Commit 39 updates a flux dependency and this improvement is also visible in the application benchmark for both query types. Commit 84 adds a profiler option and modifies 68 lines in the *query.go* file. The query performance in the application benchmark, however, is not affected by this change.

The second most relevant microbenchmark is named *BenchmarkWritePoints* (942s impact) and it "benchmarks writing new series to a shard" [84], thus affecting insert requests. The first three detections are potential changes at commits 20, 41, and 65, which introduce minor features or fix small bugs. None of the three potential detected changes are visible in the application benchmark. The detected definite trend at commit 68 is caused by a minor configuration-related change. Neither this one, nor the changes from the previous commits (the root cause for the trend detection might also be in earlier commits), however, show any performance change in application performance. Next, commit 81 introduces an optimization which is identified as a potential jump and a definite trend. This optimization, however, does

not have any influence on the application performance. Commit 88 is a minor change but also updates the flux dependency. The application benchmark, on the other side, also detects a performance change for inserts. Finally, there is a minor fix at commit 97 identifying a potential change which is not relevant for application performance.

The third most relevant benchmark *BenchmarkParsePointsTagsUnSorted* (842*s* impact) benchmarks parsing of values and detects potential changes at commit 46 and 63 which are both also detected by the application benchmark. Commit 46 changes a default parsing option and this is also reflected in a potential improvement signal for inserts and a definite one for group-by queries in the application benchmark. The change introduced with commit 63 prevents a formatting of time strings in certain situations. This improvement is also detected as a potential improvement for inserts in the application benchmark.

The next microbenchmarks, *BenchmarkDecodeFloatArrayBlock* (251 seconds) and *BenchmarkIntegerArrayDecodeAllPackedSimple* (116 seconds), decode array blocks of float64 and integer values and have a significantly lower reference impact. The float benchmark detects one potential and nine definite changes, but only the changes at commit 39 (already identified by the most relevant microbenchmark), 46 (already identified by the third most relevant microbenchmark), and 60 are also detected by the application benchmark. Commit 60 fixes a cache-related race condition and this also impacts the performance of inserts and simple queries in the application benchmark. All other detections, however, are false positives. The least relevant integer benchmark detects changes for 17 commits. Here, five of the 17 detections (for commits 51, 63, 70, 87, and 94) correspond to the detections of the application benchmark for inserts and one matches a detection for grouping queries (commit 86). Nevertheless, because all changes introduce only minor features and smaller bug fixes, which are not related to any core functionality, we assume no direct correlation and consider all of them as false positives.

In total, the optimized suite detects five true positives (commits 39, 46, 60, 63, and 88), but also raises false alarms for 27 commits. Moreover, the optimized suite could not detect the two major performance changes at commit 37 and 48. While the performance change at commit 37 might not be detected because there is no microbenchmark covering the relevant code sections, the change at commit 48 can not be detected because it is related to the runtime environment.

7.3.3 Complete Microbenchmark Suite

Running the complete suite with hundreds of microbenchmarks for each commit in practice is unrealistic, as it is too expensive and time-consuming to do so. Nevertheless, to rate the improvement and better compare the optimization technique to this alternative, we execute the complete suite for every fifth commit. In contrast to the optimized suite, we cannot use

the reference impact to rank the results because many microbenchmarks do not overlap or only barely overlap with the reference application benchmark call graph. Thus, we aggregate the respective detection when interpreting the results, e.g., if ten microbenchmarks detect a definite change, this change might be practically relevant. Moreover, we also adapt the dynamic detection threshold to the evaluation of every fifth commit only and consider only the last three values (instead of 10).

Figures 7.10 and 7.11 show the application metrics (above) and the detections from the complete microbenchmark suite (below).

VictoriaMetrics's complete microbenchmark suite detects 91 changes in total and those cover all evaluated code changes. In particular, we observe that there are not only multiple detections for each commit, but that these detections are also often contradictory ([40] also report this phenomenon). Except for commit 46, where all detected changes are improvements, there are always at least one microbenchmark each measuring a performance degradation and improvement respectively.

The complete suite of *InfluxDB* detects 392 performance changes in total. Similarly, the suite detects contradictory performance changes at each commit and these cannot be matched with the metrics of the application benchmark.

In total, the complete suites detect hundreds of performance changes at high cost but only some of them are relevant. Without further information and criteria, such as an impact or relevance factor, it is impossible to identify the relevant ones.

7.3.4 Findings and Implications

In total, we examined 180 code changes in two open-source TSDBs using 540 application benchmark runs, 495 executions of optimized microbenchmark suites, and 102 runs of complete microbenchmark suites. These correspond to approximately $1,900$ hours of benchmark execution duration. Despite this vast number of experiments, we cannot demonstrate a clear benefit of using an optimized microbenchmark suite: while some benefits exist, there are limitations. Overall, our results help to better understand the trade-off between the execution of application benchmarks, optimized, and complete microbenchmark suites (see Tables 7.4 and 7.5).

Application Benchmarks

The setup of an automated application benchmark is complex and time-consuming. It requires scripts for starting the SUT and client instances, triggering and orchestrating the benchmark, collecting the measurements, and finally for analyzing the measurements. Running this complete pipeline took about $40min$ for *VictoriaMetrics* and $130min$ for *InfluxDB*

Table 7.4: **Benchmarking durations and prices.** Running complete microbenchmarks suites takes a lot of time while an optimized suite is faster and less expensive than an application benchmark.

Benchmark	*VictoriaMetrics*	*InfluxDB*
Application Benchmark	$\sim 40min$	$\sim 130min$
	($\sim$ \$0.13)	($\sim$ \$0.40)
Optimized Suite	$\sim 20min$	$\sim 40min$
	($\sim$ \$0.03)	($\sim$ \$0.06)
Complete Suite	$\sim 4h$	$\sim 11h$
	($\sim$ \$0.38)	($\sim$ \$1.05)

in our experiments, which corresponds to costs of about \$0.13 and \$0.40 per experiment repetition. Once set up, however, an application benchmark is a great tool for reliably detecting performance regressions or improvements in code changes. In our studied systems, this advantage can be illustrated especially with *InfluxDB*: A clear improvement caused by a new feature and a clear drop caused by a misconfiguration (which is impossible to detect using a microbenchmark) can be directly linked to specific commits. Furthermore, although small performance shifts between two successive commits may not be detected due to variability, an application benchmark can also be used to reliably detect performance trends. We can observe this characteristic especially for *VictoriaMetrics*: both query types show performance improvements between commit 10 and 20, but due to the large confidence intervals the changes cannot be directly connected to a single commit. Finally, our experiments also show that the Duet Benchmarking technique can not be applied everywhere without further modifications. Due to a non-deterministic characteristic of *VictoriaMetrics*, it is difficult to evaluate insert operations accurately.

Our experiments show that a well-designed application benchmark can reliably detect performance changes even in highly variable cloud environments. In our use cases, an application benchmark is relatively fast and cost-efficient, because we have chosen a rather simple setup with only two instances. In more complex setups using more complex application benchmarks, however, the price per benchmark will be higher, and the execution may also take longer. These more complex benchmarks include different load scenarios, involve many more instances and components, or evaluate the impact of changes in the environment, e.g., network fluctuations or (temporal) outages of individual components [152, 76]. Thus, depending on the frequency of code changes, we argue that an application benchmark should usually be scheduled to run daily, weekly, or after major code changes.

Table 7.5: **Result summary.** While application benchmark (app) and optimized microbenchmark suite (opti) detect a rather small number of performance changes, the complete suite (full) finds a lot more.

	Number of definite changes (+potential)					
Project	*VictoriaMetrics*			*InfluxDB*		
Benchmark	App	Opti	Full	App	Opti	Full
Jump up	0(+4)	5(+6)	23(+14)	4(+6)	6(+14)	72(+77)
Jump down	1(+5)	4(+4)	17(+15)	5(+6)	10(+10)	83(+67)
Trend up	8(+2)	1(+2)	6(+5)	3(+0)	4(+2)	20(+30)
Trend down	11(+0)	11(+2)	7(+4)	3(+0)	1(+1)	19(+24)

Complete Microbenchmark Suite

Running the complete microbenchmark suites of our evaluated projects takes around $4h$ for *VictoriaMetrics* and $11h$ for *InfluxDB*. Thus, evaluating one commit using one single experiment costs about \$0.38 for *VictoriaMetrics* and about \$1.05 for *InfluxDB*. For reliable measurement results, this single experiment should be run at least 3 times (concurrently), thus multiplying the cost. Furthermore, if a microbenchmark detects a performance change, the exact evaluation of the results is still hard due to the large number of experiments, instability of microbenchmarks, and it is often unclear to which degree a change affects the production environment and application-relevant metrics. For example, when running the complete suites for every fifth commit, we observe hundreds of performance changes in the microbenchmarks (see Table 7.5), but these are not reflected in the application-relevant benchmark metrics.

Running and evaluating a complete microbenchmark suite is usually expensive, takes a long time, is difficult to evaluate, and hardly yields any findings or findings that are difficult to derive. If code changes happen at intervals of minutes or hours, then this type of benchmark is only suitable for nightly or weekly performance evaluations. Nevertheless, a complete run can help to analyze a detected performance problem in more detail, help to isolate the issue, and find the root cause. Hence, it could be triggered whenever an application benchmark run has identified a performance change.

Optimized Microbenchmark Suite

The optimized microbenchmark suite without redundancies runs much faster (around $20min$ and \$0.04 for *VictoriaMetrics*; around $45min$ and \$0.07 for *InfluxDB*), is easier to evaluate and, if covering practically relevant parts, can detect the same performance changes that can also be detected by an application benchmark (see Table 7.5). Using optimized microbenchmark suites, we can identify four true positive detections for *VictoriaMetrics*) and five true positives for *InfluxDB*. Nevertheless, both optimized suites also raise false alarms, especially

through microbenchmarks with a low reference impact (which is part of the reason that the full suite detects so many false positives).

Running only practically relevant microbenchmarks significantly reduces the execution duration and also simplifies the analysis of the results. If the optimized suite covers a large portion of the practically relevant code sections, the suite can quickly detect performance changes and link them to specific commits at low cost. On the other hand, if performance changes relate to the runtime environment, integration, and or interaction of different application components, the microbenchmark suite cannot detect them. The profiling setting, which caused a significant performance drop in *InfluxDB* at commit 48, can not be found in the microbenchmarks because it was caused by a general configuration in the production(-like) environment. Another problem when using the microbenchmark suite are the benchmarks within the suite itself. If the suite changes often and especially if this involves the most relevant microbenchmarks with a large reference impact, then a continuous comparison is not possible and the optimized suite has to be re-determined periodically. In our experiments, this problem affects *InfluxDB* twice: once at the beginning of the evaluation period (commit 15); and once at commit 80. Thus, while an optimized benchmark suite can evaluate the performance several times a day, this benchmarking strategy should not be the only benchmarking approach used.

7.4 Discussion

Our experiments show that optimized microbenchmark suites can detect application performance changes in certain situations. While both micro- and application benchmarks may not be suitable for more detailed analysis of every commit in large projects with many code changes, they are still suitable for daily (or nightly) and weekly use as well as for a more detailed analysis after major changes. An optimized microbenchmark suite covering large practically relevant code parts can complement this by providing a fast performance feedback. Nevertheless, there are some limitations and possible extensions which we discuss in the following.

The Trade-off Between Cost and Accuracy

Within our experiments, we can produce reliable and reproducible results with three experiment repetitions. Nevertheless, several microbenchmarks show wide confidence intervals of more than 20% and are unstable. For each project, it is thus essential to find a good compromise between effort and cost on one side and accuracy and reliability on the other side.

Besides narrowing the CIs through additional experiment repetitions, which also increases cost accordingly, there are further optimizations by stopping benchmark runs under certain conditions or predicting unstable ones [6, 4, 78, 103, 99]. For example, stopping benchmarks as soon as there is a reliable finding might shorten the benchmark duration, excluding unstable

microbenchmarks might avoid unnecessary effort, or multiple smaller microbenchmarks might be more reliable and thus more cost-effective than a large unstable one. In our study, excluding a large unstable microbenchmark would just reduce the practical relevance by removing some microbenchmarks from the optimized suite of both study objects without adding equally relevant ones. Thus, this is subject to further research.

Changes in the Optimized Suite over Time

In our study, we use a fixed code state to optimize the microbenchmark suite, i.e., commit No. 0 for *VictoriaMetrics* and commit No. 15 for *InfluxDB*. This base version should not be changed as long as possible to generate a long measurement series for trend detection. Nevertheless, there are situations in both application benchmarks and microbenchmarks where this base version has to be reset and the optimization has to be repeated. Thus, the optimized suite cannot be considered static and has to be changed from time to time.

Both types of benchmark require a new base version when the benchmark itself is modified. Regarding the application benchmark this is, e.g., the case if the workload is no longer realistic and needs to be adjusted (e.g., the number of customers has doubled, which means twice as many requests in the production system). An adjusted application benchmark will imply a changed reference call graph and updated reference impact values.

Regarding the microbenchmarks, for example, there is the modification of the invocation parameters and that individual microbenchmarks might be removed (as can be seen in our experiments) or new ones might be implemented. Moreover, as every commit modifies the code that is evaluated by the microbenchmarks, the respective microbenchmark call graphs has to be updated as well. Both changes may require changes to the optimized suite as well.

Identifying False Alarms

In our experiments, both optimized suites detect nine true positive performance changes but also raise false alarms for 44 commits in total. These false alarms are caused, among other things, by measurement inaccuracies, but can also be caused by the approach reacting to changes in non-practically relevant functions. For example, if the performance of a non-practically relevant function degrades, but this function is also evaluated by a microbenchmark with large reference impact (e.g., the uncovered function of MB1 in Figure 7.1), then the performance of the microbenchmark will also degrade, even though this change has no impact on application performance. Confirming a detected change or spotting a false alarm would require to start an application benchmark in a realistic setup, which would imply corresponding costs. Thus, identifying false alarms in advance would be major improvement in further research.

Besides using the reference impact as additional classification for the reliability of detections, for example, the detected changes could be flagged and stored if the respective microbenchmark raised a false alarm. Using this history of changes, it might be possible to determine a reliability value for each microbenchmark which can be used to assess whether their detected performance change should be disregarded or not. A microbenchmark that successfully detected application performance changes in the past might also do this for future code changes.

Moreover, tagging microbenchmarks that are affected by a code change (i.e., only a fraction of the optimized suite), might also ease the result analysis. If one of those microbenchmarks raises an alarm, it is worth a detailed evaluation because there is a related code change. If a detected change is not raised by a tagged microbenchmark, it might be a false alarm. Furthermore, it might even be feasible to run only those microbenchmarks that cover modified functions.

Interpretation of Microbenchmark Performance Changes

To derive concrete implications from statements such as "microbenchmark A's performance has dropped by 5%", there are several options. In the optimal case, application developers know the underlying logic of the respective microbenchmark, can directly relate a detected change to the target functionality (e.g., the request type), and rate the impact on the production environment. This is, however, not realistic, especially for large projects. We suggest interpreting and storing the detected changes as warnings which will trigger an application benchmark to verify the overall system performance and or to use them to support root cause analysis when a future application benchmark shows significant performance changes and the originating code change needs to be identified.

A strict policy that, for example, rejects a commit when a performance issue is detected by a microbenchmark would in many cases be incorrect. In our experiments, for example, a microbenchmark detected that the merging of block streams takes longer for *VictoriaMetrics* in commit 59 while the application benchmark observed faster queries. Because this corresponding code update merges 8 new features and fixes 6 bugs, we can not identify the exact reason for this phenomenon due to its complexity, but we can find two possible explanations. First, even though merging the block streams takes longer because more data is processed, the query latency decreases because fewer streams need to be merged, thus resulting in fewer calls to the respective function while running the application benchmark. Second, while merging streams takes longer, another feature is introduced that improves the query latency but is not covered by the optimized microbenchmark suite (yet). In such a scenario, the feature leading to the improvement might be the dominant code change while the microbenchmarks can only detect the less relevant degradation covered by the suite.

