Université de Lille

Graduate School MADIS-631

Reconciling performance and efficient use of hardware resources: the case of configurable software services

THESIS

to obtain the degree of:

DOCTOR OF PHILOSOPHY

in COMPUTER SCIENCE

by

***Alexandre* BONVOISIN**

Thesis defended on 5 December 2025, before a committee composed of:

Reviewers			
	Prof.	*Sophie* CHABRIDON	Télécom SudParis
	Prof.	*Adel* NOUREDDINE	University Paris Nanterre
Chair			
	Prof.	*Anne* ETIEN	University of Lille
Examiner			
	Prof.	*Olivier* BARAIS	University of Rennes
Thesis Supervisors			
	Assoc. Prof.	*Clément* QUINTON	University of Lille
	Prof.	*Romain* ROUVOY	University of Lille

CRIStAL Centre de Recherche en Informatique, Signal et Automatique de Lille

CRIStAL Laboratory

Inria

Inria Centre at the University of Lille

Université de Lille

Ecole Graduée MADIS-631

Concilier performance et utilisation efficiente des ressources matérielles: le cas des services logiciels configurables

THÈSE

pour obtenir le grade de:

DOCTEUR

dans la spécialité INFORMATIQUE

par

***Alexandre* BONVOISIN**

Thèse soutenue le 5 Décembre 2025 devant le jury composé de:

Rapporteurs			
	Pr.	*Sophie* CHABRIDON	Télécom SudParis
	Pr.	*Adel* NOUREDDINE	Université Paris Nanterre
Présidente du jury			
	Pr.	*Anne* ETIEN	Université de Lille
Examinateur			
	Pr.	*Olivier* BARAIS	Université de Rennes
Directeurs de Thèse			
	MCF.	*Clément* QUINTON	Université de Lille
	Pr.	*Romain* ROUVOY	Université de Lille

CRIStAL
Centre de Recherche en Informatique, Signal et Automatique de Lille

Laboratoire CRIStAL

Inria

Centre Inria de l'Université de Lille

Acknowledgements

I would like to express my deep gratitude to everyone who made the completion of this thesis possible.

I am grateful to my supervisors, Romain Rouvoy and Clément Quinton, for their invaluable support, insightful advice, and patience throughout this journey.

My thanks also go to all members of the thesis committee for their expertise and the attention they gave to my presentation. I thank Anne Etien for having chaired this committee with kindness and rigor. Sophie Chabridon and Adel Noureddine for devoting their time to reviewing this manuscript and for their valuable feedback. Finally, I thank Olivier Barais for the interest shown in my research work and for the depth of our exchanges.

I am also grateful to the teachers who have contributed to my training and supported me throughout my university studies. I would like to express my special thanks to Jean-Christophe Routier and Cedric Dumoulin for their availability and our enriching discussions.

I do not forget my many colleagues in the Spirals team, with whom I have been able to collaborate, share a wealth of knowledge, and enjoy countless moments.

Lastly, I would like to thank my loved ones for their support and constant encouragement to pursue my projects over all the years.

Abstract

Information and Communications Technologies (ICTs) are now embedded in almost every aspect of daily life. They shape how we communicate, work, and manage our leisure activities. A smartphone, for example, provides instant access to friends, information, and applications to manage our homes, while smartwatches facilitate real-time fitness tracking and health monitoring.

This digital transformation has been made possible by many technological advances, including progress in electronic component design, improvements in connectivity, and the lowering of computing costs. The latter was largely achieved by the rise of *cloud computing*, which enabled on-demand scalability and cost flexibility by aggregating computing resources in large data centers. Nevertheless, the resulting increase in use brought by these advances and cost reductions has also heightened many environmental impacts.

For example, the extraction of raw materials such as cobalt used in electronic components and lithium in batteries requires large amounts of water, demands significant electrical power, and generates greenhouse-gas emissions. While these mining activities already pose serious environmental challenges, it is also essential to account for the manufacturing, transport, and operation stages as additional sources of environmental impacts.

To address these concerns, initiatives are focusing on designing energy-efficient hardware components, improving cooling systems in data centers, extending the lifespan of equipment, and using renewable resources. Although these solutions are promising, they remain insufficient to eliminate all sources of pollution, especially those that arise from how software applications interact with hardware components.

This thesis therefore addresses the problem of reducing the environmental impact of software deployed in data centers by investigating how their configurations and development practices shape their performance and resource usage.

Through three contributions, this work examines several aspects of software engineering and identifies actionable levers that industry actors can use to develop software with a minimal environmental footprint. The first contribution, examines different data access library and their configurations to understand the trade-offs between targeted service quality and energy consumption. Building on that work, the second contribution explores how

application-framework selection, runtime configuration, and compilation strategy decisions shapes energy consumption, memory footprint, and raw performance of the designed system. Finally, the third contribution analyzes the interrelationships among software indicators, such as execution time and energy consumption to identify and facilitate software evaluation, which remains a challenging task in the light of the growing complexity of software systems.

Résumé

Les Technologies de l'Information et de la Communication (TIC) sont désormais présentes dans presque tous les aspects de la vie quotidienne. Elles guident notre façon de communiquer, de travailler et de gérer notre temps. Un smartphone, par exemple, offre un accès instantané à nos amis, à l'information et aux applications qui permettent de gérer nos loisirs, tandis que les montres intelligentes facilitent le suivi en temps réel de notre activité sportive et la surveillance de notre santé.

Cette transformation digitale a été rendue possible par de nombreuses avancées technologiques, incluant les progrès dans la conception de composants électroniques, les améliorations de la connectivité réseau et la réduction des coûts d'accès à des infrastructures de calcul. Ce dernier point a été notamment rendu possible grâce à l'essor du *cloud computing*, qui a facilité l'accès à des ressources de calcul à la demande et a introduit une certaine flexibilité tarifaire en regroupant les ressources informatiques dans de grands centres de données. Néanmoins, l'utilisation croissante des technologies découlant de ces avancées et la baisse des coûts a également eu pour conséquence d'intensifier de nombreuses atteintes à l'environnement.

Par exemple, l'extraction de matières premières comme le cobalt utilisé dans les composants électronique et le lithium dans les batteries nécessite de grandes quantités d'eau, requière une puissance électrique importante et génère des émissions de gaz à effet de serre. Au-delà de ces enjeux relatifs à l'extraction minière, il ne faut toutefois pas négliger les étapes de fabrication, de transport et d'exploitation comme sources additionnelles d'impacts environnementaux.

Pour répondre à ces préoccupations, les initiatives s'orientent vers la conception de composants à haute efficacité énergétique, l'amélioration des systèmes de refroidissement dans les centres de données, la prolongation de la durée de vie des équipements et l'utilisation de ressources renouvelables. Cependant, ces solutions matérielles, bien que prometteuses, ne suffisent pas à éliminer toutes les sources de pollution, notamment celles qui découlent de la manière dont les équipements sont exploités par les logiciels.

Cette thèse aborde ainsi la problématique de la réduction de l'impact environnemental des logiciels déployés au sein des centres de données, en explorant comment leurs configurations

et les pratiques de développement façonnent leurs performances et leur utilisation des ressources matérielles.

Pour cela, ce travail de recherche présente trois contributions scientifiques qui examinent plusieurs aspects de la conception logicielle et qui mettent en évidence différents leviers actionnables par les acteurs de cette industrie afin de créer des programmes informatiques ayant une empreinte environnementale minimale. La première contribution étudie différentes bibliothèques d'accès aux données et leurs configurations afin de comprendre les compromis entre la consommation d'énergie et une qualité de service ciblée. Sur la base de ce travail, la deuxième contribution analyse l'impact du choix d'un framework applicatif, des paramètres de configuration et des stratégies de compilation sur la consommation d'énergie, l'empreinte mémoire et la performance brute du système conçu. Enfin, la troisième contribution explore les relations entre des indicateurs logiciels, tels que le temps d'exécution et la consommation d'énergie, pour faciliter leur évaluation qui demeure une tâche exigeante au regard de la complexité croissante de ces systèmes.

Table of contents

List of figures

List of tables

Chapter 1

Introduction

Cloud computing centralizes computing resources in data centers, reducing infrastructure costs, simplifying maintenance, and allowing on-demand provisioning. While this facilitates the launch of new projects and the boost of software life cycles, it also results in a rebound in the use of digital technologies, leading to the creation of new data centers [137, 105] and unprecedented growth in energy consumption [59, 62], carbon emission [14, 43] and water usage [78].

To address these environmental concerns, the research community and industry stakeholders have already proposed different strategies, including designing energy-efficient hardware [57, 79], extending equipment lifespans [122, 129], and optimizing infrastructure utilization [68, 58, 110, 136]. Power capping [65] and resource over-subscription [7, 50] are also notable methods that are commonly implemented by cloud providers as infrastructure-level optimization to limit power draw, and consolidate underused equipment by allocating more logical resources than physically available.

Another innovative approach explored by researchers to reduce carbon emissions related to energy supply is to relocate workloads across space and time [131, 135, 105, 119, 121]. By shifting compute tasks between geographically distributed data centers and by scheduling execution windows to align with peaks in sustainable energy supply, it becomes possible to maximize the use of renewable energy without compromising service levels. However, these approaches have implications for data center infrastructures: they must be sized to accommodate increased demands during periods of abundant carbon-free power, requiring additional investments (e.g., batteries, servers) whose manufacture is a source of carbon emissions [1, 137].

Although these mitigation measures are encouraging, they only tackle one aspect of the challenge. Another aspect lies in understanding the role of software in driving resource usage, specifically when they are deployed on these cloud platforms. Consequently, attention

should be given to any approach that contributes to reducing overall resource consumption and limiting the manufacturing of new servers [67].

This insight is supported by several studies demonstrating that software-level design choices directly influence the energy consumption of the hosting hardware [98, 84, 38, 116]. While these research efforts have been active research certain challenges and knowledge gaps remain.

Building on this growing body of literature, this thesis investigates software stacks and engineering practices to identify and understand the sources of energy inefficiency and sheds light on how configuration is a key factor in balancing resource usage and performance.

In the remainder of this chapter, Section 1.1 outlines the problem and challenges that motivated this thesis. Next, in Section 1.2, we summarize our contributions that address the identified problem. In Section 1.3, we list the publications and future submissions that resulted from this research effort. Finally, Section 1.4 describes the outline of this document and introduces each chapter of this document.

1.1 Problem Statement

As previously introduced, reducing the environmental footprint of software requires that hardware resource management be integrated from the earliest stages of the development cycle. However, many developers remain unaware of the consequences of their design choices and still lack tools, metrics, or guidelines to guide their decisions [71, 91].

Moreover, assessing the efficiency of a piece of software is increasingly challenging because systems are massively distributed, parallelized, and expose numerous configuration parameters. Each configuration option introduces an additional trade-off, compelling developers to anticipate and manage a large number of usage scenarios.

In this context, identifying an optimal configuration that best utilizes hardware resources while delivering the highest quality of service is a demanding task. It is also important to consider that evaluating a configuration is also a complex process, as it involves determining which metrics are relevant, dealing with metrics that are hard to measure, and building a test environment that reflects the real deployment. For instance, while it is easy to determine the size of a compiled binary, assessing the energy consumption associated with executing that binary is far more involved.