Implications for Production

Our extensive experiments using two time series database systems show many interesting aspects when running optimized microbenchmark suites. Nevertheless, there is no general (micro-) benchmarking strategy that can simply be applied to every project. The strategy needs to be determined individually for each project and depends, among other things, on the general development progress, the number of code changes per day, the production environment, the expected load, and the impact of a potential performance issue.

Optimized microbenchmark suites can be a helpful tool for large projects with multiple developers, a large code base, and many code changes per day (e.g., our studied time series database systems). Here, a detailed performance evaluation of every code change is not possible, but optimized suites can help to evaluate these changes well enough, i.e., covering practically used code sections. In smaller projects it can also be useful to save costs. For example, a cost-intensive execution of application benchmarks for each code change possibly can be replaced with the optimized suite while the application benchmark is, e.g., executed only weekly, for every 10th commit, or for each major change.

As the concrete parameter values have to be defined individually for each project, we recommend analyzing past code changes and running some trial benchmarks first, e.g., to estimate variances. Based on these results, it is then possible to derive concrete values such as (micro-) benchmark frequency, detection thresholds, or actions in case of a detected performance change.

Limitations of Application Benchmarks

Application benchmarking offers the possibility of placing the evaluated system in any requested situation. From examining increased usage during holiday season to studying the effects of component failure, application benchmarks can be implemented for many situations. Nevertheless, they effectively use an artificial load and do not run on the production system, which also has limitations. For example, the actual production load may not match the load assumed by the benchmark, resulting in different results and implications. In addition, not all use cases can provide a second environment that can be used for benchmarks. For example in IoT scenarios, it is hardly possible to maintain a second identical building with the same smart home devices just for testing and benchmarking purposes. Alternatively and or complementary to application benchmarks, among others, application performance monitoring, gradual roll-outs, or dark launches can be used to detect performance changes in production.[6] While benchmarking is used before deploying to production and does not affect real users, live testing techniques such as gradual roll-outs are applied in the real production environment. Ideally, there should be a holistic combination of approaches from both phases, before and during live deployment.

[6] If it is possible to record call graphs in the production environment, these graphs can also be used as (a real) reference to compute the optimized suite.

Further Improvements and Research Directions

In our experiments, we benchmark successive code changes in the commit history of two large open-source TSDBs and analyzed them in detail. Nevertheless, our findings can not be generalized to all systems. There might be combinations of microbenchmark suites, SUTs, application benchmarks and their evolution over time in which microbenchmarks can detect performance regressions with neither false positives nor false negatives. We believe that our findings are representative for most real world combinations. This is based on the intuition that microbenchmark suites are unlikely to always have full code coverage of the SUT and that the functions studied by individual microbenchmarks may or may not have significant effects on the execution duration of application benchmarks. Overall, our study motivates further research on the computation, usage, and advantages of optimized microbenchmark suites.

7.5 Conclusion

Both microbenchmarks and application benchmarks can be used in CI/CD pipelines to ensure that performance and non-functional requirements of software systems are met in every release. For large and complex projects with multiple code changes per day, however, both are too costly to examine every single code change in detail.

In this paper, we explored to which degree application-relevant performance changes, such as an increase in query latency, can also be detected by optimized microbenchmark suites. For this, we use the commit history of *InfluxDB* and *VictoriaMetrics* and study them by running extensive benchmark experiments with application benchmarks, using an application coverage-based optimization strategy for microbenchmark suites, and running complete suites, we could show that this is indeed possible with some limitations. As we discovered, the approach requires that existing microbenchmarks cover (almost) all application-relevant code sections but still results in both false negative and false positive detections. Thus, optimized suites cannot be a proxy for a regular application benchmark but can provide a fast performance feedback at low cost after code changes in certain situations. For example, an optimized suite could be routinely run for (almost) every code change to detect most performance problems, while a more reliable application benchmark could be used as a daily backup process to detect the missed ones.

Overall, our findings open opportunities for practitioners to include new continuous benchmark steps in CI/CD pipelines and to shorten the execution times of established ones. Our results motivate further studies using other systems, developing further microbenchmark selection algorithms, and fine-tuning parameters to cost-efficiently improve the benchmark accuracy.

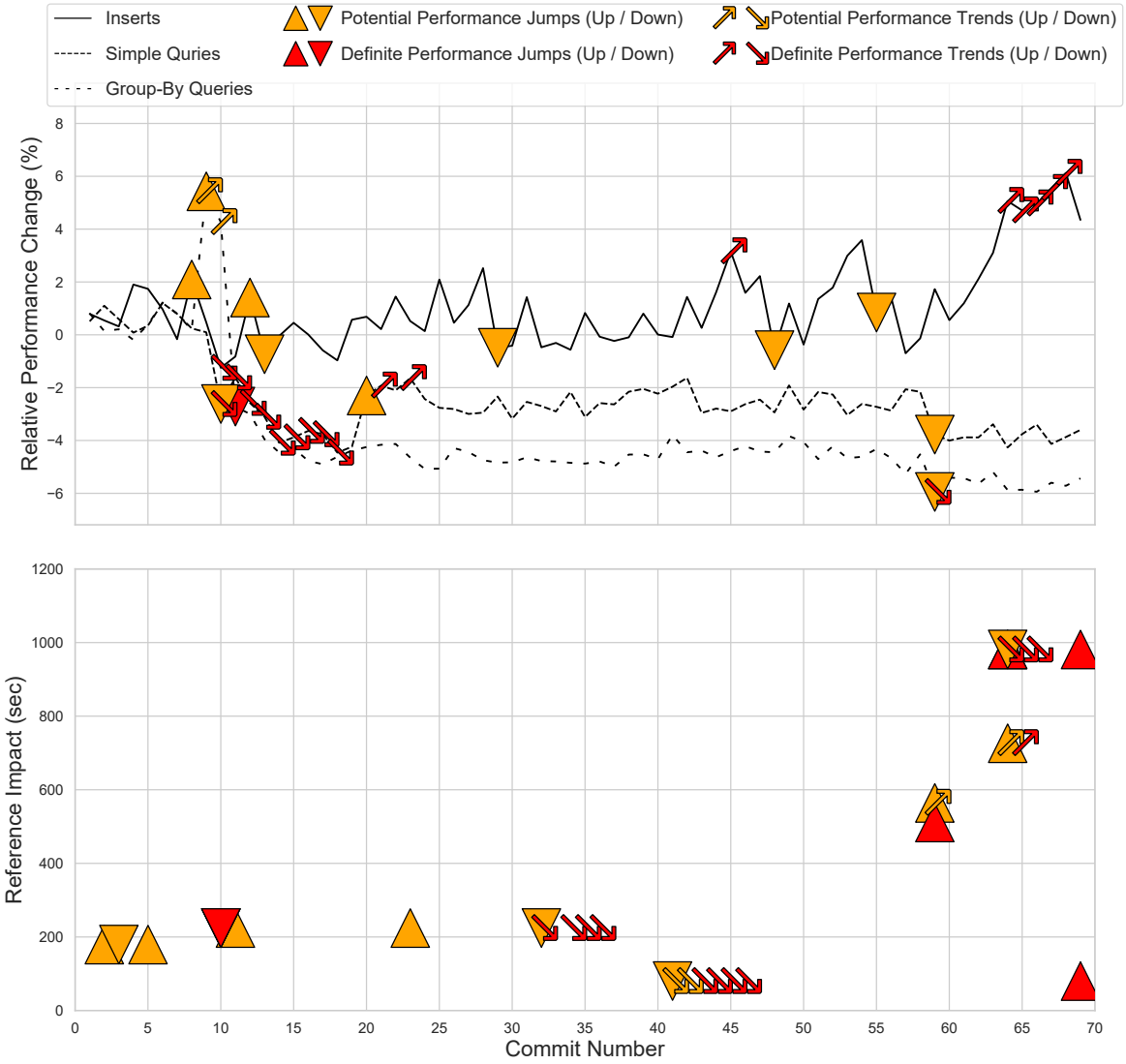


Figure 7.8: **Results from the optimized suite for *VictoriaMetrics*.** The upper part shows the results of the application benchmark and its detections while the lower part shows the detections from the microbenchmarks (higher means more likely impact on application performance). The optimized suite detects an insert-related change at commit 10, a performance change for queries at commit 59, and slower inserts at commit 64 and 65.

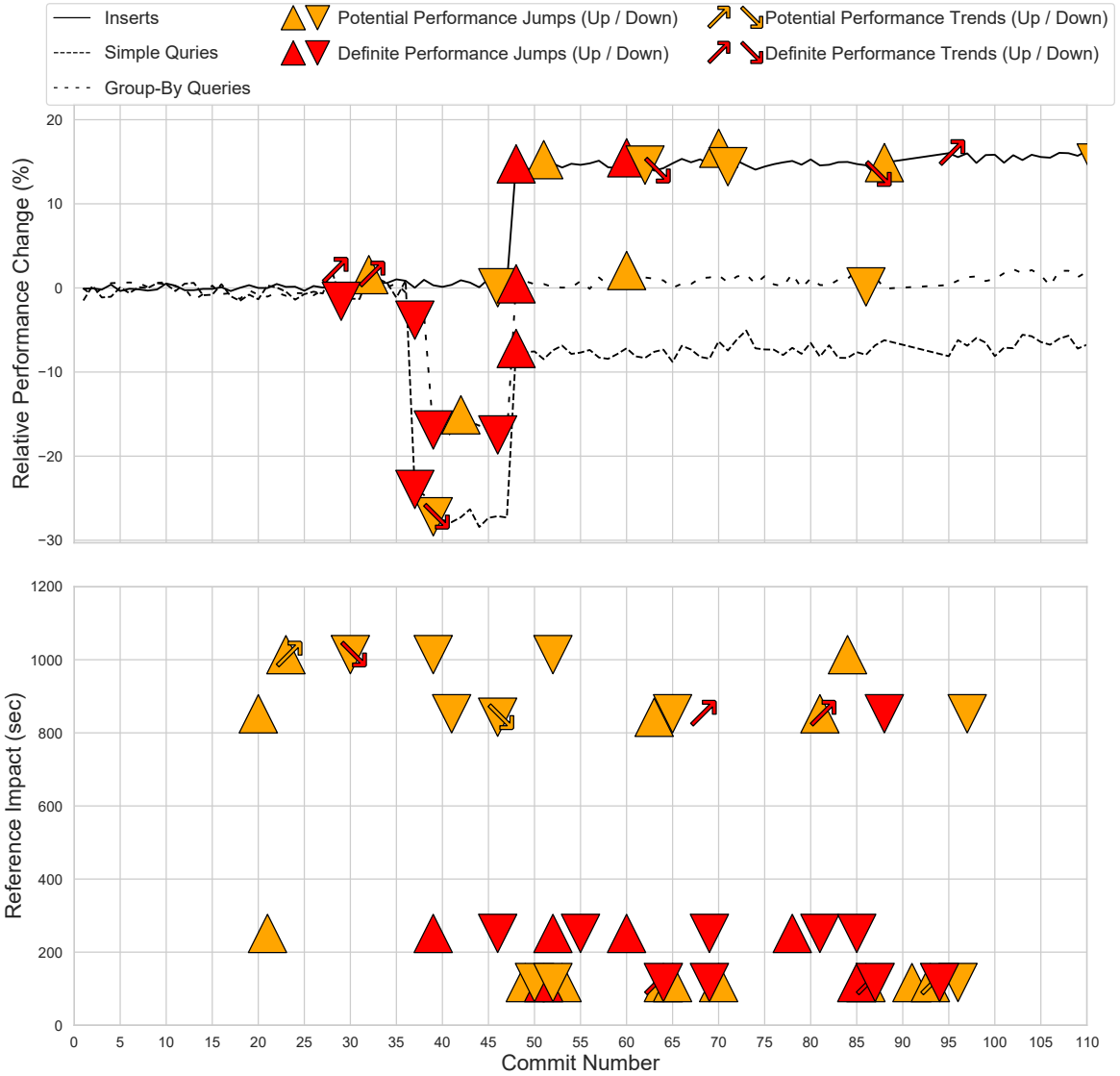


Figure 7.9: **Results from the optimized suite for *InfluxDB*.** The upper part shows the results of the application benchmark and its detections while the lower part shows the detections from the microbenchmarks (higher means more likely impact on application performance). The optimized suite of *InfluxDB* with $\approx$ 36% practically relevance detects five true positive alarms but also raises false alarms for 27 commits.

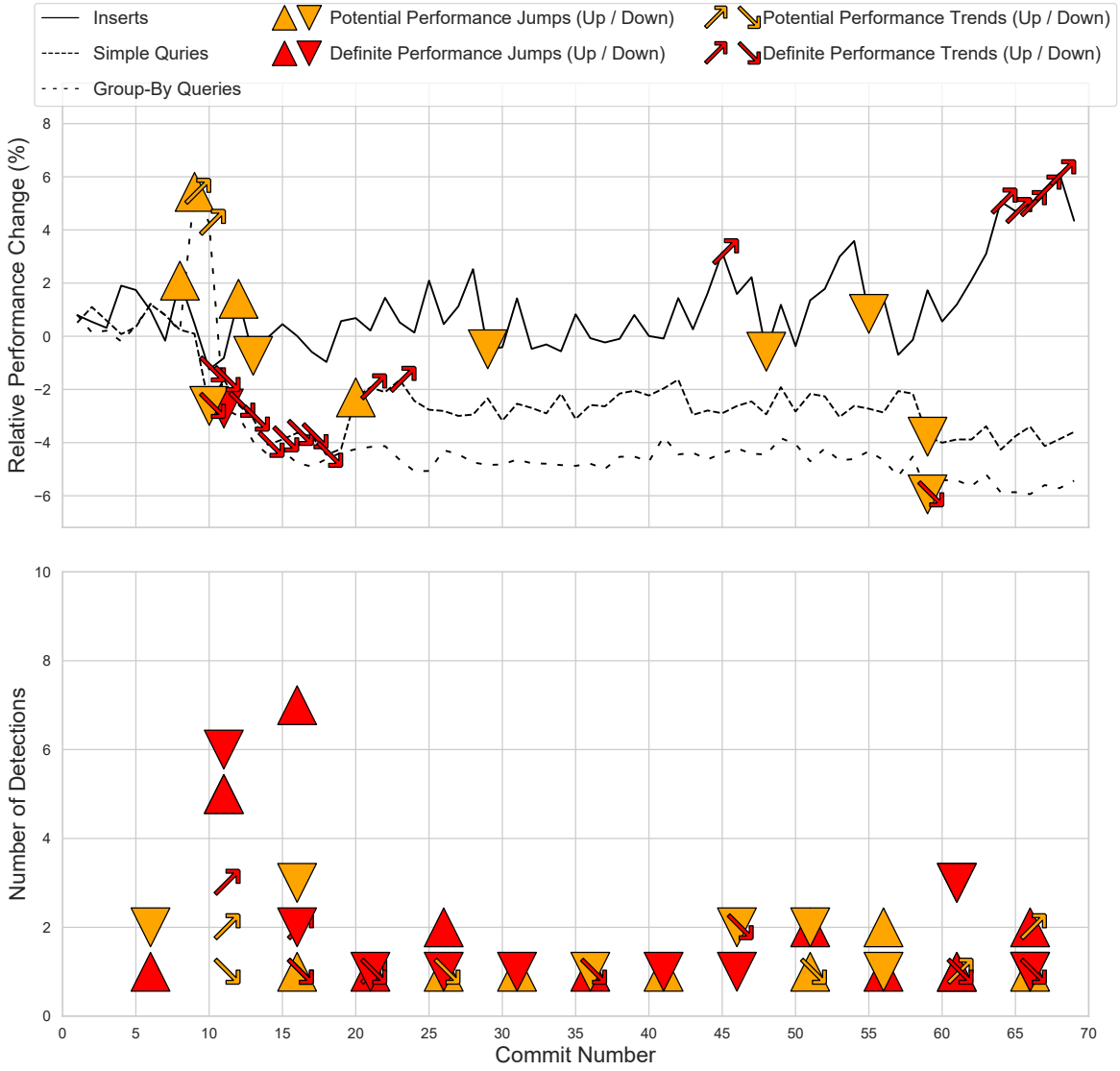


Figure 7.10: **Complete suite results for *VictoriaMetrics*.** The complete suite with 177 microbenchmarks in total detects 91 changes that can not be directly linked to the application-relevant performance metrics.

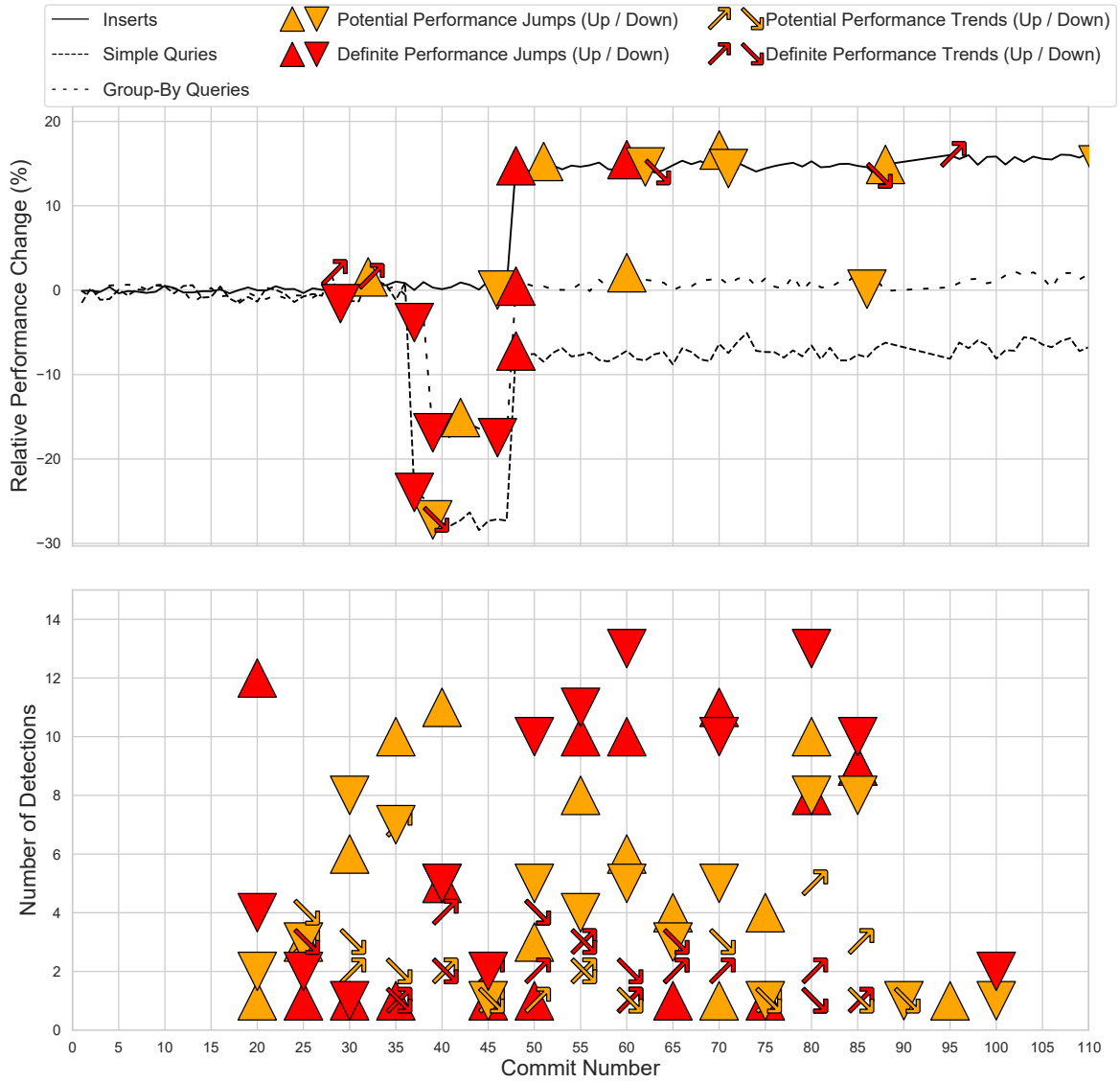


Figure 7.11: **Complete suite results for *InfluxDB*.** The suite shrinks down from 426 to 109 microbenchmarks during the evaluation period. Nevertheless, the complete suite detects 392 performance changes that can not be mapped to the application-relevant metrics.

Part IV

Benchmarking Platforms

Chapter 8

BeFaaSter

8.1 Introduction

All major cloud providers offer Function-as-a-Service (FaaS) solutions where users only have to take care of their source code (functions) while the underlying infrastructure and environment are abstracted away by the provider. FaaS applications are composed of individual functions deployed on a FaaS platform that handles, e.g., the execution and automatic scaling. Developers do not have direct control of the infrastructure and can only define high-level parameters, such as the region in which the function should run [26]. Due to this, FaaS platforms are easy to use but comparing cloud platform performance [106, 18] is challenging, as the cloud variability is further compounded by an additional, unknown infrastructure component.

Existing work on benchmarking of FaaS platforms usually focuses on the execution of small, isolated microbenchmarks that deploy and call a single function, e.g., a matrix multiplication [12] or a random number generator [114]. While *microbenchmarks* are useful for studying and comparing specific characteristics, they can give only limited insights into the platform behavior that will impact real applications [16]. An *application-centric benchmark*, in contrast, mimics the behavior of a realistic application in order to more realistically measure platform performance. This allows developers to better compare different service options, a strategy also taken by the TPC benchmarks.[1]

In previous work, we proposed *BeFaaS* [73], an extensible framework for executing application-centric benchmarks against FaaS platforms that included a realistic e-commerce benchmark as an example, to address this gap. At the time, BeFaaS was the only FaaS benchmarking framework with out-of-the-box support for federated cloud [98] and edge-to-cloud deployments [25, 14], which allowed us to evaluate complex application configurations distributed over platforms running on a mixture of cloud, edge, and fog nodes. Beyond this, BeFaaS fo-

[1] `https://www.tpc.org`

cuses on ease-of-use and collects fine-grained measurements which can be used for a detailed post-experiment drill-down analysis, e.g., to identify cold starts or trace request chains in detail.

In this extended version of our original paper [73], we enhance the BeFaaS framework with asynchronous cross-provider event pipelines, design and implement three further FaaS benchmark applications, add support for benchmarking edge-based FaaS platforms, and use BeFaaS to evaluate several FaaS providers in realistic and typical FaaS application setups. Specifically, we (i) compare FaaS offerings using a typical microservice-based application, (ii) evaluate hybrid edge-cloud FaaS setups, (iii) analyze the event pipeline interaction within and between providers, and (iv) study cold start behavior of FaaS platforms.

In total, we make the following contributions:

- We derive requirements for an application-centric FaaS benchmarking framework (Section 8.2).
- We propose BeFaaS, an extensible framework for the execution of application-centric FaaS benchmarks and describe four example benchmark applications (Section 8.3).
- We present our proof-of-concept prototype which is available as open source (Section 8.4).
- We run a number of experiments and use the results to compare FaaS offerings in several setups (Section 8.5).

Our extended study evaluates four different typical FaaS use cases on Amazon Web Services (AWS), Google Cloud Platform (GCP), Microsoft Azure (Azure), and tinyFaaS [130]. Our experiments result in the following findings:

1. For simple functions which work as glue code between frontend and storage layer, network transmission is a major contributor to overall response latency while the pure computing time of functions is almost negligible.
2. Even with a cloud database backend, an edge-only function deployment can outperform a mixed edge-cloud deployment in response latency as a result of transmission latency between functions.
3. While publishing events to event pipelines can be done within 100ms for all studied providers, the delay between a published event and the start of the triggered function ranges between about 100ms for AWS and 800ms for GCP.
4. Despite function execution duration being the highest on Azure Functions, our experiments find that the platform outperforms AWS Lambda and Google Cloud Functions in cold start behavior.

BeFaaS can help FaaS application developers compare cloud offerings and find the best provider for their specific use case by either deriving findings from our benchmark applications, adjusting the workload profiles to match their scenario, or using the BeFaaS library in their specific FaaS application for most accurate findings. Furthermore, FaaS platform developers could use BeFaaS as part of their CI/CD pipelines [160, 69] to detect performance regressions prior to live testing.

8.2 Requirements

While microbenchmarks are highly useful for studying individual features of a system-under-test (SUT), application-centric benchmarks support end-to-end comparison of different platforms and configurations. Aside from standard benchmarking requirements such as portability or fairness [83, 22, 24, 59, 16], an application-centric FaaS benchmarking framework needs to fulfill a number of specific requirements which we describe in this section.

R1 – Realistic Benchmark Application: The performance of a FaaS platform depends on the application that is deployed on it. For instance, an application that frequently causes cold starts through a growing request rate will be better off on AWS Lambda while an application that frequently causes cold starts through short temporary load spikes will be better off on Apache OpenWhisk due to their different request queuing mechanisms [19]. This means that the benchmark application should be as close as possible to the real application for which the analysis is made [16], in line with the findings of Shahrad et al. [150]. A key requirement is, hence, that *a FaaS benchmark should mimic real applications as closely as possible.*

R2 – Extensibility for New Workloads: FaaS platforms are highly flexible and can be used for a wide variety of applications, so the world of FaaS applications is evolving rapidly. As such, any set of "typical" FaaS applications – and thus the workload profile for a FaaS platform – can only be considered a snapshot in time. Likewise, the load profiles of existing FaaS applications, i.e., the amount and type of requests that the application handles, are likely to evolve over time. Therefore, we argue that *a FaaS benchmarking framework should be easily extensible in terms of adding new benchmark applications and updating load profiles for existing benchmarks.*

R3 – Support for Modern Deployments: FaaS is often used as the "glue" between cloud services, web APIs, and legacy systems [26]. Thus, a benchmarking framework must also consider these links and support external services. Furthermore, applications today are often distributed over cloud, edge, and fog resources, possibly even to the LEO edge [23, 165, 131]. Here, for example, edge servers can keep sensitive functions on premises while non-critical functions are hosted in a public cloud; similar setups exist for edge and fog com-

puting use cases [125, 68, 10, 25]. As such, assuming a single-cloud deployment is unrealistic for benchmarks aiming to be as similar as possible to realistic applications. *A benchmarking framework needs to support external services and federated setups in which application functions are deployed on one or more FaaS platforms distributed across cloud, edge, and fog.*

R4 – Extensibility for New Platforms: Today, all major cloud service providers offer FaaS platforms and there is a growing range of open-source FaaS systems, e.g., systems that specifically target the edge [65, 130]. As interfaces are constantly evolving and new platforms are being introduced, a cross-platform benchmarking framework *needs to be extensible to support future FaaS platforms.*

R5 – Support for Drill-down Analysis: An application-centric FaaS benchmark can help to evaluate the suitability of different sets and configurations of FaaS platforms for a specific application. What it can usually not provide are explanations for its finding, e.g., the different cold start management behavior of AWS Lambda and Apache OpenWhisk mentioned above [19]. To facilitate root cause analysis and help evaluators explain the patterns they see in the benchmark results, we argue that *an application-centric FaaS benchmarking framework should support drill-down analysis by logging fine-grained measurement results including typical metrics of microbenchmarks.*

R6 – Minimum Required Configuration Overhead: An application-centric FaaS benchmarking framework should be easy to use and provide reproducible results. This includes configuration, deployment, execution, as well as collection and analysis of results, e.g., using infrastructure automation. Hence, *a FaaS benchmarking framework should be designed to require as little manual effort as possible.*

8.3 Design

In this section, we give an overview of the BeFaaS design, starting with an overview of the BeFaaS architecture and components (Section 8.3.1) before describing the key features of BeFaaS (Sections 8.3.2 to 8.3.5).

8.3.1 Architecture and Components

In BeFaaS, executing functions of a benchmark application is the *workload* that actually benchmarks the FaaS platform, i.e., executing a function creates *stress* on the SUT. Since functions do not "self-start" executing, we need an additional load generator that invokes the FaaS functions of our benchmark application We show a high-level architecture overview in Figure 8.1.

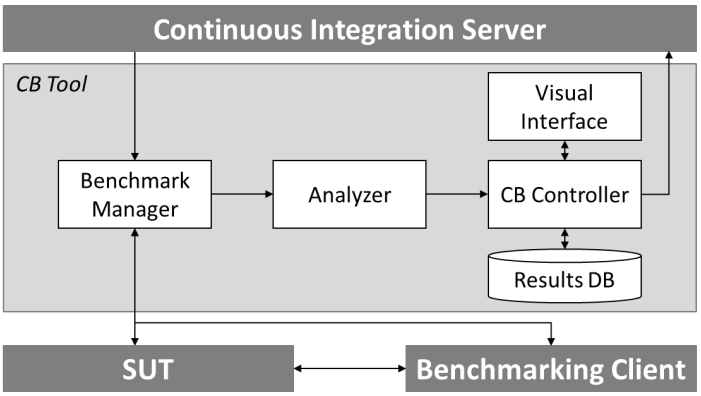


Figure 8.1: High-level overview of the BeFaaS architecture.

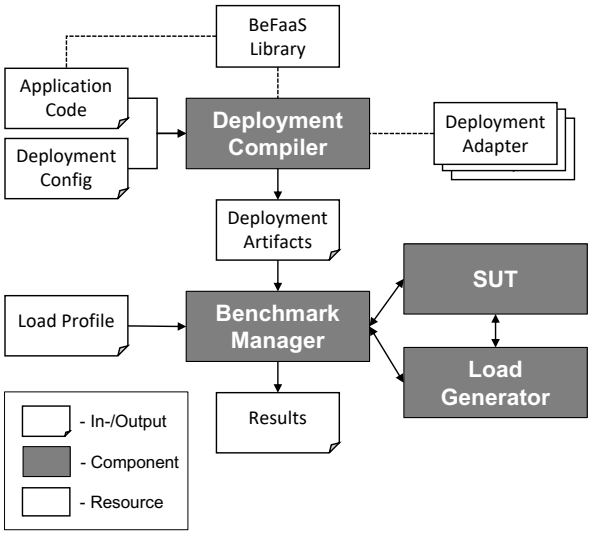


Figure 8.2: The Deployment Compiler transforms application code into individual deployment artifacts based on a deployment configuration. These are then deployed and invoked by the Load Generator to retrieve measurement results. Finally, the Benchmark Manager aggregates and reports fine-grained results.

For a benchmark run, BeFaaS requires three inputs: (i) the source code of the FaaS functions forming the benchmark application, (ii) a load profile for the load generator, and (iii) a deployment configuration that describes the environment configuration for each function and FaaS platform (the SUTs). We show the components of BeFaaS and their interaction in Figure 8.2.

Application code and deployment configuration are initially converted into deployment artifacts by the *Deployment Compiler*. The Deployment Compiler instruments and wraps each function's code with BeFaaS library calls and injects vendor-specific instructions defined in deployment adapters to enable request tracing and fine-grained metrics. The resulting deployment artifacts are passed to the *Benchmark Manager*.