This thesis therefore, explores the question: How can we reconcile performance with efficient use of hardware resources in the context of highly configurable software services?

1.2 Contributions

This section summarizes our contributions that aim to answer the research question formulated in the previous section.

1.2.1 First Contribution

This contribution aims to determine whether *Object-Relational Mapping* (ORM) frameworks prove to be energy-efficient solutions for database interaction.

Motivation To address the object-relational impedance mismatch between how relationships are represented in the object-oriented paradigm and how they are managed in relational databases, developers have increasingly adopted *Object-Relational Mapping* (ORM) frameworks. Although such frameworks simplify data access and reduce development effort by abstracting database interactions, their influence on energy consumption remains underexplored.

Method Java remains a dominant platform for large-scale enterprise applications [42], largely because of its mature ecosystem and comprehensive database connectivity libraries. Among the many technical specifications available in this platform, the *Jakarta Persistence API* (JPA) standard offers programming interfaces to manage relational data without writing explicit *Structured Query Language* (SQL) queries, thereby enabling an extensive analysis of existing data access strategies. Our experimental framework is built around the *Transaction Processing Performance Council benchmark C* (TPC-C), an industry standard for evaluating relational database performance. The database transactions specified in this benchmark were implemented in Java applications using either hand-crafted database queries or data access strategies provided by JPA-compatible ORM frameworks (EclipseLink or Hibernate). The performance and energy consumption of these applications were then measured while running a range of workload profiles on a PostgreSQL database. For the ORM implementations, we also configured and measured a set of configuration parameters to quantify how these settings influence the balance between performance and energy consumption.

Outcomes The key contributions of this work are threefold. First, we provide a comprehensive analysis of how different ORM solutions affect both performance and energy consumption across a variety of workloads, demonstrating that the selection of an ORM framework is a decisive factor for the overall efficiency of a system. Second, we have identified preferred programming interfaces and configuration options to improve the energy

efficiency of the developed systems. Third, we make all experimental artifacts—including the source code, measurement scripts, and the full set of performance-energy data—publicly available, thereby supporting reproducibility and enabling further investigation by the research community [9].

1.2.2 Second Contribution

This contribution aims to quantify the individual and combined impacts of application framework selection, compilation strategy, and system configuration on application performance and resource usage.

Motivation Application frameworks are widely embraced for their productivity gains, but the additional abstraction layers they introduce often incur performance and energy overheads [13, 73]. Inefficiencies can also arise from the runtime environment, including how instructions are processed and how system settings are tuned. The evolution of programming platforms to support diverse compilation strategies illustrates the importance of these runtime factors. Practical examples include the GraalVM Native Image tool (and its alternatives) [132, 89, 70], which enable *Ahead-Of-Time* (AOT) compilation of Java applications (*i.e.*, to compile software into environment-specific binaries that runs without a standard *Java Virtual Machine* (JVM)), and numerous Python compilers [118]. Beyond the immediate gains promised by the different compilation techniques, trade-offs between performance and resource usage driven by runtime-configuration choices remain unclear. For instance, selecting a particular garbage-collector policy in a memory-managed runtime determines when and how memory is reclaimed, thereby shaping the application's memory usage. Similarly, tuning the number of database connections in a distributed environment controls how many requests can be processed in parallel, directly influencing both latency and resource utilization.

Method Building on the knowledge and experimental foundation established in the first contribution, this second work also relied on the *Transaction Processing Performance Council benchmark C* (TPC-C) and technologies from the Java platform to evaluate the implications of the aforementioned design aspects.

In our work, we employed Micronaut, Quarkus, and Spring, three widely used open source application frameworks for developing web services. Using each of these frameworks, we developed web applications that expose the business operations defined in the benchmark specification. In this architecture, benchmark workloads are routed to the web applications instead of being executed directly on the database engine (PostgreSQL).

The effects of compilation techniques and system settings have been evaluated through separate deployments that either employed the standard *Just-In-Time* (JIT) compilation features of an Eclipse Temurin JVM, or were built with GraalVM Native Image based compilers.

Outcomes Four main contributions emerge from this empirical investigation. First, we provide a comprehensive understanding of the factors that influence performance and resource usage in cloud-native Java services by investigating three software-engineering dimensions: the choice of an application framework, the selection of compilation techniques, and the tuning of configuration parameters. Second, we deliver empirical insight into the trade-offs between JIT and AOT compilation strategies in realistic application scenarios, showing that AOT compilation produces binaries with low memory footprint, whereas applications running on a standard *Java Runtime Environment* (JRE) with JIT features attain higher sustained throughput and, unexpectedly, consume less energy under continuous load. Third, we demonstrate that the combined optimizations of framework, compilation, and configuration choices yield a greater efficiency gain than any factor considered in isolation. Finally, we release the experimental suite and all measured data as open-access resources, thereby fostering reproducibility and enabling the research community to extend this work in the future [10].

1.2.3 Third Contribution

This contribution aims to identify interrelationships among software indicators, such as performance, energy consumption, and binary size, which could be used to simplify the evaluation of configurable software systems and guide optimization decisions.

Motivation Software configuration lies at the heart of modern systems: a single system can be tuned through countless combinations, such as compiler flags, library versions, and runtime parameters. Exhaustively exploring this configuration space to locate a set of settings that best satisfy a given set of requirements (e.g., execution time, memory footprint, energy consumption) is neither realistic nor sustainable as this could rapidly exceed the time and computational resources available. Furthermore, a configuration that is optimal for a given workload and hardware platform can quickly become sub-optimal when requirements evolve, workload characteristics shift, or the underlying infrastructure is upgraded.

Even evaluating a single configuration is expensive, particularly when multiple software indicators must be measured and analyzed. Consequently, identifying relationships between metrics can substantially simplify the evaluation process, reducing the need for exhaustive runtime experimentation.

Method This contribution uses the open-source JHipster [54] development platform, aiming at facilitating the generation of full-stack web services. Using methods and tools from *Software Product Line* (SPL) engineering (*i.e.*, techniques for managing a set of related software), we created a variability model of JHipster's configuration options to automate the generation of complete software stacks. From this model, 118 distinct software stacks were identified, each of them were then built and assessed using the default load test provided by JHipster through the Gatling load testing tool. During this evaluation process, we collected a range of runtime indicators, including stack boot time, response time of benchmark requests, and energy consumption.

Outcomes The present work makes four principal contributions. First, we demonstrate that JHipster constitutes a compelling case study for investigating software-component variability, offering a rich set of configuration options and a tool chain that turns configuration files into runnable applications with minimal effort. Second, by evaluating the impact of JHipster's configuration parameters on a range of software indicators, we provide a detailed picture of how each option influences the measured metrics, showing that a configuration that has only a marginal effect on one indicator may significantly affect another. Third, we identify correlations of varying magnitude among the measured metrics—strongest between complementary indicators such as processor and memory energy consumption, weaker between metrics of different nature—and we further demonstrate that, by combining heterogeneous indicators, it is possible to estimate metrics that are otherwise difficult to measure. Finally, to support future research on configurable software systems, we release a complete replication package that includes the full dataset of measurements [41].

1.3 Publications

In this section, we present the research publications and future submissions resulting from the work discussed in this thesis.

1.3.1 Published

The following studies have been published in international conferences:

1. A. Bonvoisin, C. Quinton, R. Rouvoy, **Understanding the Performance-Energy Tradeoffs of Object-Relational Mapping Frameworks**. In proceedings of the 31st IEEE *International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2024.

2. E. Guégain, A. Bonvoisin, M. Acher, C. Quinton, R. Rouvoy, **Exploring Performance of Configurable Software Systems: the JHipster Case Study**. In proceedings of the 29th ACM *International Conference on Evaluation and Assessment in Software Engineering (EASE)*, 2025.

1.3.2 Future Submission

The following work is still ongoing and aims to be submitted to a journal dedicated to software engineering and empirical analysis:

1. A. Bonvoisin, V. Amand, C. Quinton, R. Rouvoy, **Optimizing Performance and Resource Efficiency in Java-based Cloud Native Services**.

1.4 Outline of This Document

In this section, we describe the structure of this dissertation, organized into the seven chapters presented below.

Chapter 1 In this chapter, we introduce the research context, as well as the problem and challenges that motivated the work discussed in this dissertation. We then detail our research goals and contributions, followed by a presentation of our published work and future submissions. Finally, we conclude with this outline of the document's structure.

Chapter 2 In this chapter, we introduce the conceptual and methodological foundations of this thesis. First, we introduce approaches and techniques used to measure the energy consumption associated with running software applications. Second, we discuss research on software engineering practices including studies that investigates how design patterns and refactoring affect energy consumption.

Chapter 3 In this chapter, we provide a literature review on work related to software system application layers, with a particular focus on frameworks, runtime environments, and configuration management, which are central aspects of this thesis. By discussing and comparing state-of-the-art studies, then we position our contributions within these research areas.

Chapter 4 In this chapter, we explore the trade-offs between performance and energy consumption that arise from using ORM frameworks. We first present different programming approaches for implementing data access, illustrating both the basic method without using an ORM and the solutions provided by the programming interfaces offered by ORM compatible with the JPA specification. Then, we detail our research methodology by describing the experimental environment, the metrics we measure, and how they are collected. Next, we present our analysis of the data collected and discuss our findings. Finally, we conclude ORM selection and configuration to balance performance and energy consumption.

Chapter 5 In this chapter, we investigate how application frameworks, configuration choices, and compilation strategies affect both performance and resource efficiency. First, we provide some background knowledge on application frameworks and compilation tools involved in the development of Java applications. Second, we detail our experimental methodology. Next, we present the results of our experiments to answers our research questions and we discusses both the implications of our findings and the validity concerns. Finally, <e conclude by outlining the factors that influence performance and resource utilization in modern Java web services.

Chapter 6 In this chapter, we examine the interrelations among performance indicators and how they are influenced by the different options available in a configurable software system. We begin by introducing JHipster as our case study, outlining the performance metrics we measured and describing the experimental setup. We then investigate the correlation among the collected indicators and assess the impact of individual configuration options on system performance. Finally, we discuss the implications of our findings and the relevance of JHipster for research on configurable systems.

Chapter 7 In the final chapter, we conclude this thesis by summarizing the key findings of our research, and proposing short-term research directions and long-term perspectives.

Chapter 2

Background and Concepts

In this chapter, we outline the contextual background of this thesis by providing an overview of the key concepts that underpin this work.

2.1 Measuring Software Energy Consumption

Evaluating a software system is not a one-off activity. First, we must identify the metrics that are relevant to the evaluation such as binary size, memory usage, or energy consumption. Next, we need to define precise, repeatable methods for measuring those metrics. Finally, the software component should be tested in an environment that faithfully reproduces its intended execution context, ensuring that the results are realistic and comparable.

In this section, we provide a focused overview of the essential concepts related to measuring or estimating the energy consumption associated with running software. The solutions presented in this section are based on knowledge acquired during the thesis, therefore this section does not aim to provide an exhaustive inventory of all existing tools or approaches, but rather to give an overview of the techniques and levels of granularity they provide.

While the traditional measurement approach for collecting energy consumption of hardware components relies on physical wattmeters [30, 27], in practice this method is costly and lacks the precision to determine the energy consumption of specific software operations. Nevertheless, measuring the energy consumption of individual instructions, libraries, or entire applications remains a complex task that researchers aim to address by developing new measurement approaches and tools.

Among these approaches, hardware-integrated sensors or on-chip power models constitute the foundation for measuring software energy consumption. A notable example of such solutions is the *Running Average Power Limit* (RAPL) interface presented by David *et al.* [26]

as a power management technology that can be used to both limit and estimate the energy consumption of processors that provide built-in support. Since its integration into Intel micro-architectures beginning with the Sandy Bridge architecture (2011), RAPL has become one of the most studied on-chip power measurement solutions. For instance, Khan *et al.*[61] explored its strengths and weaknesses, while Desrochers *et al.* [28] discussed its accuracy in measuring *Dynamic Random Access Memory* (DRAM) power. Both studies support the consensus within the research community that RAPL provides accurate measurements of processor-related element.

Although such approaches provide reliable indications of the host's energy consumption, they cannot attribute that consumption to a particular process in practical contexts. Consequently, recent efforts have been dedicated to developing software-defined power meters that aim to distribute hardware energy consumption among several running processes.

WattWatcher [66] proposed by LeBeane *et al.* and SmartWatts [36] by Fieni *et al.* are two such initiatives. They estimate software energy consumption using energy models, generally built using learning techniques and hardware performance counters—which are processors registers dedicated to counting various hardware activities related to software execution. While their accuracy needs to be perfected, they are a promising solutions for providing developers application-level view of energy usage that can guide optimization and energy-aware design decisions.

Finally, initiatives have also been developed to estimate energy consumption at the level of programming constructs or code snippets. For example, the RJoules [16] library provides fine-grained energy measurements for R scripts, while the jRAPL [69] offers a similar capability Java applications. However, these tools require the insertion of specific function calls around the code snippets to be monitored, which makes them less suitable for large-scale energy measurement. As a notable alternative to jRAPL for automated energy evaluation of Java applications, JoularJX [83] does not require source-level instrumentation, as it operates as an agent that can be plugged into a JVM to measure the power usage of each executed Java method.

To conclude this section, the discussed methods and tools are summarized in Table 2.1.

2.2 Efficient Software Design

As methods for measuring or estimating software energy use become more accurate and easier to apply, researchers have begun to study the efficiency of software building blocks and how software engineering practice influences their energy consumption.

Table 2.1 Synthesis of approaches and solutions for measuring or estimating software energy consumption.

Granularity	**Approaches and tools**	
Computer and physical infrastructure	Physical power meter discussed in [30]	
Chip power domains	RAPL [26]	
Software (process-level energy estimations)	WattWatcher [66] SmartWatts [36] PowerJoular [83]	
Software (code snippets energy estimations)	Java Platform	jRAPL [69] JoularJX [83][1]
	R Platform	RJoules [16]

[1] JoularJX relies on PowerJoular to estimate the energy consumption of executed Java methods.

In particular, several studies have investigated how to balance performance and resource utilization within programming languages and libraries.

2.2.1 Programming Languages

Different languages use different mechanisms for handling operations, which can influence how much energy the software uses. For example, some languages might be interpreted at runtime, while others are directly compiled into machine-specific instructions. This difference could lead to varying levels of energy consumption. Therefore, the relationship between execution time, memory usage, energy consumption and programming languages has been explored in several studies.

Pereira *et al.* [98] analyzed 27 programming languages and found that compiled languages are generally faster and more energy-efficient than interpreted ones.

Georgiou *et al.* [38] provided recommendations for developers regarding the most energy-efficient languages for specific types of tasks. For instance, *Go* appears to be particularly well-suited for sorting operations, while *Rust* is more efficient for I/O-intensive workloads.

Kumar *et al.* [64] conducted a comprehensive evaluation of Java programming constructs, including data types, operators, and control statements, to understand their energy efficiency implications.

2.2.2 Libraries

Libraries can be optimized for speed at the expense of increased memory usage, or be tuned for low memory usage with a potential trade-off in speed. In this section, we discuss studies that examine these trade-offs.

Shanbhag *et al.* [116] analyzed the energy consumption of various data-frame processing libraries, also explored by Nahrstedt *et al.* [80]. Their study compared the energy usage and performance of two popular *Python* data analysis libraries, *Pandas* and *Polars*, highlighting differences in computational efficiency and resource consumption.

Pinto *et al.* [100] examined the thread-safe *Java* collections from the standard library and found significant variations in the energy footprint of analogous operations among various implementations of the same Java collection. Building upon this work, Oliveira *et al.* proposed a tool [87] designed to assist developers in selecting energy-efficient collection implementations to build more sustainable software. Other research efforts delved into the energetic profiles of *Java* Collections and I/O libraries [46, 97], highlighting the need to select the proper implementation.

By classifying the energy consumption of 22 file read and write methods in Java native I/O classes, Rocha *et al.* [109] shown that small changes may improve the energy consumption by 2. Ournani *et al.* [93] goes beyond this work by assessing additional standard and third-party I/O libraries.

2.2.3 Code Quality, Design Patterns and Refactoring

The quality of the source code, the use of design patterns, and the way developers refactor their source code can all influence how much power a system draws. In this section we therefore present research that explore the energy implications of everyday software-engineering practices.

Ournani *et al.* [94, 95] assessed that structural design pattern had no negative impact on the energy consumption of Java-based software.

Noureddine *et al.* [85] studied the energy consumption of Observer and Decorator pattern implementations, while Connolly Bree *et al.* [22] investigated the Visitor one and they have shown that implementation details can have a notable impact on software efficiency.

Poy *et al.* [102] conducted a systematic study on the energy impact of design patterns, code smells, and refactoring techniques, showing how structural choices in software architecture influence energy consumption. Similarly, Rani *et al.* [106] explored the applicability of mobile energy patterns to web applications, analyzing how specific energy-efficient de-

sign practices—such as dynamic retry delay and content lazy loading—compare to their alternatives, fixed delay retry and eager loading, respectively.

Finally, Danglot *et al.* [25] investigated energy consumption in continuous integration systems, highlighting how developers' test executions can be leveraged to detect energy regressions. Their study emphasizes the importance of energy-aware testing, shifting the focus beyond traditional performance metrics.

Chapter 3

State of the Art

In this chapter, we review and discuss the body of literature concerning software frameworks, runtime environment and application configuration, outlining the limitations in designing efficient configurable software services and positioning this dissertation at the intersection of these research areas.

3.1 Software Frameworks Research Landscape

For more than three decades, software frameworks [55, 35, 115] have been used to accelerate the development and deployment of complex systems. They provide a structured foundation that enforces established architectural patterns, fosters code reuse, and encourages modular design, which aims to improve software quality, enhance maintainability, and reduce development effort. To design such frameworks [113, 55], recurring solutions to similar problems are usually generalized through abstractions that application developers leverage to build their systems. Yet, a framework engineered to satisfy a broad set of use cases may impose constraints that do not align with the specific needs of a given application, leading to reduced flexibility, bugs, and performance overhead [123, 19, 20, 103].

In this section we review research on software frameworks, with a specific focus on ORM and web application frameworks, given their widespread use in enterprise applications [134] and in popular open-source projects [120] (e.g., Broadleaf Commerce [12], Discourse [21], and Mastodon [34]).

3.1.1 Object-Relational Mapping Frameworks

In this section, we survey and discuss studies on *Object-Relational Mapping* frameworks to present an overview of the body of knowledge and prevailing research trends in the field.

Torres *et al.* [123] surveyed popular ORM solutions selected from 10 development platforms. Using documentation review, practitioner experience, and literature analysis, the authors mapped each ORM solution to established architectural and structural patterns. The study therefore reports that, despite the inherent differences between object-oriented languages and relational databases, the ORM solutions share many similarities, while their differences arise mainly from varying flexibility and feature sets. As a contribution, the paper proposes a set of pattern -based criteria that can be used to evaluate and compare ORM frameworks, thereby helping developers assess and migrate between ORM implementations.

Bailis *et al.* [3] investigated ORM framework mechanisms for maintaining database integrity at the application level. Their work primarily focused on the ORM features of the Ruby on Rails framework (*i.e.*, Active Record) by reviewing a set of 67 open-source applications built on that framework. To a lesser extent, they also examined how data-model constraints and validation are implemented in other frameworks such as Hibernate and Waterline. They found that enforcing data integrity at the application level via framework constructs is a common practice, but they also reported that this approach could lead to inconsistencies (e.g., uniqueness violations) under concurrent workloads, since validation and association rules may not be correctly enforced at the database level.

Procaccianti *et al.* [103] examined the energy efficiency of data-access strategies in PHP applications, comparing hand-written SQL queries, the Propel ORM framework, and TinyQueries—a library that compiles SQL queries from query definitions in the *JavaScript Object Notation* (JSON) format. They measured the execution time and power draw of each approach under identical workloads, revealing that the Propel framework increased both execution time and overall energy consumption. In contrast, the hand-crafted query strategy showed the best energy efficiency, while TinyQueries delivered intermediate energy efficiency.

Verdecchia *et al.* [128] conducted a study on the performance and energy implications of refactoring code smells in Java ORM-based applications. The research revealed that addressing code smells through refactoring not only enhanced energy efficiency but also potentially improved the performance of the system. Surprisingly, when all code smells were refactored, they observed a performance overhead of approximately 6.8%, while the energy consumption was reduced by 10.7%. This finding underscores the complex relationship between performance and energy consumption, highlighting that these two aspects do not always correlate directly with each other.

Chen *et al.* investigated various performance issues related to software developed using ORM frameworks. Specifically, they conducted two distinct studies on the detection of performance anti-patterns related to JPA [17] and the performance impact of redundant

data access [20]. In the first study, they review code patterns that can lead to performance issues and propose a framework to assist developers in automatically flagging performance anti-patterns in the source code of their ORM framework. In the second work, they focus on localizing and evaluating the performance penalties of redundant-data problems—situations in which a data-access request retrieves or modifies more data than is actually required. As a result, they reported that resolving such problems improved the response time of the studied systems by an average of 17%. Alongside these studies, this research group also provided an industrial experience report [19] that discusses the practical challenges they encountered in identifying data-access bugs in large-scale systems and details five bugs observed while using JPA frameworks, providing developers with concrete examples and possible solutions to avoid similar pitfalls.

In parallel, they also proposed CacheOptimizer [18], a tool aimed at optimizing the configuration of cache frameworks integrated with Hibernate. In their evaluation, they measured throughput gains and examined the memory cost of the configurations proposed by the tool against no cache, default cache, and an all-cache-enabled configuration, and reported that the optimized settings improved application throughput by at least 27%, maintained memory usage within acceptable limits, and in some cases even reduced memory consumption.

3.1.2 Web Application Frameworks

The literature specifically addressing the performance and energy efficiency of web application frameworks remains limited [13, 73]. Most studies focus on the building blocks of these frameworks (programming languages, support libraries, and ORMs as previously stated) instead of web frameworks as integrated systems.