The Benchmark Manager orchestrates the experiment: First, it configures the *SUT* by deploying each function based on the information in the respective artifact. If there are external services, e.g., a database service, these can either be deployed by the Benchmark Manager as well or linked to the SUT using environment variables. In the second step, the Benchmark Manager initializes the *Load Generator* with the workload information described in a load profile. Then, the benchmark run is triggered and the Load Generator invokes the functions of the benchmark application, which log every request in detail, including timestamps, origin function, and called functions (if applicable). Once the benchmark run is completed the Benchmark Manager collects function logs, aggregates them, and destroys all provisioned resources.

8.3.2 Realistic Benchmarks

To provide a relevant and realistic application-centric benchmark (**R1**), BeFaaS already comes with four built-in benchmarks which mimic and represent typical use cases for FaaS applications. These include a microservice-based web application to study request-response patterns, an IoT application scenario to evaluate hybrid edge-cloud setups, a smart factory application to measure event trigger performance, and a microservice application to study cold start behavior and elasticity capabilities (details are further explained in Section 8.4). Our application benchmarks are in line with the empirical findings of Shahrad et al. [150] regarding typical FaaS applications: All are composed of several functions that interact with each other to form function chains, use synchronous HTTP or asynchronous event triggers, and use external services such as a database system for persistence. The benchmark applications come with a default load profile that covers all relevant aspects as well as several further load profiles to emphasize selected stress situations, e.g., to provoke more cold starts. In combination, the benchmarks each represent complete FaaS applications: load balancing at the provider endpoint(s), interconnected calls of several functions, calls to external services such as database systems, and multiple load profiles which, e.g., provoke cold starts of functions.

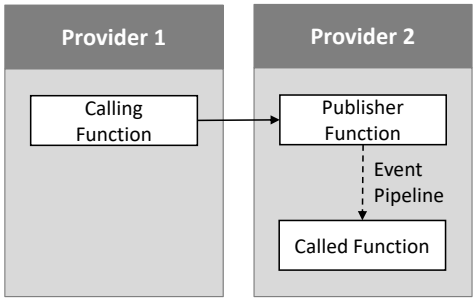


Figure 8.3: A publisher function forwards incoming events to the respective event pipeline to trigger the called function.

The modular design of BeFaaS, however, also allows us to easily add further benchmark applications and load profiles or to adapt existing ones to the concrete needs of the developer (**R2**). For adding a new benchmark, the respective application only needs to use the BeFaaS library (described in Section 8.4) for function calls and to have unique function names.

8.3.3 Benchmark Portability and Federated FaaS Deployments

To support portability of benchmarks and federated deployments, BeFaaS relies on unique function names, individual deployment artifacts for every function, and a single endpoint for every deployed function (**R3**): With globally unique function names, the endpoints of the deployed functions are already known during the compilation phase. The Deployment Compiler maps these endpoints to the canonical function names (defined in the application) and compiles them into the source code. Moreover, the compiler also injects endpoints to external services such as database systems using environment variables which were set in the respective setup script or defined manually. To enable asynchronous function calls, the Deployment Compiler creates and assigns a topic-based event pipeline on the respective provider for each asynchronous function. To trigger this pipeline, requests are sent as events to a publisher function, which is deployed for every provider and forwards the request to the respective event topic (see Figure 8.3). In total, this decouples the ability of a function to call another function or a platform service from its deployment location and enables BeFaaS to support arbitrarily complex deployments: it is indeed possible to run every function on a different FaaS platform – as configured by the benchmarker.

Each FaaS platform offers a different interface for life-cycle and configuration management of functions. As the smallest common interface, BeFaaS requires that each platform provides API-based access to (i) deploying functions, (ii) retrieving log entries from the standard logging interface, and (iii) removing functions. The Deployment Compiler wraps this functionality using an adapter mechanism and selects the appropriate instructions for the target platform specified in the deployment configuration. Additional FaaS platforms that fulfill this minimal interface can easily be added by implementing a corresponding adapter (**R4**).

8.3.4 Detailed Request Tracing

To enable a detailed drill-down analysis of experiment results (**R5**), the Deployment Compiler injects and wraps code that collects detailed measurements during the benchmark run: The compiler adds timestamping to determine start, end, and latency of calls to functions and external services.

Besides these timestamps, the compiler also injects code that generates context IDs and pair IDs to assign individual calls to their respective context later on. Here, a context ID is generated once for each function chain (with the first function call) and propagated to every subsequent call to other functions. To link the individual calls of a function chain, the compiler injects source code to create pair IDs of randomly generated keys that link caller and callee. Thus, it is possible to trace every single request through the benchmark application and to generate call trees for every context and function chain during post-experiment analysis.

Finally, to independently and reliably detect cold starts, the Deployment Compiler also injects code that evaluates a local environment variable on the executor at the provider side. If this variable is not present, the function runs on a new executor (cold start), the variable is created, filled with a randomly generated key, and the cold start is logged.

All data that enable fine-grained results (timestamps, context IDs, pair IDs, and executor keys) are recorded on the console using the standard logging interface of the respective FaaS vendor. In initial experiments with AWS, GCP, and Azure, we verified that the cost of logging is at most in the microsecond range.

8.3.5 Automated Experiment Orchestration

The BeFaaS framework requires only the application code, a deployment configuration, and a load profile to automatically perform the benchmark experiment (**R6**). First, all business logic, dependencies, and BeFaaS instrumentation logic are bundled into a single deployment artifact by the Deployment Compiler. Next, the Benchmark Manager orchestrates the experiment and provides a simple interface for starting the benchmark run, monitoring its process, and collecting fine-grained results for further analysis.

8.4 Implementation

Our open-source prototype implementation of BeFaaS[2] includes (i) the BeFaaS library, (ii) four deployment adapters, (iii) the Deployment Compiler, (iv) the Benchmark Manager, (v) four realistic benchmark applications, and (vi) several load profiles for the benchmark applications.

[2] `https://github.com/Be-FaaS`

The BeFaaS library is written in JavaScript and handles calls to other functions depending on their canonical name, generates tracing IDs, and takes timestamps. BeFaaS deployment adapters are implemented using Terraform[3] commands. Currently, BeFaaS thus supports three major cloud offerings (AWS Lambda, Google Cloud Functions, and Azure Functions) as well as the open-source system tinyFaaS [130], which supports the deployment of functions on private infrastructure, including edge or fog nodes. The Deployment Compiler is a shell script that uses several tools to build the deployment adapters for the respective platforms, parses and injects information from the Deployment Configuration, and generates the deployment artifacts from the application code. The Benchmark Manager uses Terraform to create the infrastructure based on these artifacts, collect the logs, and later remove provisioned resources. The implemented benchmark applications are written in JavaScript and include calls to external services such as a Redis[4] instance. The Load Generator uses Artillery[5] to call the benchmark application. It either executes a realistic default load profile that stresses all relevant aspects of the application or specific additional load profiles that emphasize stress situations, e.g., to provoke more cold starts. New load profiles can easily be added by specifying new Artillery load descriptions (YAML[6] configuration files).

8.4.1 Benchmark 1: Web Shop (Microservices)

Our e-commerce benchmark implements a web shop as a FaaS application derived from Google's microservice demo application.[7] Our corresponding benchmark implementation follows a typical request-response invocation style, comprises 17 functions, and uses a Redis instance as an external service to persist state (see Figure 8.4). Besides functions that provide recommendations and advertising, customers can log-in, set their preferred currency, view products, fill a virtual shopping cart, check out orders, and finally observe order shipping. Each task is implemented in a separate function and all requests arrive at a single function, the frontend, which takes the customer calls and routes them to the respective backend functions. There are blocking synchronous calls to other functions as well as asynchronous call blocks that idle until all called functions return.

The default load profile simulates four different customer workflows and constant traffic for 15 minutes. Our e-commerce benchmark is particularly well suited for comparing request-response behavior and study request details of different cloud providers but can also be used to explore federated cloud deployments, e.g., for scenarios in which the application is running on multiple cloud platforms.

[3] `https://www.terraform.io/`
[4] `https://redis.io/`
[5] `https://artillery.io/`
[6] `https://yaml.org/`
[7] `https://github.com/GoogleCloudPlatform/microservices-demo`

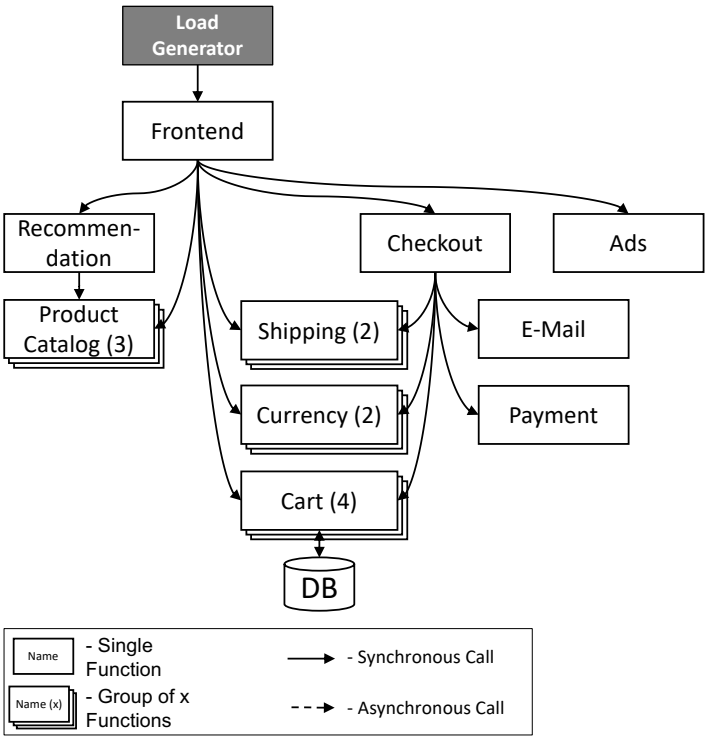


Figure 8.4: The e-commerce application implements a web shop in 17 functions. The Frontend serves as a single entry point and an external database is used to store state. We group some functions to increase legibility.

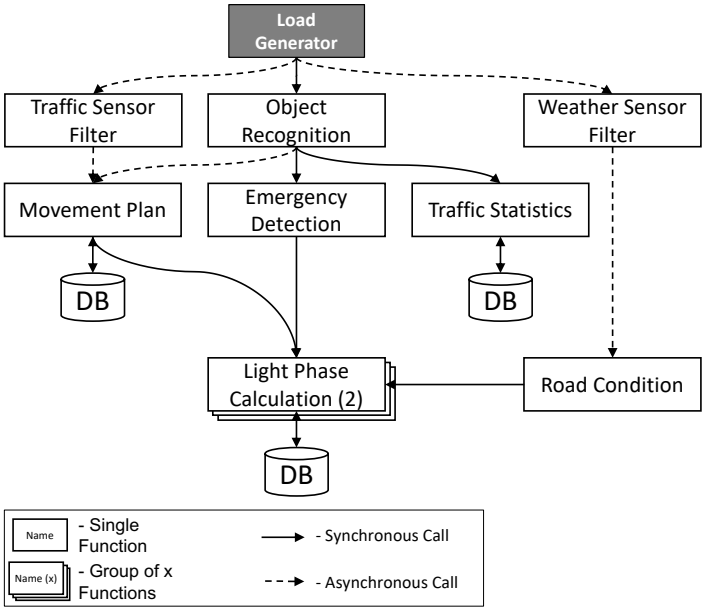


Figure 8.5: The IoT application implements a smart traffic light scenario in nine functions. The Load Generator emulates sensor data and sends them to three different entry points.

8.4.2 Benchmark 2: Smart City (Hybrid Edge-Cloud)

Although several IoT applications and use cases already exist in research (e.g., [9, 30, 135, 118, 68]), none of them could directly be used or adapted as a FaaS application. Thus, we designed our smart city benchmark application around typical IoT patterns and implemented a use case based on a smart traffic control scenario inspired by the *InTraSafEd5G* system [110, 112].

The benchmark application uses a mix of synchronous and asynchronous function calls and implements an IoT use case with a smart traffic light which adapts its light phase based on traffic sensors, a camera, and weather inputs (see Figure 8.5). The functions initially filter incoming data streams and perform object recognition on camera footage to create a movement plan, detect ambulance/emergency cars, and maintain a traffic statistic. The regular light phase is then determined based on this movement plan, road conditions, and the current light phase. Emergency services can override the regular phase at any time by raising an emergency event that stops all other traffic.

The load profile for this application emulates sensor data and injects emergency events. The traffic sensor sends an update every two seconds to the Traffic Sensor Filter, the Object Recognition processes one image every two seconds, and the weather is updated every twenty seconds. Furthermore, the Load Generator also injects an emergency event every two minutes which lasts five seconds each. This default load profile runs for 15 minutes. As this use case

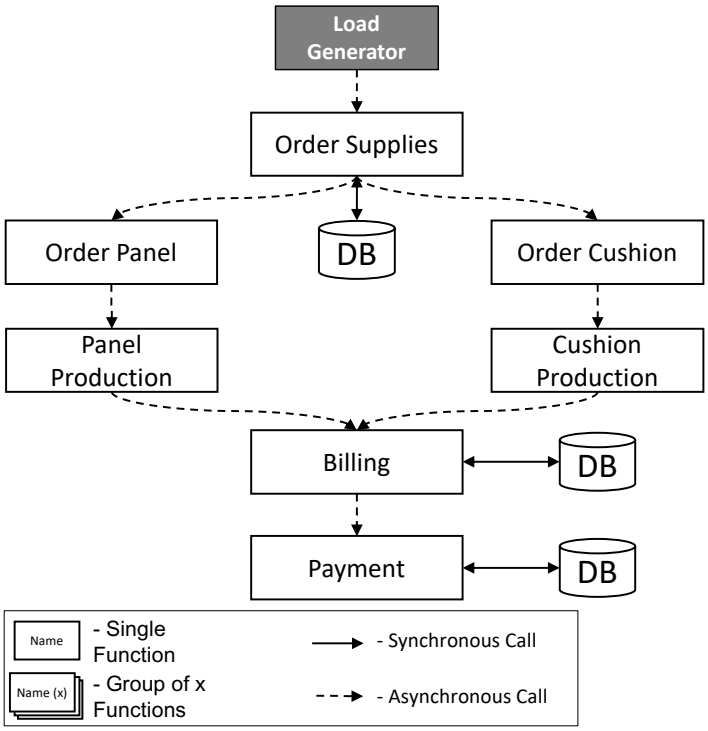


Figure 8.6: The Industry 4.0 application implements a smart factory in seven functions. An Order Supplies function serves as single entry point for different asynchronous event pipelines which can be distributed among several providers.

will in practice typically have a very predictable and stable load profile, we did not implement alternative load profiles – benchmark users can, however, easily add them if needed.

The smart city benchmark is particularly well suited for comparing different deployments across cloud, edge, and fog.

8.4.3 Benchmark 3: Smart Factory (Event Trigger)

Our smart factory benchmark application implements asynchronous event-based pipelines. In our example use case, users order personalized couches consisting of panels and cushions (see Figure 8.6). First, an order function determines the number and individual sizes of panels and cushions which are then each ordered by sending an event to the respective order function. Both order functions, in turn, transform their order into a production event which is sent to the production function which mimics the production of the panel or cushion. After production, the functions emit an accounting event which is consumed by the billing function. Once all panels and cushions are produced, and all accounting events are processed, the payment function finally issues an invoice.

The default load profile orders a new couch every five seconds for 15 minutes. Each order, in turn, implies 8 to 18 order events, depending on the ordered couch model. This smart

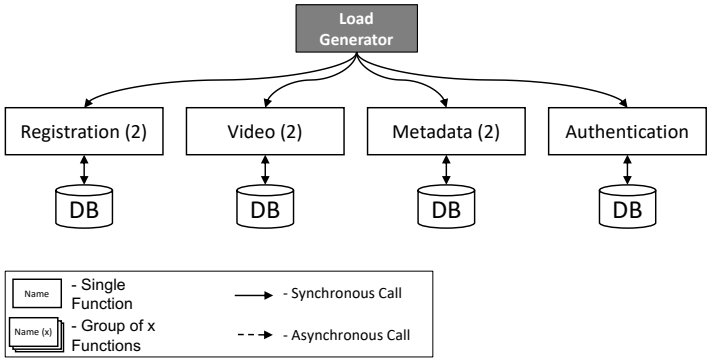


Figure 8.7: The streaming service comprises seven functions and a workload that triggers cold starts. After an Internet outage, there is a high load on functions dealing with authentication and metadata.

factory benchmark application is particularly well suited to study event pipelines, but can also be used to analyze the interplay between different FaaS providers. For example, when collaborating with suppliers (panels and cushions in our case), they may also be deployed on another provider, thus, requiring cross-cloud interoperability.

8.4.4 Benchmark 4: Streaming Service (Cold Start Behavior)

An often stated advantage of FaaS applications is their elastic scalability. Thus, we include a streaming service benchmark application which triggers cold starts and can require automatic scaling capabilities. The Load Generator here mimics video streaming devices which register users, request video files, update meta information such as viewing progress, and handle backend authentication (see Figure 8.7). In case of a larger Internet outage, these devices are offline, but it is possible to continue watching already downloaded movies while the corresponding metadata is updated. As soon as network connectivity is restored and the streaming devices are back online, all clients reconnect and concurrently invoke functions, which triggers cold starts.

The default load profile for this benchmark is split into four phases. First, an initial set of users and streaming devices is registered. Once all initial data has been read, the normal load phase starts for 5 minutes in which 500 request flows add new videos, request videos, and update metadata. Third, the failure is simulated by pausing requests for 20 minutes. Finally, the benchmark triggers cold starts by suddenly sending 1,500 request flows distributed over another 5 minutes. Our streaming application benchmark is particularly well suited for comparing the cold start behavior and automatic scaling capabilities of different FaaS providers.

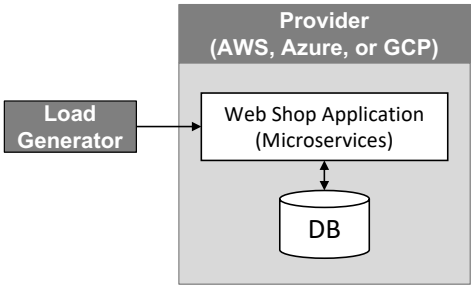


Figure 8.8: As part of the FaaS application, the database instance is deployed in the same region and on the same provider as the rest of the web shop.

8.5 Evaluation

Our evaluation is split into two parts: First, we present the results of four experiments in which we use BeFaaS to stress different FaaS platforms (Sections 8.5.1 to 8.5.4). Second, in Section 8.5.5, we discuss to which degree BeFaaS fulfills our requirements from Section 8.2.

In all experiments, we deploy the Load Generator on a (vastly over-provisioned) virtual machine (2 vCPUs and 4 GB RAM) and let it execute the default load profile of the respective benchmark application against the SUT deployed in either *eu-west-1* for AWS, *westeurope* for Azure, or *europe-west1* for GCP. As runtime for the benchmark applications, we run node.js 18 on all SUT options and use 256MB of memory per function (SKU Y1 on Azure). Moreover, the Redis database system used by the SUTs also runs on an over-provisioned virtual machine (2 vCPUs and 4 GB RAM; *ta3.medium* at AWS, *Standard_B2S* in Azure, and *e2-medium* at GCP) at the respective provider site. This ensures that the database instance and Load Generator will not be a bottleneck during the experiments [16]. All experiment results reported here are from the period June to July 2023. We explicitly decided not to compare to the results from our original paper [73] as we made a number of smaller changes across the BeFaaS codebase and also used the opportunity to update all libraries and platform SDKs used. As a result, we reran all experiments from scratch since we could not rule out effects from our benchmarking tool.

8.5.1 Experiment 1: Using the web shop application benchmark to compare major cloud FaaS providers

In our first experiment, we deploy BeFaaS in single cloud provider setups in which all functions of the web shop application are deployed on a single provider (namely AWS, Azure, and GCP) and use the default load profile to compare them (see Figure 8.8). During each experiment, the Load Generator executes 18,000 workflows, which each consist of 1 to 9 requests, over a time span of 15 minutes.

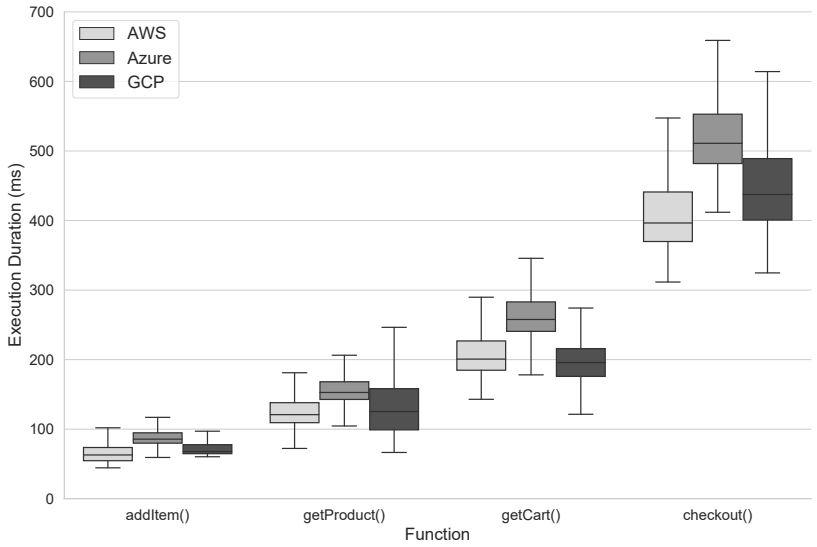


Figure 8.9: A detailed analysis of four functions called from the frontend shows that AWS provides the best performance and that the execution duration has the highest variance on GCP.

Figure 8.9 shows the execution duration of four selected functions with varying degree of complexity which are called from the frontend function (visualized as box plots; boxes represent quartiles, whiskers show the minimum and maximum values without outliers beyond 1.5 times the interquartile range). For the four functions examined in more detail, the overall picture is similar for all three providers: As expected, simpler functions that only read or write a single value have a lower execution duration than more complex ones such as the *getCart()* or *checkout()* function. In our experiment, Azure provided the slowest environment for this single run while GCP showed a higher variance for the *getProduct()* and *checkout()* function.

In a further fine-grained analysis, we investigate the distribution of computing, network transmission, and database query latency for a function sequence putting an item into the shopping cart. This includes synchronous and blocking calls to two functions and several database operations.

For this evaluation, we consider the (i) computation part as function execution duration without the duration of outgoing network calls, (ii) network latency as the duration of outgoing calls to other function without the execution duration of the called function itself, and (iii) query latency as the duration of calls to the external database. The detailed timestamp mechanisms of BeFaaS allow us to easily separate these times.

The results of this analysis are shown in Figure 8.10. In this specific but typical interaction in which FaaS functions are the glue code to interact with external services, it is noticeable that for all providers time is mostly spent on network transmission followed by the database

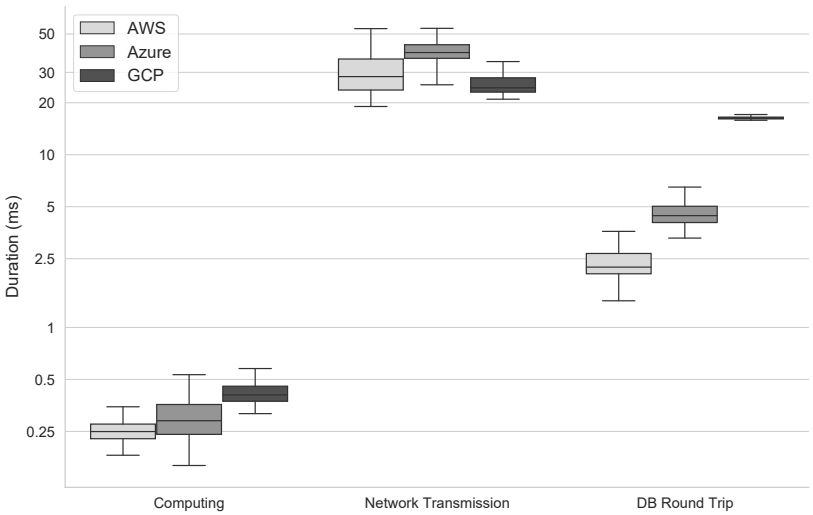


Figure 8.10: A drill-down analysis of a function sequence reveals that the network transmission time is the most relevant driver of execution time on all providers.

round-trip time while the actual computing time is below 1ms for all providers. Furthermore, database access takes longer for GCP than for the other providers.

8.5.2 Experiment 2: Evaluating hybrid edge-cloud setups using the smart city application

In this experiment, we compare an edge-focused and a hybrid edge-cloud setup. For the mixed setup, we split the smart city application into a cloud part, which is deployed on either AWS, Azure, or GCP, and an edge part, which is deployed on a local Raspberry Pi in Berlin, Germany running the tinyFaaS platform (see Figure 8.11). For the edge-focused setup, we deploy all functions of the smart factory application on tinyFaaS and only use a cloud-located database at the respective provider. During the experiment, the Load Generator simulates the smart city scenario for 15 minutes by triggering the traffic sensor and object recognition every two seconds and the weather sensor every 10 seconds for both evaluated setups.

Similar to our first experiment, we analyze the computing, network transmission, and database round trip times for both setups and all providers (see Figure 8.12). For all providers, the default workload finishes sightly faster in the edge-focused setup. Here, the network transmission times are reduced as all functions run on the same edge device, the compute duration is shorter compared to the mixed cloud-edge setup, but the database round trip time increases as every database access triggers a cloud request. This last addition, however, does not outweigh the other two improvements for the given default load.

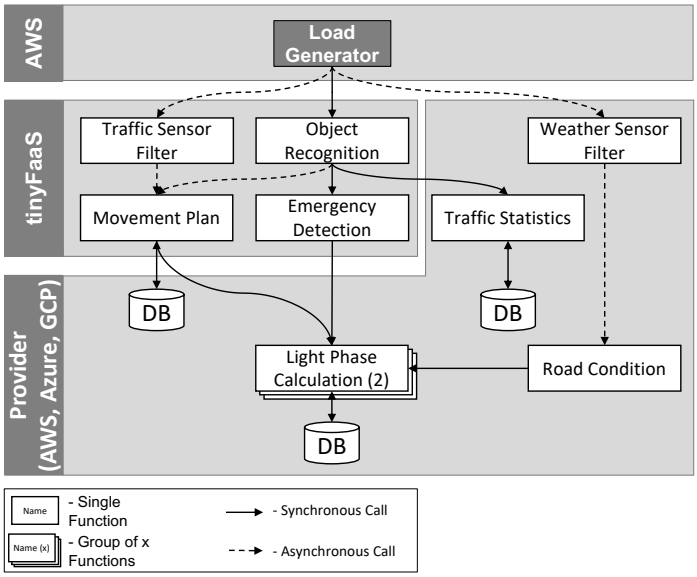


Figure 8.11: Functions related to the traffic light are deployed on a Raspberry Pi at the edge location while others run on public cloud providers.

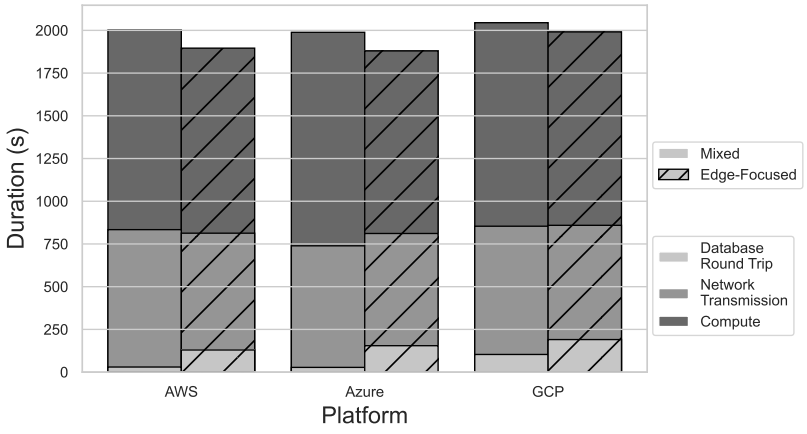


Figure 8.12: The cloud database increases the database duration for the edge setup for all providers. In total, however, both compute and network duration are reduced, and the total execution duration is lower for all edge setups.

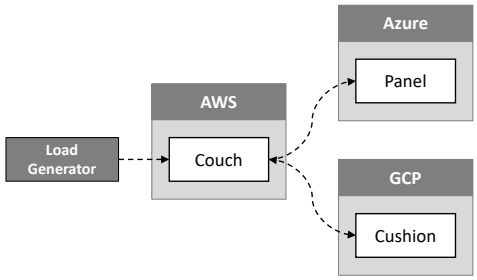


Figure 8.13: Each provider hosts a part of the smart factory application.

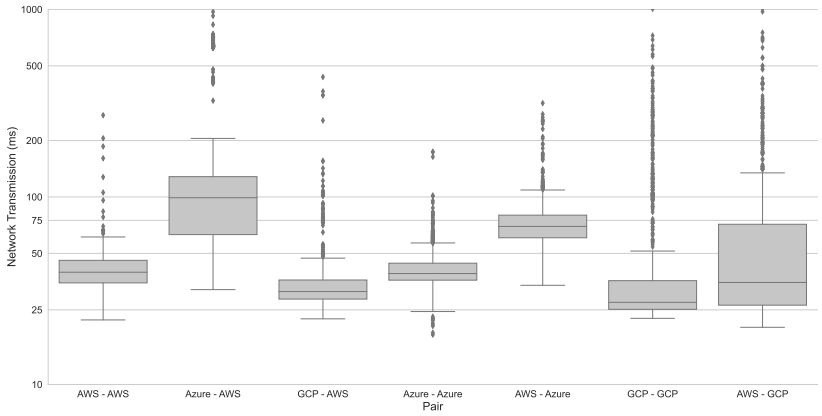


Figure 8.14: The network latency to publish an event usually ranges between 25ms and 100ms.

8.5.3 Experiment 3: Analyzing the event pipeline interplay within and across FaaS providers.

This experiment intends to investigate both how event pipelines perform within a provider, but also how well the interplay of cross-provider pipelines works. Thus, we split the event-based smart factory application into three parts: 1. Couch (running supplies, billing, and payment), 2. Panel (running panel order and production), and 3. Cushion (running cushion order and production). Each part is deployed on AWS, Azure, or GCP (see Figure 8.13). The load for this experiment consists of 180 couch orders over a time span of 15min, which triggers thousands of function invocations.

First, we analyze the time it takes to publish an event at the respective provider endpoint (see Figure 8.14). Again, we measure the outgoing call from the calling function and subtract the execution duration from the execution time of the called publisher function running on the destination provider. Depending on the origin and destination provider this usually takes between 25ms and 100ms, except for the Azure-AWS pair, which may take up to 200ms. Moreover, there are outlier values of more than one second in five pairs.