Calero *et al.* [13] investigated differences in execution time and energy consumption between applications developed with and without the Spring framework. Their findings revealed that extensive usage of Java reflection in the Spring framework significantly increases energy consumption due to runtime overhead.

Meglio *et al.* [73] studied the performance and energy efficiency of various frameworks running on different runtime environments. Their evaluation found that JavaScript implementations powered by Nodejs or Bun outperformed both Java and Python-based configurations in terms of throughput and response times. Conversely, Python deployments created with the Django framework and running on CPython or PyPy showed the best resource usage. These results highlight the trade-offs that arise at the framework and runtime levels, highlighting the need for further empirical analysis examining how web framework and runtime selection affects performance and energy consumption.

3.2 Runtime Environment

The runtime environment, which includes components such as a language interpreter or virtual machine, a JIT compiler, a garbage collector, and libraries, plays a decisive role in how programs run and how much energy they use.

For instance, in their empirical analyses of application framework performance, Meglio *et al.* [73] also reported that the polyglot runtime GraalVM delivered mixed outcomes, matching the performance of OpenJDK for a Java-based stack (Spring Boot and Micronaut), but falling behind Node.js and Bun when running JavaScript stacks (Express and Nest).

Stoico *et al.* [118] demonstrated that compiling Python—either *Ahead-Of-Time* to produce binaries or *Just-In-Time* alongside interpretation—can enhance energy efficiency compared to the CPython reference implementation that runs only in interpreted mode.

Finally, Ournani *et al.* [92] evaluated the impact of JVM on energy consumption by comparing 52 JVM distributions and reported that, although the implementations exhibit comparable energy usage, the distributions can be clustered into three categories based on JVM architecture.

Taken together, these findings underscore the importance of considering the runtime environment when assessing both performance and energy consumption.

3.3 Configuration and Customization

Software systems are generally designed to be configurable, offering users extensive customization possibilities to meet diverse requirements across different environments and use cases. However, the extensive range of configuration options makes it challenging to evaluate every possible setup and identify those that optimize functional requirements, performance, and resource efficiency.

Many efforts have been put into managing the performance of configurable software. A survey by Pereira *et al.* reports on 69 publications in that field [96], and shows that research has mainly addressed one of the following three areas: *(i)* performance prediction, intending to estimate the performance of a configuration without actually measuring it, *(ii)* performance optimization, to generate optimal configurations of a system, and *(iii)* recommendation systems, to assist developers during the feature selection process.

Numerous studies have proposed deterministic methods for identifying near-optimal configurations in relation to a set of performance indicators [39, 138, 86, 81]. But other studies also employ other approaches and techniques. For instance, Hierons *et al.* [48] use genetic algorithms to minimize the number of measurements needed for configuration

optimization, while Siegmund *et al.* [117] rely on performance prediction methods to identify optimal settings.

3.4 Chapter Conclusion

Table 3.1 Synthesis of ORM frameworks studies.

Ref.	Contribution Type	Aspects considered	Framework	Non-ORM solutions considered
[123]	Survey	Design patterns	Hibernate Cayenne ODBend SQLAlchemy Entity Framework Doctrine Bookshelf.js Active Record	MyBatis library[1]
[103]	Empirical Analysis	Energy Performance	Propel	Hand-written queries TinyQueries library
[128]	Empirical Analysis	Energy Performance	Hibernate	✗
[17]	Empirical Analysis	Performance	JPA implementations	✗
[19]	Empirical Analysis	Performance Correctness	JPA implementations	✗
[20]	Empirical Analysis	Performance	Hibernate	✗
[18]	Tool	Configuration Memory Performance	Hibernate	✗
[3]	Empirical Analysis	Correctness	Active Record	✗
[127]	Tool	Performance	Sequelize	✗

[1] MyBatis should not be considered an ORM framework because it requires explicit SQL query definitions and data structure mappings.

As shown in Table 3.1 several studies have investigated the performance of ORM frameworks in practical contexts however fewer have investigated the energy consumption and the trade-offs associated with these solutions. While Procaccianti *et al.* [103] investigated both performance and energy consumption of various data access strategies, they did not

Table 3.2 Summary of web application framework related works.

Ref.	Contribution Type	Aspects considered	Frameworks	Runtime Environment
[13]	Empirical Analysis	Performance Energy	Spring (Java)	Not reported
[73]	Empirical Analysis	Performance Energy	Spring (Java) Micronaut (Java) Express (JavaScript) Nest (JavaScript) Django (Python)	Hotspot JRE GraalVM JRE Node.js Bun CPython PyPy

address JPA implementations, but rather compared raw SQL queries with Propel (a PHP ORM) and Tiny Queries. Furthermore, they did not investigate the potential improvement brought by changing ORM configurations contrarily to the contribution discussed in the next chapter. The second study that investigated both performance and energy consumption [128] considered Hibernate and certain configuration parameters, but they did not compare the different *Application Programming Interfaces* (APIs) offered by the JPA specification or compare Hibernate with EclipseLink.

Then, as shown in Table 3.2, two studies have investigated the energy consumption and performance of web application frameworks within the Java platform. These studies have laid the foundation for understanding the performance and resource efficiency of Java frameworks and their configurations. However, they do not address the role of AOT compilation and the influence of deployment configurations on performance and energy consumption of software based on web application frameworks.

The identified research gaps therefore motivate the three contributions presented in the next chapters.

Chapter 4

Understanding the Performance-Energy Trade-offs of ORM Frameworks

In the previous chapters, we highlighted that the energy consumption of software systems has gained prominence as a significant environmental and societal concern. Multiple studies have underscored the substantial influence of software on energy consumption and emphasized the importance of green software design to improve the energy efficiency of software systems. For instance, previous works in this field have studied and compared the energy consumed by common operations on Java collections [46] or the use of different Java I/O libraries [93]. However, while various tools and methods have been developed to comprehend, measure, and predict software energy consumption [108], the complexity of software environments and the composition of software layers make this sustainability objective particularly challenging [93].

To advance this line of research it is essential to investigate a wide set of the components and development practices that are pervasive in software systems. In this context, the mechanisms that enable applications to access and manipulate persistent data represent a broad class of software building blocks whose impact on energy consumption has not yet been thoroughly examined. While data access operations such as retrieving and updating records in a database constitute a core functionality of many software systems, the energy implications of the libraries and programming patterns used to perform these operations remain insufficiently understood. Among these, *Object-Relational Mapping* (ORM) frameworks are a typical example of widely adopted method for bridging relational databases with object-oriented programming languages [134]. Although these frameworks simplify data access and accelerate development, it is unclear whether they provide energy-efficient solutions for querying and persisting data at scale.

To explore this issue, this chapter presents an empirical investigation that leverages the extensive set of Java data-access solutions. By contrasting a low-level approach—direct database interaction through drivers that implement the *Java DataBase Connectivity* (JDBC) specification—with higher-level ORM frameworks that sit on top of JDBC, we evaluate the balance between performance and energy consumption across the two main categories of data-access approaches employed in Java-based applications.

In particular, in this chapter we aim to address the following research questions:

RQ1: *How much energy overhead is introduced by ORM strategies compared to plain JDBC?*

RQ2: *How does ORM framework performance relate to energy efficiency?*

RQ3: *Does the configuration of an ORM framework influence its energy consumption?*

To answer these questions, we leverage the *Transaction Processing Performance Council benchmark C* (TPC-C), a widely accepted standard for assessing the performance of relational database. The benchmark is executed against two of the most popular and state-of-the-art ORM frameworks[19, 60]—ECLIPSELINK [33] and HIBERNATE [47]. To evaluate the systems under a variety of realistic scenarios, the benchmark is executed with several workload profiles, each representing different database transaction mixes and concurrency levels. Finally, we quantify their energy footprints and identify tuning levers that can help developers balance throughput and energy efficiency when building data-intensive Java applications.

The remainder of this chapter is organized as follows. Section 4.1 introduces the data access strategies commonly implemented by Java applications. Section 4.2 details the experimental protocol we adopted to measure the energy efficiency of ORM frameworks. Section 4.3 presents the experimental results that directly address the research questions outlined above. Section 4.4 discusses the findings and addresses concerns regarding the validity of our work. Finally, Section 4.5 concludes this chapter.

4.1 Data Access Strategies

In this section, we introduce 5 data access strategies that we have implemented to assess the trade-offs associated with using an ORM framework. The first strategy illustrates how data access can be handled without any ORM abstraction, while the remaining 4 strategies are built on *Jakarta Persistence API* (version 3.1) [51] features—a Java specification that eases relational database interaction by simplifying the mapping between Java objects and

relational records. To illustrate how each strategy handles requests, we provide for each one a code sample that processes the same request; *i.e.*, *select the email for a given customer*.

4.1.1 Plain JDBC Access Strategy

When they are not using an ORM, developers directly leverage a JDBC driver to interact with the *DataBase Management System* (DBMS). As illustrated in Listing 4.1, they explicitly manage database connections, craft *Structured Query Language* (SQL) queries, and handle query results. This approach provides fine-grained control over the DBMS but demands specific skills for developers, including writing and maintaining custom SQL queries and dealing with low-level database intricacies. In the remainder of this chapter, we will refer to this implementation as the plain JDBC one.

```
String query = "SELECT email FROM customer_table "
               + "WHERE unique_id = ?";
ResultSet rs = jdbcCon.prepareStatement(query)
                        .setInt(1, id).executeQuery();
String email = rs.next()?rs.getString("email"):"";
```

Listing 4.1 Plain JDBC access strategy.

4.1.2 ORM Access Strategies

In this subsection we first provide an overview of *Jakarta Persistence API* (JPA), highlighting its core concepts (e.g., data-model mapping, caching layers) and then we illustrate 4 distinct data access strategies that can be considered when adopting a JPA-compatible ORM framework to interact with a relational DBMS. Each of these data access strategies comes with its benefits and trade-offs, and the choice of a strategy depends on various factors, such as performance requirements, compatibility with DBMS features, and code maintainability. By assessing these different ORM-based strategies, we aim to gain better insights into the influence of such strategies and the trade-off they offer between performance and energy efficiency.

Jakarta Persistence API Core Concepts

Within JPA, entities are Java classes representing persistent data structures stored in relational databases. These entities are typically annotated to define how they map to corresponding database tables, outlining the relationships between Java objects and database columns (cf. Listing 4.2). Each instance of an entity corresponds to a record or row in the associated

database table. Furthermore, entities can be annotated to specify additional metadata, including the definition of plain-text queries associated with unique names, later used to identify a query to be executed.

```
@SqlResultSetMapping(
    name = "Customer.getEmailSQLMapping",
    columns = @ColumnResult(
                name = "email", type = String.class))
@NamedNativeQuery(name = "Customer.getEmailSQL",
    query = "SELECT email FROM customer_table "
           + "WHERE unique_id = ?",
    resultSetMapping = "Customer.getEmailSQLMapping")
@NamedQuery(name = "Customer.getEmailJPQL",
    query = "SELECT c.email FROM Customer c "
           + "WHERE c.id = ?1")
@Entity @Table(name = "customer_table")
public class Customer implements Serializable {
    @Column(name = "unique_id")
    @Id private Integer id; // Primary key
    @Column(name = "email")
    private String email;
}
```

Listing 4.2 Entity class declarations.