Second, we also investigate the total execution time of the publisher function running on the destination provider and the trigger delay between the start of the publisher function and the

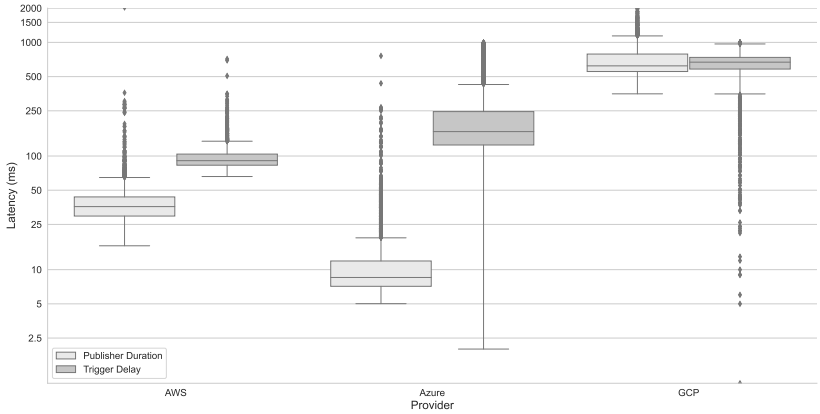


Figure 8.15: The execution time of the publisher function ranges from about 10ms on Azure to about 800ms on GCP. The trigger delay between publisher start and function start varies from 100ms on AWS to about 800ms on GCP.

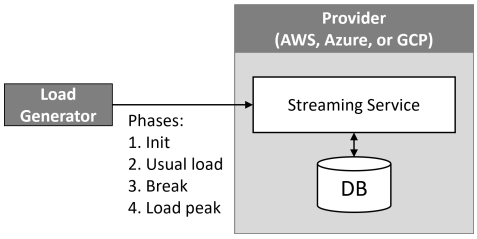


Figure 8.16: Similar to experiment 1, we deploy BeFaaS in single cloud provider setups in which all functions of the steaming service application are deployed on a single provider (namely AWS, Azure, and GCP).

start of the triggered function (see Figure 8.15). Here, Azure and AWS show fast publishing functions which usually finish within 100ms while for GCP it usually takes between 500ms and 1000ms to run the publisher function. For the trigger delay, AWS triggers the respective function fastest with about 100ms, followed by Azure with 75% of values below 250ms, and GCP with 75% of values above 500ms. Furthermore, it is noticeable that outliers in the other direction are possible, i.e., an event sometimes triggers the function execution immediately within 10ms for Azure and GCP.

8.5.4 Experiment 4: Studying the cold start behavior of different providers.

In our last experiment, we study the cold start behavior of all cloud providers and deploy the streaming service application once on each provider (see Figure 8.16). Running the default workload for this use case, we execute 500 requests within the first 5min load phase, which is then followed by a 20min break, and finally execute 1,500 requests for another 5 minutes.

Figure 8.17 shows the function execution time of all calls of the streaming application in the first 30 seconds of the *Load Peak* phase. In this phase, AWS and GCP both log cold starts and show larger latency values in the first seconds of the experiment. While for AWS the

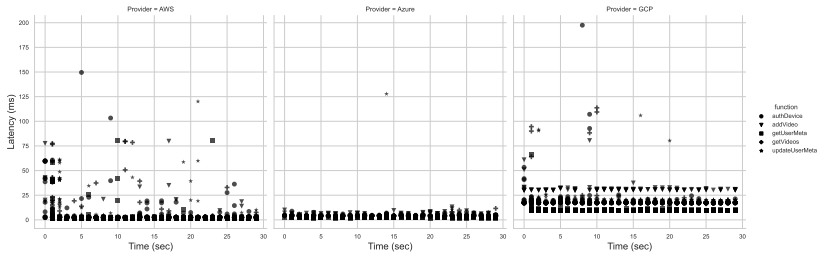


Figure 8.17: In the first 30 seconds of the load peak phase, AWS and GCP show larger latencies due to cold starts. Azure is presumably not affected by cold starts for this workload.

values do not stabilize for the first 30 seconds, the GCP execution times are stable after 10 seconds. Azure does not report any cold starts and shows stable execution durations. However, we possibly also missed the respective log entry due to the log framework limitations: The Azure logging framework only writes 250 log lines per second at maximum, further lines are discarded. Thus, only 9,143 out of about 15,000 values across the whole experiment time are available for this analysis.

8.5.5 Discussion of Requirements

In Section 8.2, we had identified six requirements for application-centric FaaS benchmarking frameworks. We now discuss to which degree BeFaaS fulfills these requirements.

BeFaaS already comes with four standard benchmark applications that cover many representative FaaS application scenarios, namely standard web applications, a hybrid edge-cloud scenario, an event-based smart factory application, and a microservice-based streaming service. Moreover, BeFaaS can be easily extended by implementing more FaaS application scenarios using the BeFaaS library and further workload profiles. We, hence, believe that BeFaaS fulfills the requirements **R1** (*Realistic Benchmark Application*) and **R2** (*Extensibility for New Workloads*).

In BeFaaS, benchmark users can define arbitrarily complex deployment mappings of functions to target FaaS platforms including federated multi-cloud setups or mixed cloud/edge/fog deployments. In fact, each function could run on a different platform. To achieve this, BeFaaS transforms the benchmark application into deployment artifacts fitted to the target platform. Adding another target platform is also straightforward and only requires the benchmark user to implement an adapter component for the respective FaaS platform or to copy and adapt an existing adapter component. Based on this, we argue that BeFaaS fulfills the requirements **R3** (*Support for Modern Deployments*) and **R4** (*Extensibility for New Platforms*).

At runtime, BeFaaS collects fine-grained measurements and traces individual requests similar to what Dapper [151] does for microservice applications. This offers the necessary information basis for drill-down analysis. Beyond this, BeFaaS also offers visualization capabilities for

select standard measurements to further support analysis needs. Overall, we hence conclude that BeFaaS addresses requirement **R5** (*Support for Drill-down Analysis*).

Finally, we believe that BeFaaS is easy to use due to its experiment automation features and requires only very few configuration files (requirement **R6** – *Minimum Required Configuration Overhead*). Nevertheless, this is a highly subjective matter that depends on the respective individual.

8.6 Discussion

BeFaaS is a powerful application-centric FaaS benchmarking framework. There are, however, also some points to consider and limits when using BeFaaS.

Tracing Overhead BeFaaS supports a detailed tracing of requests by injecting a small token in each call. On the one hand, this supports the clear mapping of different calls to function chains, yet on the other hand, it also causes an additional network overhead. This token, however, is of constant size (depending on the length of the respective function name), so the overhead can be easily determined and considered in results analysis. Furthermore, this will only matter if the goal of the benchmark is to find the optimal deployment for an existing application which is then instrumented to be used as a BeFaaS benchmark. For any of our standard benchmarks, it simply increases the benchmark workload stress slightly.

Measuring External Services Currently, BeFaaS handles external services and components as a black-box and only measures end-to-end latency of such service calls. In future work, however, we plan to implement a small BeFaaS sidecar proxy that can be deployed on external service instances to forward calls to the respective service and to inject the BeFaaS tracing token there as well.

Fairness with External Dependencies The included benchmark uses an external database system to persist state but further benchmarks and use cases may also require external services such as pub/sub message brokers or web APIs. Although the modular design of BeFaaS supports this, there are also some pitfalls in terms of fairness and comparability: In our experiments, we deployed the database instance with the same provider and in the same region as our functions to minimize latency between functions and database. In this setup, a function calling the external service and awaiting the response will not idle for a long time and the execution environment at the provider side will soon be available again for the next request. On the other hand, a function calling an external service in another region with larger latency will block the environment and (may) cause a cold start for the next incoming request. Thus, when using external services, these should be located and deployed with similar latency for all alternatives. Moreover, as cloud environments at least appear to be infinitely scalable, it

has to be assured that the external service does not become a performance bottleneck during the experiment. Otherwise, the benchmark would benchmark the compute resources of the external service instead of the performance of the FaaS platforms.

Provider-specific Features Competing FaaS vendors are constantly developing new and exclusive features that simplify development and deployment for customers. These features, however, can also affect the portability of the BeFaaS framework if a (future) benchmark uses exclusive features that are not available at all vendors. Thus, we strongly recommend not to use exclusive features of individual providers when developing new BeFaaS benchmarks. BeFaaS can, however, help to determine the impact of new features within a provider or across multiple providers by adjusting and configuring the respective deployment adapter.

Time Synchronization The drill-down analysis features of BeFaaS require approximately synchronized clocks. Although this will usually be provided by the provider with sufficient accuracy, a user should assert this before running experiments as this will affect the reliability of tracing insights. Nevertheless, such detailed insights may often not be needed and the tracing of BeFaaS also offers a mechanism to partially mitigate this: If the call follows a request-response pattern, BeFaaS measures the total round trip time at the calling function and knows the computing duration at the called function. Thus, it is possible to approximate the network transmission latency under the assumptions that both directions took comparably long. This is even possible for event-based calls that do not return a message to the sender, as calling functions submit the trigger events to the respective publisher function which runs on the same provider as the called function. In our experience, though, this is neither a problem in the cloud nor for self-hosted FaaS platforms, where the user has direct control over clock synchronization.

8.7 Conclusion

FaaS platforms are a popular cloud compute paradigm and have also been proposed for edge environments. For comparing and choosing different FaaS platforms in terms of performance, developers usually rely on benchmarking. Existing FaaS benchmarks, however, tend to fall into the microbenchmark category – an application-centric FaaS benchmarking framework is still missing.

In this paper, we presented BeFaaS, an extensible framework for executing application-centric benchmarks against FaaS platforms which comes with four realistic FaaS benchmark applications. BeFaaS is the first benchmarking framework with out-of-the-box support for federated cloud setups which allows us to also evaluate complex configurations in which an application is distributed over multiple FaaS platforms running on a mixture of cloud, edge, and fog nodes.

Beyond this, BeFaaS is focused on ease-of-use through automation and collects fine-grained measurements which can be used for a detailed post-experiment drill-down analysis, e.g., to identify cold starts or other request-level effects; it can easily be extended with additional benchmarks or adapters for further FaaS platforms.

With BeFaaS, we provide developers with the necessary tool to explore, compare, and analyze FaaS platforms for their suitability for application scenarios. We also offer researchers the ability to study the performance effects of different FaaS deployment options across cloud, edge, and fog through experiments. Finally, FaaS platform developers can use BeFaaS in their CI/CD pipeline to compare their own platform to previous versions of it as well as to their competitors.

Part V

Conclusions

Although the approaches presented for optimizing the benchmarks are primarily applicable to microservice applications and platforms in the cloud, they can also be used in other domains.

List of Figures

List of Tables

Bibliography

[1] Ali Abedi and Tim Brecht. "Conducting Repeatable Experiments in Highly Variable Cloud Computing Environments". In: *Proc. of the International Conference on Performance Engineering (ICPE '17)*. ACM, 2017, pp. 287–292.

[2] Ali Abedi, Andrew Heard, and Tim Brecht. "Conducting Repeatable Experiments and Fair Comparisons using 802.11n MIMO Networks". In: *ACM SIGOPS Operating Systems Review*. ACM, 2015, pp. 41–50.

[3] Carlos M Aderaldo et al. "Benchmark requirements for microservices architecture research". In: *Proceedings of the 1st International Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering*. IEEE Press. 2017.

[4] Hammam M AlGhamdi et al. "Towards reducing the time needed for load testing". In: *Journal of Software: Evolution and Process*. Wiley, 2020.

[5] Hammam M. AlGhamdi et al. "Towards Reducing the Time Needed for Load Testing". In: *Journal of Software: Evolution and Process*. Wiley, July 2020. DOI: 10.1002/smr.2276. URL: https://doi.org/10.1002/smr.2276.

[6] Hammam M. AlGhmadi et al. "An Automated Approach for Recommending When to Stop Performance Tests". In: *Proc. of the International Conference on Software Maintenance and Evolution (ICSME '16)*. IEEE, 2016, pp. 279–289.

[7] Deema Alshoaibi et al. "PRICE: Detection of Performance Regression Introducing Code Changes Using Static and Dynamic Metrics". In: *Proceedings of the 11th International Symposium on Search Based Software Engineering*. SSBSE '19. Springer Nature, 2019. DOI: 10.1007/978-3-030-27455-9_6.

[8] David Ameller et al. "How do Software Architects Consider Non-Functional Requirements: An Exploratory Study". In: *Proceedings of the 20th International Requirements Engineering Conference*. RE '12. IEEE, 2012.

[9] Atakan Aral and Ivona Brandic. "Learning Spatiotemporal Failure Dependencies for Resilient Edge Computing Services". In: *Transactions on Parallel and Distributed Systems* 32.7 (2020), pp. 1578–1590.

[10] Mohammad S. Aslanpour et al. "Serverless Edge Computing: Vision and Challenges". In: *Proc. Australasian Computer Science Week Multiconference (ACSW '21)*. ACM, 2021, pp. 1–10.

[11] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. “REST-ler: automatic intelligent REST API Fuzzing”. In: (2018).

[12] Timon Back and Vasilios Andrikopoulos. “Using a Microbenchmark to Compare Function as a Service Solutions”. In: *Proc. European Conference on Service-Oriented and Cloud Computing (ESOCC ’18)*. Springer, 2018, pp. 146–160.

[13] Daniel Barcelona-Pons and Pedro García-López. “Benchmarking parallelism in FaaS platforms”. In: *Future Generation Computer Systems* 124 (2021), pp. 268–284.

[14] Luciano Baresi and Danilo Filgueira Mendonça. “Towards a Serverless Platform for Edge Computing”. In: *Proc. International Conference on Fog Computing (ICFC ’19)*. IEEE. 2019, pp. 1–10.

[15] D Bermbach and S Tai. “Benchmarking Eventual Consistency: Lessons Learned from Long-Term Experimental Studies”. In: *Proceedings of the 2nd International Conference on Cloud Engineering (IC2E)*. IEEE. 2014.

[16] D. Bermbach, E. Wittern, and S. Tai. *Cloud Service Benchmarking: Measuring Quality of Cloud Services from a Client Perspective*. Springer, 2017.

[17] David Bermbach. “Benchmarking Eventually Consistent Distributed Storage Systems”. PhD thesis. Karlsruhe Institute of Technology, 2014.

[18] David Bermbach. “Quality of Cloud Services: Expect the Unexpected”. In: *IEEE Internet Computing (Invited Paper)*. IEEE, 2017, pp. 68–72.

[19] David Bermbach, Ahmet-Serdar Karakaya, and Simon Buchholz. “Using Application Knowledge to Reduce Cold Starts in FaaS Services”. In: *Proc. 35th ACM Symposium on Applied Computing (SAC ’20)*. ACM, 2020, pp. 134–143.

[20] David Bermbach and Erik Wittern. “Benchmarking Web API Quality”. In: *Proc. of the International Conference on Web Engineering (ICWE '16)*. Springer, 2016, pp. 188–206.

[21] David Bermbach and Erik Wittern. “Benchmarking Web API Quality – Revisited”. In: *Journal of Web Engineering* (2020).

[22] David Bermbach et al. “Towards an Extensible Middleware for Database Benchmarking”. In: *Proc. of the Technology Conference on Performance Evaluation and Benchmarking (TPCTC '14)*. Springer, 2015, pp. 82–96.

[23] David Bermbach et al. “A Research Perspective on Fog Computing”. In: *Proc. 2nd Workshop on IoT Systems Provisioning & Management for Context-Aware Smart Cities (ISYCC 2017)*. Springer, 2017, pp. 198–210.

[24] David Bermbach et al. “BenchFoundry: A Benchmarking Framework for Cloud Storage Services”. In: *Proc. of the International Conference on Service-Oriented Computing (ICSOC '17)*. Springer, 2017, pp. 314–330.