Beyond mapping, JPA also introduces two caching features. The first-level cache *a.k.a.* "Persistence Context" is bound to a database transaction and cannot be disabled, as its goal is also to monitor changes on "managed" entities. When an entity is retrieved from the database, it is stored in this cache, ensuring data consistency by automatically updating changes to the object and reducing the number of database queries during the transaction.

As for the second cache level, it is optional. If used, it extends the persistence context and developers must take care of the data cached to prevent data staleness. In this work, we disabled this second-level cache—which requires an external caching library—to solely assess ORM capabilities without introducing any bias related to the choice of a particular caching implementation.

Finally, a key benefit of conforming to JPA features is that one can seamlessly transition between various ORM implementations without modifying the application source code. This migration process merely involves substituting the vendor-specific package in the project dependencies. By adopting this approach in this work we ensured a fair comparison among providers (EclipseLink and Hibernate).

Native Access Strategy

This approach refers to the use of raw SQL queries using the JPA programming interfaces (cf. Listing 4.3). As the specification's terminology refers to raw SQL queries as "native queries", we named this strategy as the "native" one. In contrast to the plain JDBC strategy, which imposes manual data mapping, the JPA standard encourages that the mapping between query result sets and Java objects are declared through the framework configuration (cf. Java annotations in Listing 4.2).

```
EntityManager em = // ...
String email = (String) em.createNamedQuery("Customer.getEmailSQL")
               .setParameter(1,id).getSingleResult();
```

Listing 4.3 Native access strategy.

JPQL API Access Strategy

This data access strategy uses the *Jakarta Persistence Query Language* (JPQL), a query language with an SQL-like syntax. This language provides an abstraction layer over raw SQL queries that enables developers to define queries independently of the SQL dialect used by the targeted data store. In JPQL, developers manipulate Java fields and class names, which are mapped to database tables—instead of explicitly specifying column and table names within the queries (cf. Listing 4.2). This approach simplifies the query creation process and enhances portability, as the queries can be adapted to different databases without any modification (cf. Listing 4.4).

```
EntityManager em = // ...
String email = em.createNamedQuery(
        "Customer.getEmailJPQL", String.class
      ).setParameter(1,id).getSingleResult();
```

Listing 4.4 JPQL access strategy.

Criteria API Strategy

This strategy relies on the "Criteria" programming interface of JPA, which enables developers to construct queries using a set of Java objects and methods calls, rather than composing queries as plain text like raw SQL or JPQL (cf. Listing 4.5).

```
EntityManager em = // ...
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<String> query =
                          cb.createQuery(String.class);
Root<Customer> root = query.from(Customer.class);
ParameterExpression<Integer> idParameter =
                            cb.parameter(Integer.class);
Predicate predicate = cb.equal(idParameter,
                                 root.get(Customer_.id));
String email = em.createQuery(query
                    .select(root.get(Customer_.email))
                    .where(predicate)
                  ).setParameter(idParameter,idValue)
                  .getSingleResult();
```

Listing 4.5 Criteria access strategy.

Managed Entity Strategy

As introduced previously, managed entities are objects mapped to database records, which are actively monitored by the ORM framework to ensure that any changes are automatically reflected in the underlying database (cf. Listing 4.6).

```
EntityManager em = // ...
Customer c = em.find(Customer.class,id);
String email = c.getEmail();
```

Listing 4.6 Entity access strategy.

4.2 Experimental Methodology

This section outlines the benchmark and workload parameters, the software systems evaluated, the experimental setup, the metrics collected, and the data-collection process.

4.2.1 Benchmark Description and Workload Parameters

To assess the energy efficiency of strategies described in Section 4.1, we opted for the *Transaction Processing Performance Council benchmark C* (TPC-C) [125, 124] defined in the BenchBase (formerly OLTPBench) framework, an open-source database benchmarking tool [29, 8]. While this well-known benchmark was initially designed to assess database capabilities through an intensive workload simulating business activities between customers

and a wholesale supplier, many aspects of its specification are well-suited for evaluating data access solutions.

Database Schema

The schema consists of 9 tables representing various entities involved in wholesale supplier operations. For instance, the Warehouse table stores unique identifiers and addresses of warehouses, while the Customer table holds details about individual customers, including identifiers, names, and number of payments.

Input Workload

The TPC-C specification [125] defines 5 distinct transaction types, each simulating a specific wholesale supplier activity and differing in complexity. A New-Order transaction inserts a new order into the system, while a Payment modifies accounting records to reflect a customer payment. Delivery simulates the actual transfer of goods by processing a batch of 10 undelivered orders. Order-Status returns the status of the most recent order placed by a customer, and a Stock-Level operation queries items whose inventory has dropped below a specified threshold, both executing a read-only transaction.

The queries defined within the transactions are pivotal to this study, as they provide a comparable basis to the queries generated by the different ORM strategies.

Workload Parameters

The BenchBase tool offers many configuration options for setting up different workload profiles. Because our work is based on software applications derived from that tool, those same options can be used to evaluate data access strategies under various scenarios. In this section we present 5 of those options.

The first two options are the *number of warehouses* and the *number of clients* to simulate, which are often adjusted together. The number of warehouses is a key parameter that serves as a scaling factor for adjusting the number of records to be inserted into the tables, the values used in database query parameters executed during the workload phase, and the number of concurrent users to be simulated.

This number of warehouse is a key parameter serves as a scaling factor, for adjusting the number of records to be inserted into the tables, the values used in database query parameters executed during the workload phase, and the number of concurrent users to be simulated.

In the default configuration of the BenchBase tool, this scaling factor is set such that it simulates a workload with too many clients in comparison to the capabilities of the servers

used. To align with the benchmark specification [125] recommending 10 clients per unit of the scaling factor, we set the number of warehouses to 5 and the number of clients that concurrently query the database to 50.

The third option is the *transaction isolation level*, which we have left unchanged from the default: SERIALIZABLE [101]. This guarantees that concurrent transactions do not interfere with each other and prevents inconsistencies.

The fourth parameter is the *transaction frequency distribution*. For each transaction type described in Section 4.2.1 we use the same proportions reported in the literature [125, 124], ensuring that the mix of workload operations mirrors established benchmark references.

Finally, the fifth configuration option is the *targeted transaction rate*, which enables specifying the number of transactions to be performed per second. We set the target rate from 10 to an "unlimited" number of *Transactions Per Second* (TPS), with increments of 10, 100, 1000, and 10.000 TPS. The special unlimited value instructs the system to generate as many transactions as possible.

4.2.2 Evaluated Software Systems

As introduced previously, we employed the TPC-C implementation from the BenchBase tool as the basis for implementing the data access strategies presented in Section 4.1. We have therefore developed a software variant of the tool for every data-access strategy.

Apart from modifying the data access layer of the original implementation, we also edited the source code related to a transaction retry feature. Rather than making additional attempts to execute a failed transaction again, our implementations categorize such attempts as failures from the outset. This adjustment has been implemented in each software variant (including for the plain JDBC alternative) to ensure that the measurements collected remain unaffected by potential variations introduced by retry attempts.

Since ORM strategies were implemented according to the most recent 3.1 version of JPA, the considered JPA provider at build time were EclipseLink (version 4.0.0), the reference implementation, and Hibernate (version 6.1.3.Final), the most popular alternative in the community.

4.2.3 Experimental Setup

To ensure reliable and consistent measurements, we conducted all our experiments on the consistent technical setup described in this section.

Hardware Settings

The hardware infrastructure comprises two identical Dell PowerEdge R640 servers, each equipped with a single Intel Xeon Gold 5220 @2.20GHz processor (18 cores/36 threads), 96GB of memory, 480GB *Solid-State Drive* (SSD), and two 25Gb/s network cards.

Software Settings

Both servers were configured with a minimal version of Ubuntu 20.04, including only essential components required for system operation and remote access through *Secure Shell* (SSH). On one server, we installed Git, XMLStarlet, and Docker to facilitate the build and execution of Java applications implementing the strategies discussed in Section 4.1.

These Java 17 implementations were compiled using a Maven container set up for reproducible builds. To ensure a lightweight runtime environment, we packaged these applications into Docker images that contain only a *Java Runtime Environment* (JRE) rather than a full *Java Development Kit* (JDK), using the official JRE-only Eclipse Temurin 17.0.8_7 Alpine Linux as a base image.

The other server was exclusively equipped with Docker to run an on-demand, volatile PostgreSQL 14.5 database mounted with a Docker volume cleaned before each measurement.

While Docker introduces an additional layer that might lead to increased energy consumption, it has a consistent and minimal impact on energy variation, resulting in a constant, negligible overhead [112].

4.2.4 Collected Metrics

In this subsection, we present the indicators we collected to assess the efficiency of data access layer strategies: power consumption, which reflects resource usage, and goodput, which highlights operational efficiency.

Power Consumption We measured power usage for both the processor *package* (PKG) and the *Dynamic Random Access Memory* (DRAM). These measurements were carried out using the *Running Average Power Limit* (RAPL) interface [26], a feature available with recent Intel processors known for its high accuracy and reliability in power monitoring [61, 28].

Goodput The goodput is a metric reported by the original BenchBase tool that is still reported by our implementations and indicates the efficiency of a database system by measuring the number of successful transactions completed per second. A higher goodput indicates

increased effectiveness, as it directly correlates with the system's ability to execute successful operations at a faster rate.

4.2.5 Data Collection Process

In this subsection, we describe our two-stage measurement protocol.

First Stage: Dataset Initialization Before measuring performance or energy consumption, we must guarantee that each measurement run starts from an identical database state. Reusing the same dataset is critical for fair comparisons because it removes data-related variance from the observed metrics. To that end, a PostgreSQL container is first started on the second server and bound to a Docker volume that points to a directory on the host machine. The database is then populated using the data-loader feature of the BenchBase tool. Since this data loader injects random values into each inserted record, we set the random seed setting to 0 to facilitate reproducibility.

The use of a Docker volume ensures that all data is persisted in that host directory, which is then considered an immutable backup of the initial dataset. For each new measurement run, this backup directory is duplicated and mounted as a volume for the transient PostgreSQL instance used in that run, so every temporary database instance in the evaluation phase receives exactly the same dataset. It is also important to note that, in our experimental setup, relying on a backup folder saves time and avoids unnecessary processing (e.g., network communication, query parsing) compared to employing the data loader before each measurement. Finally, this stage ends with the dismantling of the database.

Second Stage: Metrics Collection A single measurement run is a multi-step process conducted as follows. First, on the second server, we duplicate the dataset backup folder presented in the previous stage. Then, we measure the power consumption of both servers while they are at rest. These measurements are taken over 5 seconds, as we observed that if the CPU is not actively in use, power consumption quickly stabilizes within this period. Throughout this time frame, both systems remain in an idle state, with no active applications or processes running. Next, we start a database container on the second server and wait 30 seconds for it to be set up and ready to accept requests. The database being already populated by mounting a docker volume in a temporary folder copied from the backup, wee then initiate the execution of the assessed software variant, during which we gather data on the energy usage and the achieved goodput throughout the workload duration, set to 60 seconds in the BenchBase default configuration file. Finally, all running containers are stopped to put both

systems in an idle state, and we perform an additional 5-seconds measurement to assess the power consumption of both servers while back at rest.