[25] David Bermbach et al. “AuctionWhisk: Using an Auction-Inspired Approach for Function Placement in Serverless Fog Platforms”. In: *Software: Practice and Experience* 52.2 (2021).

[26] David Bermbach et al. “On the Future of Cloud Engineering”. In: *Proc. International Conference on Cloud Engineering (IC2E '21)*. IEEE, 2021, pp. 264–275.

[27] Cor-Paul Bezemer et al. “How is Performance Addressed in DevOps?” In: *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*. ICPE '19. ACM, 2019.

[28] Carsten Binnig et al. “How is the Weather tomorrow? Towards a Benchmark for the Cloud”. In: *Proc. of the International Workshop on Testing Database Systems (DBTest '09)*. ACM, 2009, pp. 1–6.

[29] Amir Hossein Borhani et al. “WPress: An Application-Driven Performance Benchmark For Cloud-Based Virtual Machines”. In: *Proc. of the International Enterprise Distributed Object Computing Conference (EDOC '14)*. IEEE, 2014, pp. 101–109.

[30] Giacomo Brambilla et al. “A Simulation Platform for Large-Scale Internet of ThingsScenarios in Urban Environments”. In: *Proc. International Conference on IoT in Urban Space (URB-IOT '14)*. ICST, 2014, pp. 50–55.

[31] Jake Brutlag. *Speed matters for Google web search*. 2009.

[32] Lubomír Bulej, Vojtěch Horký, and Petr Tůma. “Initial Experiments with Duet Benchmarking: Performance Testing Interference in the Cloud”. In: *Proc. of the International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS '19)*. IEEE, 2019, pp. 249–255.

[33] Lubomír Bulej, Tomáš Kalibera, and Petr Tůma. “Repeated results analysis for middleware regression benchmarking”. In: *Performance Evaluation*. Elsevier, 2005, pp. 345–358.

[34] Lubomír Bulej et al. “Capturing Performance Assumptions Using Stochastic Performance Logic”. In: *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*. ICPE '12. ACM, 2012.

[35] Lubomír Bulej et al. “Unit testing performance with Stochastic Performance Logic”. In: *Automated Software Engineering*. Springer, 2017.

[36] Lubomír Bulej et al. “Duet Benchmarking: Improving Measurement Accuracy in the Cloud”. In: *Proc. of the International Conference on Performance Engineering (ICPE '20)*. ACM, 2020, pp. 100–107.

[37] Mike Burrows. “The Chubby Lock Service for Loosely-coupled Distributed Systems”. In: *Proc. of OSDI*. USENIX Association, 2006.

[38] Andrea Caracciolo, Mircea Filip Lungu, and Oscar Nierstrasz. “How Do Software Architects Specify and Validate Quality Requirements?” In: *European Conference on Software Architecture*. ECSA '14. Springer, 2014.

[39] Fay Chang et al. “Bigtable: A Distributed Storage System for Structured Data”. In: *Proc. of OSDI*. USENIX Association, 2006.

[40] Jinfu Chen and Weiyi Shang. “An Exploratory Study of Performance Regression Introducing Code Changes”. In: *Proc. of the International Conference on Software Maintenance and Evolution (ICSME '17)*. IEEE, 2017, pp. 341–352.

[41] Jinfu Chen, Weiyi Shang, and Emad Shihab. “PerfJIT: Test-level Just-in-time Prediction for Performance Regression Introducing Commits”. In: *IEEE Transactions on Software Engineering*. IEEE, 2020.

[42] TY Chen and MF Lau. “A simulation study on some heuristics for test suite reduction”. In: *Information and software technology*. Elsevier, 1998.

[43] Brian F. Cooper et al. “Benchmarking Cloud Serving Systems with YCSB”. In: *Proc. of the Symposium on Cloud Computing (SOCC '10)*. ACM, 2010, pp. 143–154.

[44] Marcin Copik et al. “SeBS: A Serverless Benchmark Suite for Function-as-a-Service Computing”. In: *Proc. 22nd ACM/IFIP International Middleware Conference (Middleware '21)*. ACM, 2021, pp. 64–78.

[45] Robert Cordingly, Wen Shu, and Wes J Lloyd. “Predicting Performance and Cost of Serverless Computing Functions with SAAF”. In: *Proc. International Conference on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/CyberSciTech '20)*. IEEE. 2020, pp. 640–649.

[46] Robert Cordingly et al. “Implications of Programming Language Selection for Serverless Data Processing Pipelines”. In: *Proc. 6th IEEE International Conference on Cloud and Big Data Computing (CBDCOM '20)*. IEEE, 2020, pp. 704–711.

[47] David Daly. “Creating a Virtuous Cycle in Performance Testing at MongoDB”. In: *Proc. of the International Conference on Performance Engineering (ICPE '21)*. ACM, 2021, pp. 33–41.

[48] David Daly et al. “The Use of Change Point Detection to Identify Software Performance Regressions in a Continuous Integration System”. In: *Proc. of the International Conference on Performance Engineering (ICPE '20)*. ACM, 2020, pp. 67–75.

[49] Diego Elias Damasceno Costa et al. “What’s Wrong With My Benchmark Results? Studying Bad Practices in JMH Benchmarks”. In: *Transactions on Software Engineering*. IEEE, 2019.

[50] AndrÃ© De Camargo et al. “An Architecture to Automate Performance Tests on Microservices”. In: *Proc. of the 18th International Conference on Information Integration and Web-based Applications and Services*. ACM, 2016.

[51] Akon Dey et al. “YCSB+ T: Benchmarking web-scale transactional databases”. In: *Proc. of the International Conference on Data Engineering Workshops (ICDE 2014)*. IEEE, 2014.

[52] Djellel Eddine Difallah et al. “OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases”. In: *Proc. of the International Conference on Very Large Data Bases (VLDB '13)*. VLDB Endowment, 2013, pp. 277–288.

[53] Zishuo Ding, Jinfu Chen, and Weiyi Shang. “Towards the Use of the Readily Available Tests from the Release Pipeline as Performance Tests. Are We There Yet?” In: *Proceedings of the 42nd International Conference on Software Engineering*. ICSE '20. ACM, 2020.

[54] Ted Dunning et al. *Time Series Databases: New Ways to Store and Access Data*. O’Reilly, 2014.

[55] Simon Eismann et al. “The State of Serverless Applications: Collection, Characterization, and Community Consensus”. In: *Transactions on Software Engineering* 48.10 (2021), pp. 4152–4166.

[56] Erwin van Eyk et al. “Beyond Microbenchmarks: The SPEC-RG Vision for A Comprehensive Serverless Benchmark”. In: *Proc. ACM/SPEC International Conference on Performance Engineering Companion (ICPE '20 Companion)*. ACM, 2020, pp. 26–31.

[57] Dror G Feitelson, Eitan Frachtenberg, and Kent L Beck. “Development and deployment at facebook”. In: *IEEE Internet Computing* 17.4 (2013).

[58] Kamil Figiela et al. “Performance evaluation of heterogeneous cloud functions”. In: *Concurrency and Computation: Practice and Experience* 30.23 (2018), pp. 1–16.

[59] Enno Folkerts et al. “Benchmarking in the Cloud: What it Should, Can, and Cannot Be”. In: *Proc. of the Technology Conference on Performance Evaluation and Benchmarking (TPCTC '12)*. Springer, 2013, pp. 173–188.

[60] King Chun Foo et al. “Mining Performance Regression Testing Repositories for Automated Performance Analysis”. In: *Proc. of the International Conference on Quality Software (QSIC '10)*. IEEE, 2010, pp. 32–41.

[61] King Chun Foo et al. “An Industrial Case Study on the Automated Detection of Performance Regressions in Heterogeneous Environments”. In: *Proc. of the International Conference on Software Engineering (ICSE '15)*. IEEE, 2015, pp. 159–168.

[62] Martin Fowler. *Richardson Maturity Model*. 2010. URL: `https://martinfowler.com/articles/richardsonMaturityModel.html`.

[63] Martin Fowler and Matthew Foemmel. “Continuous integration”. In: *Thought-Works* 122 (2006).

[64] Yu Gan et al. “An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems”. In: *Proc. of the Twenty-Fourth*

International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'19). ACM, 2019.

[65] Gareth George et al. "NanoLambda: Implementing Functions as a Service at All Resource Scales for the Internet of Things". In: *Proc. ACM/IEEE Symposium on Edge Computing (SEC '20)*. IEEE, 2020, pp. 220–231.

[66] Andy Georges, Dries Buytaert, and Lieven Eeckhout. "Statistically Rigorous Java Performance Evaluation". In: *Proceedings of the 22nd ACM SIGPLAN Conference on Object-Oriented Programming, Systems, and Applications*. OOPSLA 2007. Montreal, Quebec, Canada: Association for Computing Machinery (ACM), 2007, pp. 57–76. ISBN: 978-1-59593-786-5. DOI: 10.1145/1297027.1297033.

[67] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. "The Google File System". In: *Proc. of SOSP*. ACM, 2003.

[68] Martin Grambow, Jonathan Hasenburg, and David Bermbach. "Public Video Surveillance: Using the Fog to Increase Privacy". In: *Proc. 5th Workshop on Middleware and Applications for the Internet of Things (M4IoT '18)*. ACM, 2018, pp. 11–14.

[69] Martin Grambow, Fabian Lehmann, and David Bermbach. "Continuous Benchmarking: Using System Benchmarking in Build Pipelines". In: *Proc. of the Workshop on Service Quality and Quantitative Evaluation in new Emerging Technologies (SQUEET '19)*. IEEE, 2019, pp. 241–246.

[70] Martin Grambow, Erik Wittern, and David Bermbach. "Benchmarking the Performance of Microservice Applications". In: *SIGAPP Applied Computing Review*. ACM, 2020, pp. 20–34.

[71] Martin Grambow et al. "Is it Safe to Dockerize my Database Benchmark?" In: *Proc. of the ACM Symposium on Applied Computing, Posters Track (SAC '19)*. ACM, 2019, pp. 341–344.

[72] Martin Grambow et al. "Benchmarking Microservice Performance: A Pattern-based Approach". In: *Proc. of the 35th ACM Symposium on Applied Computing (SAC 2020)*. ACM, 2020.

[73] Martin Grambow et al. "BeFaaS: An Application-Centric Benchmarking Framework for FaaS Platforms". In: *Proc. 9th IEEE International Conference on Cloud Engineering (IC2E '21)*. IEEE, 2021, pp. 1–8.

[74] Martin Grambow et al. "Using application benchmark call graphs to quantify and improve the practical relevance of microbenchmark suites". In: *PeerJ Computer Science*. PeerJ, 2021.

[75] Martin Grambow et al. "Using Microbenchmark Suites to Detect Application Performance Changes". In: *Transactions on Cloud Computing*. IEEE, 2023.

[76] Jonathan Hasenburg, Martin Grambow, and David Bermbach. “MockFog 2.0: Automated Execution of Fog Application Experiments in the Cloud”. In: *IEEE Transactions on Cloud Computing*. IEEE, 2021.

[77] Jonathan Hasenburg et al. “MockFog: Emulating Fog Computing Infrastructure in the Cloud”. In: *Proceedings of the First IEEE International Conference on Fog Computing 2019*. ICFC '19. IEEE, 2019.

[78] Sen He et al. “A Statistics-Based Performance Testing Methodology for Cloud Applications”. In: *Proc. of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19)*. ACM, 2019, pp. 188–199.

[79] Christoph Heger, Jens Happe, and Roozbeh Farahbod. “Automated Root Cause Isolation of Performance Regressions During Software Development”. In: *Proc. of the International Conference on Performance Engineering (ICPE '13*. ACM, 2013, pp. 27–38.

[80] Vojtěch Horký et al. “Utilizing Performance Unit Tests To Increase Performance Awareness”. In: *Proceedings of the 6th International Conference on Performance Engineering*. ICPE '15. ACM, 2015.

[81] Peng Huang et al. “Performance Regression Testing Target Prioritization via Performance Risk Analysis”. In: *Proceedings of the 36th International Conference on Software Engineering*. ICSE '14. ACM, 2014.

[82] Jez Humble and David Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley Professional, 2010.

[83] Karl Huppler. “The Art of Building a Good Benchmark”. In: *Proc. of the Technology Conference on Performance Evaluation and Benchmarking (TPCTC '09)*. Springer, 2009, pp. 18–30.

[84] *InfluxDB 2.0*. `https://github.com/influxdata/influxdb/tree/2.0`. InfluxData Inc., 2021.

[85] Henrik Ingo and David Daly. “Automated System Performance Testing at MongoDB”. In: *Proc. of the workshop on Testing Database Systems (DBTest '20)*. ACM, 2020, pp. 1–6.

[86] Alexandru Iosup, Nezih Yigitbasi, and Dick Epema. “On the Performance Variability of Production Cloud Services”. In: *Proc. of the International Symposium on Cluster, Cloud and Grid Computing (CCGRID '11)*. IEEE, 2011, pp. 104–113.

[87] Ana Ivanchikj, Ilija Gjorgjiev, and Cesare Pautasso. “RESTalk Miner: Mining RESTful Conversations, Pattern Discovery and Matching”. In: *Proc. of International Conference on Service-Oriented Computing - Workshops (ICSOC 2018)*. Springer, 2018.

[88] Omar Javed et al. “PerfCI: A Toolchain for Automated Performance Testing during Continuous Integration of Python Projects”. In: *Proc. of the International Conference on Automated Software Engineering (ASE '20)*. IEEE, 2020, pp. 1344–1348.

[89] Zhen Ming Jiang and Ahmed E. Hassan. “A Survey on Load Testing of Large-Scale Software Systems”. In: *Transactions on Software Engineering*. IEEE, 2015, pp. 1091–1118.

[90] Tomas Kalibera and Richard Jones. *Quantifying Performance Changes with Effect Size Confidence Intervals*. 2020. URL: https://arxiv.org/abs/2007.10899.

[91] Chia Hung Kao, Chun Cheng Lin, and Juei-Nan Chen. “Performance Testing Framework for REST-based Web Applications”. In: *13th International Conference on Quality Software*. IEEE, 2013.

[92] Alexander Keller and Heiko Ludwig. “The WSLA framework: Specifying and monitoring service level agreements for web services”. In: *Journal of Network and Systems Management* 11.1 (2003), pp. 57–81.

[93] Jeongchul Kim and Kyungyong Lee. “FunctionBench: A Suite of Workloads for Serverless Cloud Function Service”. In: *Proc. 12th IEEE International Conference on Cloud Computing (CLOUD '19)*. IEEE, 2019, pp. 502–504.

[94] Jóakim v. Kistowski et al. “How to Build a Benchmark”. In: *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering (ICPE 2015)*. ACM, 2015, pp. 333–336. DOI: 10.1145/2668930.2688819.

[95] M. Klems, D. Bermbach, and R. Weinert. “A Runtime Quality Measurement Framework for Cloud Database Service Systems”. In: *Proceedings of the 8th International Conference on the Quality of Information and Communications Technology (QUATIC)*. 2012.

[96] Markus Klems, Michael Menzel, and Robin Fischer. “Consistency Benchmarking: Evaluating the Consistency Behavior of Middleware Services in the Cloud”. In: *Service-Oriented Computing*. Ed. by Paul Maglio et al. Vol. 6470. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010, pp. 627–634. ISBN: 978-3-642-17357-8. DOI: 10.1007/978-3-642-17358-5_48. URL: http://dx.doi.org/10.1007/978-3-642-17358-5_48.

[97] Jörn Kuhlenkamp, Markus Klems, and Oliver Röss. “Benchmarking Scalability and Elasticity of Distributed Database Systems”. In: *Proc. of the International Conference on Very Large Databases (VLDB '14)*. VLDB Endowment, 2014, pp. 1219–1230.