4.3 Empirical Results

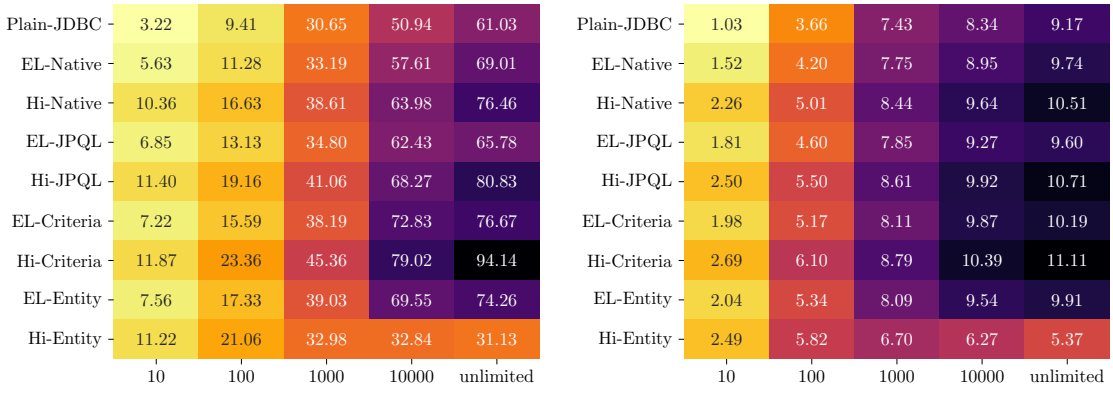
This section presents the results of our experiment. We consider the plain JDBC strategy as the baseline to evaluate the overhead induced by other strategies.

Given the stochastic nature of TPC-C based workloads, we repeated the measurement runs 40 times for each combination of rate goal (presented in Section 4.2.1) and data access strategy, then we calculated the median of the results, adding up to 4200 runs.

The reported power usage of an assessed strategy is determined by the system's power consumption during the workload, after subtracting the idle power consumption, as described in [39].

Lastly, although few extreme outliers are observed, we use the median as the central-tendency measure in most figures to reduce the influence of these anomalies on the displayed trend.

4.3.1 RQ1: How much energy overhead is introduced by ORM strategies compared to plain JDBC?



(a) CPU median power usage (Watts). (b) DRAM median power usage (Watts).

Fig. 4.1 CPU and DRAM median power usage (Watts) across configurations and a varying target rate.

To answer our first research question, we thus compared the energy consumption of the different ORM access strategies with the plain JDBC one when executing a certain workload,

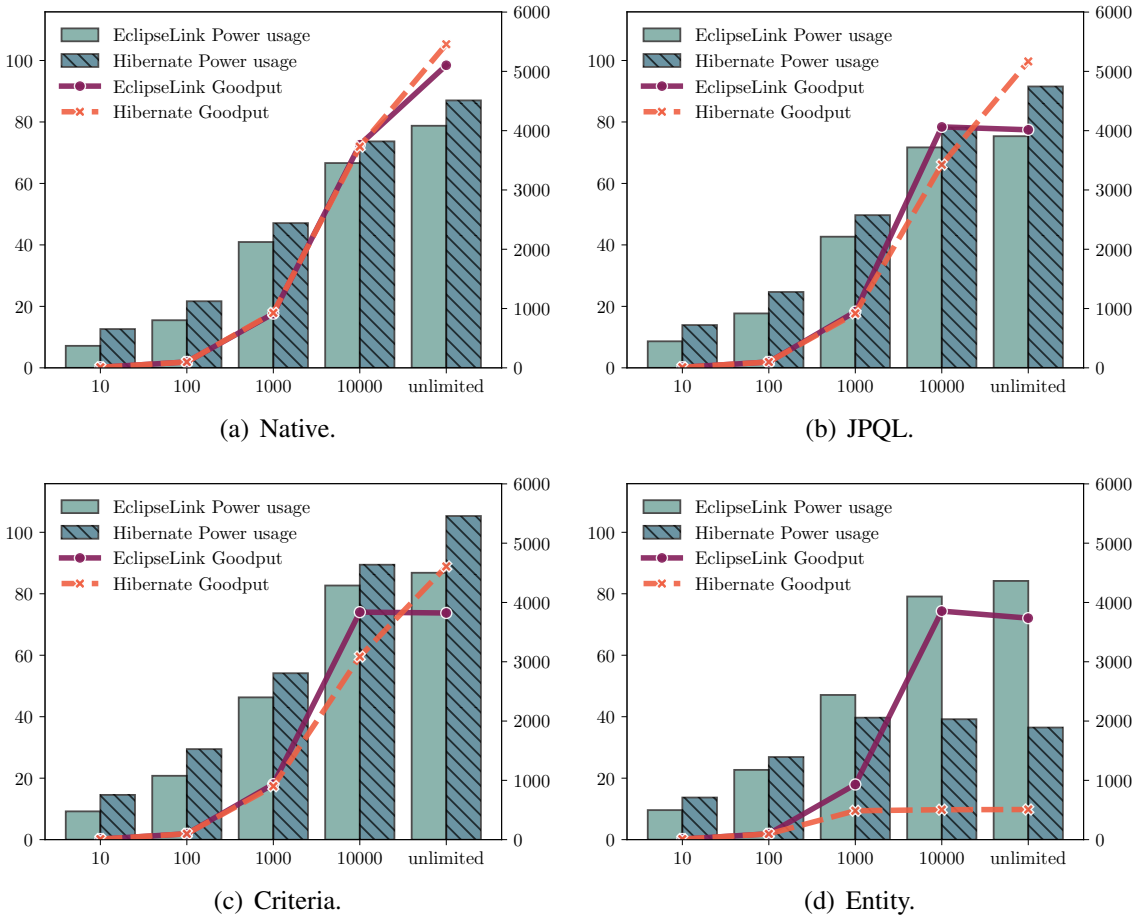
as explained in Section 4.2.5. Figures 4.1(a) and 4.1(b) depict, for each implementation, the median power usage depending on the target rate for *Central Processing Unit* (CPU) and DRAM, respectively (EL stands for EclipseLink and Hi for Hibernate). Both figures highlight consistent trends as the target rate increases, the power usage increases accordingly. Yet, one can observe that Hibernate Entity demonstrates low energy consumption at higher rates. This can be explained by its poor performance. As shown by Figure 4.2, the number of completed transactions Hibernate Entity can process is plateauing once the target rate reaches 1000 transactions per second. Hibernate Entity thus faces performance issues related to completed transactions, arising because of the increased concurrency resulting from the higher target rate. Consequently, Hibernate Entity transitions into a degraded mode, causing the built-in workload injector to reduce the number of initiated transactions which, in turn, reduces energy consumption.

Hibernate Entity left apart, we observe that the plain JDBC strategy exhibits the lowest energy consumption, while the use of ORM strategies results in a significant power usage increase, especially regarding CPU consumption. For instance, at the target rate 100, ORM strategies overhead varies from 16% to 59% for the EclipseLink native and Hibernate Criteria strategies, respectively. As ORM strategies introduce an additional abstraction layer compared to the plain JDBC one (see Section 4.1.2), such trends were predictable. For a comprehensive and fair analysis, the benchmark must execute an identical input workload. As some ORM strategies generate the executed SQL queries, we ensured that strategy implementations adhered closely to the business logic defined in the benchmark while complying with the concepts outlined by each ORM strategy.

Among all studied ORM strategies, only the Native ones guarantee the execution of SQL queries that match those defined in the plain JDBC strategy. When focusing on Native strategies, one can observe that, although the overhead is less significant, it is non-negligible. For instance, at the target rate of $10,000$, there is an 11% and 20% increase for the native EclipseLink and Hibernate ORM regarding CPU usage, respectively. While less important, the energy consumption increase regarding DRAM usage for the same target rate even so reaches 6% and 13% for the native EclipseLink and Hibernate ORM strategies, respectively.

Answer to RQ1: Our empirical study shows that the plain JDBC strategy consistently reports being the most energy-efficient compared to the ORM ones. The overhead introduced by ORM strategies is imposed by the additional abstraction layers offered by such frameworks. Among ORM strategies and apart from the specific case of Hibernate Entity which performs poorly at high rates, Criteria is the most consuming strategy for both Hibernate and EclipseLink frameworks.

4.3.2 RQ2: How does ORM framework performance relate to energy efficiency?



(a) Native. (b) JPQL.

(c) Criteria. (d) Entity.

Fig. 4.2 Goodput (Tx/sec, right-Y axis) and global power usage (Watts, left-Y axis) median variations across varying ORM strategies.

To answer our second research question, we compared the goodput of EclipseLink and Hibernate given their energy consumption when executing a varying workload. Figure 4.2 provides an overview of this comparison. Several observations can be made.

When employing the Native strategy, EclipseLink and Hibernate demonstrate similar performances as they both execute identical raw SQL queries, resulting in comparable sets of completed transactions. However, EclipseLink seems to offer the best energy efficiency, consuming 45% less energy at a rate of 10 and consistently showing an average reduction of 22% across all rates.

Regarding the other three strategies, EclipseLink always exhibits a recurrent performance pattern, performing consistently across varying rates and reaching equivalent or higher

goodput than Hibernate, until reaching 10,000 transactions per second. Once this rate is overtaken, EclipseLink goodput is then plateauing.

In contrast, the Hibernate goodput remains consistent across the Native, JPQL, and Criteria strategies, even at unlimited rates. Regarding these 3 strategies, EclipseLink is always better performing while consuming less energy than Hibernate at rates lower or equivalent to 10,000, as highlighted by Figures 4.2(b) and 4.2(c).