[98] Tobias Kurze et al. “Cloud Federation”. In: *Proc. 2nd International Conference on Cloud Computing, GRIDs, and Virtualization*. IARIA, 2011, pp. 32–38.

[99] Christoph Laaber, Mikael Basmaci, and Pasquale Salza. “Predicting unstable software benchmarks using static source code features”. In: *Empirical Software Engineering*. Springer, 2021, pp. 1–53.

[100] Christoph Laaber, Harald C Gall, and Philipp Leitner. "Applying test case prioritization to software microbenchmarks". In: *Empirical Software Engineering*. Springer, 2021, pp. 1–48.

[101] Christoph Laaber and Philipp Leitner. "An Evaluation of Open-Source Software Microbenchmark Suites for Continuous Performance Assessment". In: *Proc. of the .International Conference on Mining Software Repositories (MSR '18)*. ACM, 2018, pp. 119–130.

[102] Christoph Laaber, Joel Scheuner, and Philipp Leitner. "Software Microbenchmarking in the Cloud. How Bad is it Really?" In: *Empirical Software Engineering*. Springer, 2019, pp. 2469–2508.

[103] Christoph Laaber et al. "Dynamically Reconfiguring Software Microbenchmarks: Reducing Execution Time without Sacrificing Result Quality". In: *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2020. Association for Computing Machinery (ACM), Nov. 2020, pp. 989–1001. DOI: 10.1145/3368089.3409683.

[104] Hyungro Lee, Kumar Satyam, and Geoffrey Fox. "Evaluation of Production Serverless Computing Environments". In: *Proc. 11th IEEE International Conference on Cloud Computing (CLOUD '18)*. IEEE, 2018, pp. 442–450.

[105] Philipp Leitner and Cor-Paul Bezemer. "An Exploratory Study of the State of Practice of Performance Testing in Java-Based Open Source Projects". In: *Proc. of the International Conference on Performance Engineering (ICPE '17)*. ACM, 2017, pp. 373–384.

[106] Philipp Leitner and Jürgen Cito. "Patterns in the Chaos - A Study of Performance Variation and Predictability in Public IaaS Clouds". In: *Transactions on Internet Technology*. ACM, 2016, pp. 1–23.

[107] Alexander Lenk et al. "What are you paying for? performance benchmarking for infrastructure-as-a-service offerings". In: *Cloud Computing (CLOUD), 2011 IEEE International Conference on*. IEEE. 2011, pp. 484–491.

[108] James Lewis and Martin Fowler. *Microservices*. 2014. URL: https://martinfowler.com/articles/microservices.html.

[109] Heiko Ludwig et al. "Web service level agreement (WSLA) language specification". In: *Ibm corporation* (2003).

[110] Ivan Lujic et al. "Increasing Traffic Safety with Real-Time Edge Analytics and 5G". In: *Proc. 4th International Workshop on Edge Systems, Analytics and Networking (EdgeSys '21)*. ACM, 2021, pp. 19–24.

[111] Qi Luo et al. "How Do Static and Dynamic Test Case Prioritization Techniques Perform on Modern Software Systems? An Extensive Study on GitHub Projects". In: *IEEE Transactions on Software Engineering*. IEEE, 2018.

[112] Vincenzo De Maio, David Bermbach, and Ivona Brandic. “TAROT: Spatio-Temporal Function Placement for Serverless Smart City Applications”. In: *Proc. International Conference on Utility and Cloud Computing (UCC ’22)*. 2022, pp. 21–30.

[113] Pascal Maissen et al. “FaaSdom: A Benchmark Suite for Serverless Computing”. In: *Proc. 14th ACM International Conference on Distributed and Event-based Systems (DEBS ’20)*. ACM, 2020, pp. 73–84.

[114] Maciej Malawski et al. “Benchmarking Heterogeneous Cloud Functions”. In: *Proc. European Conference on Parallel Processing (Euro-Par 2017)*. Springer, 2017, pp. 415–426.

[115] Johannes Manner et al. “Cold Start Influencing Factors in Function as a Service”. In: *Proc. 11th IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion ’18)*. IEEE, 2018, pp. 181–188.

[116] Horácio Martins, Filipe Araujo, and Paulo Rupino da Cunha. “Benchmarking serverless computing platforms”. In: *Journal of Grid Computing* 18 (2020), pp. 691–709.

[117] David S Matteson and Nicholas A James. “A Nonparametric Approach for Multiple Change Point Analysis of Multivariate Data”. In: *Journal of the American Statistical Association*. Taylor & Francis, 2014, pp. 334–345.

[118] Jonathan McChesney et al. “DeFog: Fog Computing Benchmarks”. In: *Proc. 5th ACM/IEEE Symposium on Edge Computing (SEC ’19)*. ACM, 2019, pp. 47–58.

[119] Malcolm D McIlroy, EN Pinson, and BA Tague. “UNIX Time-Sharing System: Foreword”. In: *Bell System Technical Journal* 57.6 (1978).

[120] Nagy Mostafa and Chandra Krintz. “Tracking Performance Across Software Revisions”. In: *Proceedings of the International Conference on Principles and Practice of Programming in Java (PPPJ '09)*. ACM, 2009, pp. 162–171.

[121] Shaikh Mostafa, Xiaoyin Wang, and Tao Xie. “PerfRanker: Prioritization of Performance Regression Tests for Collection-Intensive Sotware”. In: *Proc. of the International Symposium on Software Testing and Analysis (ISSTA '17)*. ACM, 2017, pp. 23–34.

[122] Steffen Müller et al. “Benchmarking the Performance Impact of Transport Layer Security in Cloud Database Systems”. In: *Proc. of the International Conference on Cloud Engineering (IC2E '14)*. IEEE, 2014, pp. 27–36.

[123] Thanh H. D. Nguyen et al. “An Industrial Case Study of Automatically Identifying Performance Regression-Causes”. In: *Proc. of the Working Conference on Mining Software Repositories (MSR '14)*. ACM, 2014, pp. 232–241.

[124] Augusto Born de Oliveira et al. “Perphecy: Performance Regression Test Selection Made Simple but Effective”. In: *Proc. of the International Conference on Software Testing, Verification and Validation (ICST '17)*. IEEE, 2017, pp. 103–113.

[125] Frank Pallas, Philip Raschke, and David Bermbach. "Fog Computing as Privacy Enabler". In: *IEEE Internet Computing* 24.4 (2020), pp. 15–21. DOI: 10.1109/MIC.2020.2979161.

[126] Frank Pallas et al. "Evidence-based security configurations for cloud datastores". In: *Proc. of SAC*. ACM. 2017.

[127] Pallas, Frank and Günther, Johannes and Bermbach, David. "Pick your Choice in HBase: Security or Performance". In: *Proc. of the International Conference on Big Data (Big Data '16)*. IEEE, 2017, pp. 548–554.

[128] Ioannis Papapanagiotou and Vinay Chella. "NDBench: Benchmarking Microservices at Scale". In: *arXiv e-prints* (2018).

[129] Swapnil Patil et al. "YCSB++: Benchmarking and Performance Debugging Advanced Features in Scalable Table Stores". In: *Proceedings of the 2nd Symposium on Cloud Computing (SOCC)*. SOCC '11. Cascais, Portugal: ACM, 2011, 9:1–9:14. ISBN: 978-1-4503-0976-9. DOI: 10.1145/2038916.2038925. URL: http://doi.acm.org/10.1145/2038916.2038925.

[130] Tobias Pfandzelter and David Bermbach. "tinyFaaS: A Lightweight FaaS Platform for Edge Environments". In: *Proc. IEEE International Conference on Fog Computing (ICFC '20)*. IEEE, 2020, pp. 17–24.

[131] Tobias Pfandzelter, Jonathan Hasenburg, and David Bermbach. "Towards a Computing Platform for the LEO Edge". In: *Proc. 4th International Workshop on Edge Systems, Analytics and Networking (EdgeSys '21)*. ACM, 2021, pp. 43–48.

[132] Tobias Pfandzelter et al. "Streaming vs. Functions: A Cost Perspective on Cloud Event Processing". In: *Proc. 10th IEEE International Conference on Cloud Engineering (IC2E '22)*. IEEE, 2022, pp. 67–78.

[133] Michael Pradel, Markus Huggler, and Thomas R. Gross. "Performance Regression Testing of Concurrent Classes". In: *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ISSTA '14. ACM, 2014.

[134] Tilmann Rabl et al. "A Data Generator for Cloud-Scale Benchmarking". In: *Proc. of the Technology Conference on Performance Evaluation and Benchmarking (TPCTC '10)*. Springer, 2010, pp. 41–56.

[135] Brian Ramprasad, Joydeep Mukherjee, and Marin Litoiu. "A Smart Testing Framework for IoT Applications". In: *Proc. 11th IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion '18)*. IEEE, 2018, pp. 252–257.

[136] Alex Rodriguez. "Restful web services: The basics". In: *IBM developerWorks* 33 (2008).

[137] Marcelino Rodriguez-Cancio, Benoit Combemale, and Benoit Baudry. "Automatic Microbenchmark Generation to Prevent Dead Code Elimination and Constant Folding". In: *Proceedings of the 31st International Conference on Automated Software Engineering*. ASE '16. ACM, 2016.

[138] Gregg Rothermel et al. “Test Case Prioritization: An Empirical Study”. In: *Proc. of the International Conference on Software Maintenance (ICSM '10)*. IEEE, 1999, pp. 179–188.

[139] Juan Pablo Sandoval Alcocer, Alexandre Bergel, and Marco Tulio Valente. “Learning from Source Code History to Identify Performance Failures”. In: *Proceedings of the 7th International Conference on Performance Engineering*. ICPE '16. ACM, 2016.

[140] Juan Pablo Sandoval Alcocer, Alexandre Bergel, and Marco Tulio Valente. “Prioritizing versions for performance regression testing: The Pharo case”. In: *Science of Computer Programming*. Elsevier, 2020.

[141] Jörg Schad, Jens Dittrich, and Jorge-Arnulfo Quiané-Ruiz. “Runtime Measurements in the Cloud: Observing, Analyzing, and Reducing Variance”. In: *Proc. of the International Conference on Very Large Data Bases (VLDB '10)*. VLDB Endowment, 2010, pp. 460–471.

[142] G. Schermann, J. Cito, and P. Leitner. “Continuous Experimentation: Challenges, Implementation Techniques, and Current Research”. In: *IEEE Software*. IEEE, 2018.

[143] Gerald Schermann et al. “Bifrost: supporting continuous deployment with automated enactment of multi-phase live testing strategies”. In: *Proc. of Middleware*. ACM. 2016.

[144] Joel Scheuner et al. “Cloud WorkBench – Infrastructure-as-Code Based Cloud Benchmarking”. In: *Proc. of the International Conference on Cloud Computing Technology and Science (CloudCom 2014)*. IEEE, 2014.

[145] Joel Scheuner et al. “CrossFit: Fine-grained Benchmarking of Serverless Application Performance across Cloud Providers”. In: *Proc. 15th IEEE/ACM International Conference on Utility and Cloud Computing (UCC '22)*. IEEE, 2022, pp. 51–60.

[146] Joel Scheuner et al. *Let's Trace It: Fine-Grained Serverless Benchmarking using Synchronous and Asynchronous Orchestrated Applications*. 2022. arXiv: 2205.07696.

[147] Joel Scheuner et al. “TriggerBench: A Performance Benchmark for Serverless Function Triggers”. In: *Proc. 10th IEEE International Conference on Cloud Engineering (IC2E '22)*. IEEE, 2022, pp. 96–103.

[148] Trever Schirmer et al. “The Night Shift: Understanding Performance Variability of Cloud Serverless Platforms”. In: *Proc. 1st Workshop on SErverless Systems, Applications and MEthodologies (SESAME '23)*. ACM, 2023, pp. 27–33.

[149] Mohammad Shahrad, Jonathan Balkind, and David Wentzlaff. “Architectural Implications of Function-as-a-Service Computing”. In: *Proc. 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '52)*. ACM, 2019, pp. 1063–1075.

[150] Mohammad Shahrad et al. “Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider”. In: *Proc. USENIX Annual Technical Conference (USENIX ATC '20)*. USENIX, 2020, pp. 205–218.

[151] Benjamin H. Sigelman et al. *Dapper, a Large-Scale Distributed Systems Tracing Infrastructure*. Tech. rep. 20101. Google, 2010.

[152] Marcio Silva et al. "Cloudbench: Experiment Automation for Cloud Environments". In: *Proc. of the International Conference on Cloud Engineering (IC2E '13)*. IEEE, 2013, pp. 302–311.

[153] Nikhila Somu et al. "PanOpticon: A Comprehensive Benchmarking Tool for Serverless Applications". In: *Proc. 12th International Conference on COMmunication Systems & NETworkS (COMSNETS '20)*. IEEE, 2020, pp. 144–151.

[154] Petr Stefan et al. "Unit Testing Performance in Java Projects: Are We There Yet?" In: *Proceedings of the 8th International Conference on Performance Engineering*. ICPE '17. ACM, 2017.

[155] Chunqiang Tang et al. "Holistic configuration management at Facebook". In: *Proc. of SOSP*. ACM. 2015.

[156] Mert Toslali et al. "Iter8: Online Experimentation in the Cloud". In: *Proc. of the Symposium on Cloud Computing (SoCC '21)*. ACM, 2021, pp. 289–304.

[157] Dmitrii Ustiugov, Theodor Amariucai, and Boris Grot. "Analyzing Tail Latency in Serverless Clouds with STeLLAR". In: *Proc. IEEE International Symposium on Workload Characterization (IISWC '21)*. IEEE, 2021, pp. 51–62.

[158] Alexandru Uta et al. "Is Big Data Performance Reproducible in Modern Cloud Networks?" In: *USENIX Symposium on Networked Systems Design and Implementation (NSDI '20)*. USENIX, 2020, pp. 513–527.

[159] *VictoriaMetrics*. `https://github.com/VictoriaMetrics/VictoriaMetrics`. Victoria Metrics Inc, 2021.

[160] Jan Waller, Nils C Ehmke, and Wilhelm Hasselbring. "Including Performance Benchmarks into Continuous Integration to Enable DevOps". In: *Software Engineering Notes*. ACM, 2015, pp. 1–4.

[161] Liang Wang et al. "Peeking Behind the Curtains of Serverless Platforms". In: *Proc. USENIX Annual Technical Conference (USENIX ATC '18)*. USENIX, 2018, pp. 133–145.

[162] Tianyi Yu et al. "Characterizing Serverless Platforms with ServerlessBench". In: *Proc. 11th ACM Symposium on Cloud Computing (SoCC '20)*. ACM, 2020, pp. 30–44.

[163] Shahed Zaman, Bram Adams, and Ahmed E Hassan. "Security Versus Performance Bugs: A Case Study on Firefox". In: *Proc. of the Working Conference on Mining Software Repositories (MSR '11)*. ACM, 2011, pp. 93–102.

[164] Shahed Zaman, Bram Adams, and Ahmed E. Hassan. "A Qualitative Study on Performance Bugs". In: *Proc. of the Working Conference on Mining Software Repositories (MSR '12)*. IEEE, 2012, pp. 199–208.

[165] Ben Zhang et al. “The Cloud is Not Enough: Saving IoT from the Cloud”. In: *Proc. USENIX Workshop on Hot Topics in Cloud Computing (HotCloud ’15)*. USENIX, 2015, pp. 1–7.

[166] Haidong Zhao et al. “Supporting Multi-Cloud in Serverless Computing”. In: *Proc. 15th IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC ’22)*. IEEE, 2022, pp. 285–290.

[167] Qing Zheng et al. “Cosbench: A benchmark tool for cloud object storage services”. In: *Proc. of the International Conference on Cloud Computing (CLOUD 2012)*. IEEE, 2012.

Last note