We also observe that Hibernate experiences a significant drop in performance when managing transactions relying on the Entity strategy (see Figure 4.2(d)). When further investigating these results, we found out that both frameworks handle queries differently. In particular, when fetching an entity that makes references to other ones, Hibernate does not check whether referenced entities are already present in the first-level cache described in Section 4.1.2. Instead, it directly uses join operations to fetch associated data from multiple tables. Listing 4.7 illustrates such operations, which instantiate all entities corresponding to transitive relationships—*i.e.*, the full graph of relationships. Consequently, Hibernate might redundantly retrieve data that is already stored in the persistence Context (the first-level cache presented in Section 4.1.2). EclipseLink behaves differently as it (i) sends queries only when referenced entities are not present in the first-level cache, and *(ii)* employs independent queries to fetch individual foreign entities rather than using joins, as shown by Listing 4.8, hence achieving better performances at scale.

```
-- Some columns have been omitted for clarity
SELECT w1_0.w_id, w1_0.w_tax, ...
FROM warehouse w1_0 WHERE w1_0.w_id=?

SELECT d1_0.d_id, d1_0.d_next_o_id, d1_0.d_tax, d1_0.d_w_id, w1_0.w_id, ...
FROM district d1_0
JOIN warehouse w1_0 ON w1_0.w_id=d1_0.d_w_id
WHERE (d1_0.d_id,d1_0.d_w_id) IN((?,?))
FOR NO KEY UPDATE
-- Above, referenced entities already fetched
SELECT c1_0.c_id, c1_0.c_d_id,c 1_0.c_w_id, c1_0.c_credit, c1_0.c_discount,
    c1_0.c_last, d1_0.d_id, d1_0.d_w_id, w1_0.w_id, ...
FROM customer c1_0
JOIN district d1_0 ON d1_0.d_id=c1_0.c_d_id AND d1_0.d_w_id=c1_0.c_w_id LEFT
    JOIN warehouse w1_0 ON w1_0.w_id=d1_0.d_w_id
WHERE (c1_0.c_id,c1_0.c_d_id,c1_0.c_w_id) IN((?,?,?))
```

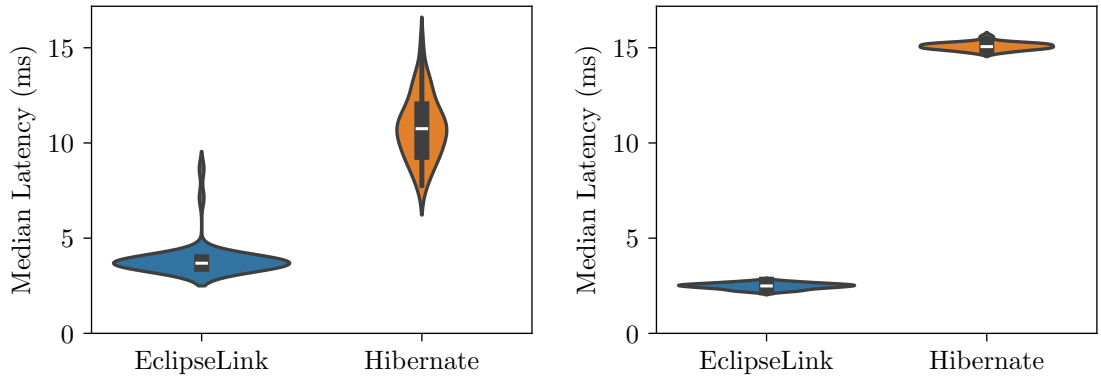
Listing 4.7 Transaction from HIBERNATE's managed entity strategy.

```
-- Some columns have been omitted for clarity
SELECT w_id, w_tax, ...
FROM warehouse WHERE (w_id = ?)

SELECT d_id, d_next_o_id, d_tax, ...
FROM district WHERE ((d_id = ?) AND (d_w_id = ?))
FOR UPDATE

SELECT c_id, c_discount, c_last, c_credit, ...
FROM customer WHERE (((c_id = ?) AND (c_w_id = ?))
AND (c_d_id = ?))
```

Listing 4.8 Transaction from EclipseLink's managed entity strategy



(a) Target rate 1,000 Tx/sec.

(b) Target rate 10,000 Tx/sec.

Fig. 4.3 Distribution of transactions median latency across measurements for the Entity strategy.

Using the JOIN clause for data retrieval from multiple tables can increase transaction complexity and disrupt concurrent operations in the DBMS. This increased workload leads to longer execution times, heightened latency (2.5 times more at rate 1,000 and up to 5 times more at rate 10,000, as depicted in Figure 4.3), and more data locks due to the SERIALIZABLE [101] isolation level, resulting in increased transaction failures.

Answer to RQ2: We analyzed the goodput of different strategies under varying workloads, in view of their power usage. Our empirical results indicate that EclipseLink consumes less energy than Hibernate to complete the same workload for rates up to 1,000. At higher rates, Hibernate exhibits a better goodput for all strategies except for the Entity one, which experiences a significant drop in performances. This is due to generated queries that play a critical role in understanding the performances of ORM frameworks. Customized SQL

queries tailored to specific tasks exhibit the best performance, but this approach does not benefit from the abstraction layers provided by ORM frameworks. Overall, EclipseLink stands out for generating well-performing queries.

4.3.3 RQ3: Does the configuration of an ORM framework influence its energy consumption?

To answer our third research question, we introduced modifications to the default ORM configuration used in previous experiments to assess whether these changes may improve energy efficiency. In particular, we focused on 3 different features of different scopes, namely *lazy fetching*, *batch processing*, and *partial update*. We chose to apply these changes to the Entity strategy for two main reasons. First, it offers the highest level of abstraction from a Java perspective. Second, we aimed to investigate whether these changes could address some of the Hibernate performance issues highlighted in the previous section. While the first feature is a JPA feature that can be enabled seamlessly on EclipseLink and Hibernate, the remaining two demand vendor-specific adjustments.

Lazy Fetching EclipseLink employs two distinct lazy variants to enable lazy fetching. Despite the compliance of the JPA's lazy annotation, bytecode instrumentation (also known as weaving) is required for actual lazy data retrieval. This instrumentation can be achieved dynamically by using a javaagent property when launching the program, or statically by enhancing the compiled classes through static class rewriting. In contrast, Hibernate enables lazy loading by default, as it relies on entity proxies. When a lazily-fetched attribute is accessed, Hibernate retrieves all the fields of an entity to transition from a proxy instance to a concrete entity. Additionally, Hibernate uses bytecode instrumentation to inject vendor-specific features, allowing fetching of a single lazy field, rather than loading all the fields, as is implemented by the proxy-based approach.

Batch Processing One limitation of JPA is the absence of a defined batch processing mechanism, especially when working with entities. The specification does not provide a standard approach to consolidate multiple update queries into a single operation treated as a batch. For instance, if one needs to update 100 records, 100 individual queries would have to be executed without batch processing, instead of one single batch operation grouping all of these updates. Fortunately, both EclipseLink and Hibernate offer support to address this gap, enabling a fair comparison between them.

Partial Update This setting is a specific feature of Hibernate. By default, when the framework generates an update query, it sends an update for all fields, even if there are no changes in multiple fields. We considered enabling this feature in Hibernate to align with the default behavior of EclipseLink.

For each ORM framework, we thus derived configurations activating each[1] of the 3 features independently or in combination, resulting in 14 different configurations. For each configuration, we measured its power usage and goodput metrics and computed an overall efficiency score P_{conf}, such that $P_{\text{conf}} = \frac{power}{goodput}$, which captures the energy consumption per successful transaction.

Then, we compared each pair of configurations in terms of their respective P_{conf}. Figure 4.4 presents the result of such pairwise comparison at a rate of $10,000$. Each cell defines the euclidean distance between the P_{conf} of the two associated configurations, computed as $|P_{\text{conf1}} - P_{\text{conf2}}|$. The closer to 0 the value is, the more similar P_{conf} is exhibited by the two compared configurations, hence reporting a little influence of the differing features. For instance, EclipseLink default configuration (EL) P_{conf} is very comparable to the Hibernate configuration with all 3 lazy fetching, batch processing and partial update features activated (Hi-la-ba-up), as it exhibits a distance of 0.0087. Activating such features thus proves particularly efficient to address the performance drop observed for Hibernate in the previous section and highlighted in Figure 4.2(d).

In addition to individual comparisons, Figure 4.4 also outlines group tendencies—*i.e.*, clusters of configurations with similar P_{conf} values—as a dendrogram. To yield these clusters, the data underwent hierarchical clustering using the single linkage method, which relies on the nearest neighbor criterion. The effectiveness of this clustering process was evaluated using the cophenetic correlation coefficient. The resulting high value of 0.96 indicates a robust preservation of the pairwise distances between the original data points. These clusters are organized into a tree-based structure defined on top of the table. For instance, the left branch of the root node relates to 4 columns gathering 4 Hibernate variants, while the right branch groups the other 10 ORM variants together. More precisely, the left cluster corresponds to all Hibernate configurations where the lazy fetching feature is not activated.

The heat map shows that these 4 Hibernate variants are the worst-performing ones. When compared with all EclipseLink variants (the 6 central rows), they exhibit the longest distances, ranging from 0.0522 (Hibernate with batch processing and partial update [Hi-ba-up] *vs.* e.g., EclipseLink default configuration [EL]) to 0.070 (HIBERNATE with partial update [Hi-up] *vs.* EclipseLink with dynamic lazy fetching and batch processing [EL-la-d-ba]). Such distance values highlight the best and worst performing configurations—*i.e.*, [EL-la-d-ba]

[1]When applicable: the partial update feature cannot be activated on EclipseLink as it is a default one.

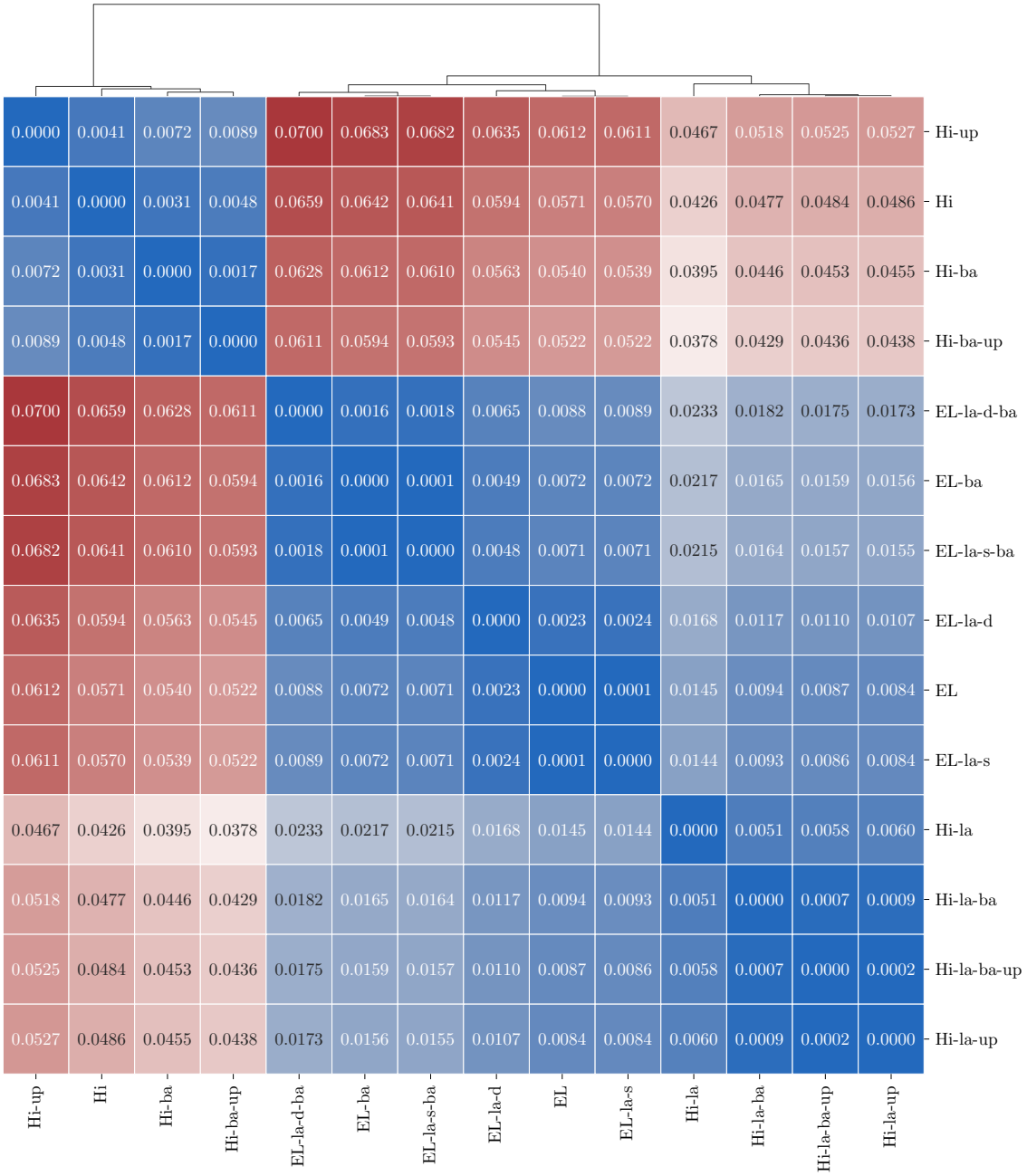


Fig. 4.4 Hierarchical clustering heat map for the Entity strategy with different enabled features (rate 10,000).
la = lazy fetching {→ for EL : s = with static weaving, d = with dynamic weaving}; ba = batch processing; up = partial update.

and [Hi-up], respectively. Within the Hibernate configurations (bottom-right cluster), the best-performing one is the [Hi-la-up]. This variant with lazy fetching and partial update shows the longest distance of 0.527 from [Hi-up] and the shortest distance of 0.0173 from [EL-la-d-ba].

Answer to RQ3: To answer this research question, we studied the impact of 3 different features of different scopes, namely *lazy fetching*, *batch processing*, and *partial update*. Changing the configuration of an ORM framework can influence its power consumption and performance. Specifically, enabling the *lazy fetching* feature for Hibernate appears to have a greater impact than any other factor. Additional performance and energy gains can also be obtained by activating batch processing and partial update features.

4.4 Discussion and Threats to Validity

In this section, we discuss our findings and issues related to the validity of our work.

4.4.1 Discussion

In this chapter, we assessed EclipseLink as the ORM reporting the best-performing queries and being the most energy-efficient, but still less energy-efficient than the plain JDBC approach. From a software engineering perspective, transitioning from an ORM approach to the plain JDBC one involves multiple considerations. While this shift may yield performance advantages, it could also result in extended development time or raise challenges in maintenance. Practitioners must carefully weigh the trade-offs between the convenience offered by an ORM and the potential performance and energy benefits associated with plain JDBC. As shown in Section 4.3, some JPQL or Entity configurations may offer a good trade-off to reduce energy consumption while improving maintenance.

4.4.2 Threats to Validity

Some factors must be considered as they could affect the validity of our contribution.

Internal factors. Our primary objective was to understand if the varying degrees of abstraction maintained comparable performance when considering the specific features and optimizations associated with each ORM. We conducted our experiments with the PostgreSQL 14.5 release. Since ORM frameworks generate queries tailored to the SQL dialect expected by the target DBMS, results may differ when reproducing our experiments

on another release of PostgreSQL or a different DBMS, such as MySQL. Regarding such assessments, the performances of the database itself could also be considered.

Other potential threats to the validity of this study lie in the choice of the JRE [92], and its potential effects on a "warm-up" phase [5], as well as the stochastic nature of the benchmark that may prevent from reproducing exact measurements.

External factors. To assess the energy consumption of ORM approaches, we adopted the well-known TPC-C, widely used to assess DBMS capabilities of online transaction processing systems through an intensive workload. However, ORM approaches may exhibit different performances when assessed against other benchmarks. While we could consider additional TPC benchmarks, like TPC-H [126], ORM frameworks may, unfortunately, fail to deliver native support for some of the ad-hoc queries that involve complex joins, aggregations, and filtering operations, especially when it comes to subqueries.

Since the most popular and widely-used ORM frameworks, only EclipseLink and Hibernate were considered in this study. Results may differ when assessing other ORM frameworks, such as Ebean [32], or non-ORM data access technologies, such as JDBI [52], JOOQ [56] or MyBatis [77].

4.5 Chapter Conclusion

In this chapter, we investigated the influence of ORM configurations on both performance and energy consumption and the potential trade-offs regarding these two metrics.

Using the widely accepted TPC-C benchmark, we assessed the energy overhead introduced by ORM frameworks, compared to data access performed without such abstractions. Across varying workloads, we identified EclipseLink as the most energy-efficient ORM framework. This framework exhibits an 11% overhead in CPU usage compared to the plain JDBC approach at a high transaction rate, an overhead nonetheless mitigated by the benefits ORM frameworks provide, particularly in terms of data consistency between the middleware and the database.

We also uncovered that energy efficiency might come at the price of performance and that developers may face potential trade-offs when configuring their ORM framework, but we showed that Hibernate can be fine-tuned to get closer to EclipseLink performances, providing insights into the varying capabilities of ORM tools when it comes to energy-aware data access.

Chapter 5

Optimizing Performance and Resource Efficiency in Java-based Cloud Native Services

In this chapter, we investigate how application frameworks, configuration choices, and compilation strategies affect both performance and resource efficiency. Given the diversity and popularity of software components available in the Java ecosystem, we focus on Micronaut, Quarkus, and Spring, three Java application frameworks renowned for developing cloud-native services [49]. In addition, these frameworks offer first-class support for compilers based on the GraalVM Native Image project, enabling us to compare the benefits and drawbacks of *Ahead-Of-Time* (AOT) compilation against traditional *Just-In-Time* (JIT) runtime execution, and to assess how different compilation strategies impact efficiency.

Another key aspect addressed in this contribution is configuration. While frameworks are bundled with coherent sets of libraries to support diverse use cases, they can still be configured in numerous ways. Understanding which configuration parameters have a tangible effect on optimizing performance, resource usage, or both, is essential for practitioners seeking energy-efficient and high-performing deployments.

Our objective is to provide empirical evidence on how these factors interact, highlighting trade-offs that practitioners must consider when building and deploying cloud-native applications. Specifically, we address the following research questions:

RQ1: *How do the performance and energy consumption of software built on different web frameworks vary under JIT compilation?*

RQ2: *What is the impact of application framework configuration on system efficiency?*

RQ3: *Does the use of AOT compilation improve the performance and reduce resource usage of web services?*

The main objective of **RQ1** is to examine the variability in performance and energy usage across different web frameworks, to determine to which extent framework design choices impact system operations under varying workloads. Through **RQ2**, we aim to investigate how changes in system's configurations, such as database connection pools settings, affect system's efficiency, offering insights into the trade-offs between performance optimization and energy usage. Lastly, **RQ3** aims to assess the benefits and limitations of AOT compilation by comparing applications built with and without this technique. Through these research questions, we intend to provide empirical evidence that informs developers, architects, and practitioners about the implications of their design and configuration choices. Overall, this work contributes to a broader understanding of how modern frameworks and compilation techniques meet the demands of energy-efficient and high-performance software development.

The remainder of this chapter is structured as follows. Section 5.1 provides some background on application frameworks and compilation tools involved in the development of modern Java applications. Section 5.2 describes our experimental methodology. Section 5.3 presents the results of our experiments and answers the research questions. Section 5.4 discusses the implications of our findings and addresses validity concerns. Finally, Section 5.5 concludes this chapter.

5.1 Background

5.1.1 Application Frameworks

Application frameworks provide standardized collections of tools, conventions, and libraries aiming to simplify, streamline, and speed up the software development process. By abstracting and managing infrastructure aspects such as data serialization and database interactions, these frameworks enable developers to focus on business logic rather than spending resources on cross-cutting technical concerns.

The Spring Framework

One of the most popular, open-source Java application frameworks is the Spring framework [11]. Released in the early 2000s, it has been widely adopted to develop projects

of varying complexity, offering a comprehensive suite of features and tools designed to streamline development processes and enhance software maintainability. To further simplify configuration management, developers often leverage the Spring Boot sub-project, which simplifies the setup and deployment of Spring-based applications by providing predefined settings based on industry best practices and community feedback. Unlike the other two frameworks discussed in this chapter, the Spring framework was not originally designed with cloud computing or cloud-native applications in mind. However, as cloud computing gained popularity and the focus on software development shifted towards cloud-native architectures, the Spring project evolved to provide support for cloud-native application development.

The Micronaut Framework

Since its release in 2018, the Micronaut framework [74] has quickly gained popularity as an open-source Java framework specifically designed for building cloud-native and microservices applications. Unlike Spring, which originally catered to a broader range of application types and later adapted to cloud-native environments, this framework was originally built with cloud-native principles in mind.

The Quarkus Framework

Introduced by Red Hat in 2019, Quarkus [104] is an open-source Java framework designed for cloud-native, containerized, and Kubernetes environments. Like the Micronaut project, Quarkus was built with modern cloud-native development in mind, but it goes a step further by optimizing Java specifically for containers and serverless platforms.

5.1.2 Java Platform Compilation Strategies

The Java platform, originally developed by Sun Microsystems in the early 1990s and later acquired by Oracle, consists of several key components. Among these, the *Java Development Kit* (JDK) provides tools for compiling Java source code into bytecode, an intermediate representation that is then interpreted by the *Java Virtual Machine* (JVM) to allow programs to run seamlessly across different operating systems and hardware. While this architecture ensures the portability of Java applications, the bytecode interpretation can also introduce performance overhead. To mitigate this, the platform has evolved to incorporate *Just-In-Time* (JIT) and *Ahead-Of-Time* (AOT) compilation technologies.

Just-In-Time Compilation

A strategy to reduce the runtime overhead of bytecode interpretation is based on JIT compilers integrated into *Java Runtime Environment* (JRE). These compilers dynamically convert Java bytecode into native machine code, enabling direct execution by the processor, while also performing optimizations such as *dead code elimination* and *method inlining* to improve execution efficiency.

To apply these optimizations, the execution engine profiles the program to identify frequently executed sections of code, known as *hot spots*, which are then targeted by the JIT compiler. This optimization process begins as soon as the program starts running and continues over time, gradually improving the application's performance and eventually reaching peak performance as more hot spots are optimized [99]. This process involves a *warm-up* period during which performance may initially be slower until sufficient profiling data is collected and optimizations are applied [5].

While JIT compilation automatically improves application performance without compromising portability, it also exhibits limitations. The speculative nature of JIT compilers [6] may result in performance penalties when incorrect assumptions about program behavior trigger deoptimization to ensure correct execution. Furthermore, the optimization metadata and compiled native code can increase memory usage over time as both are stored in memory alongside the original bytecode.

Ahead-Of-Time Compilation

This technique aims to reduce the performance overhead of integrating bytecode at runtime by converting it into platform-specific machine code during the build process. This compilation produces native executables, known as *native images*, that can be executed directly by the processor, resulting in faster startup times and reduced memory usage.

AOT compilation is supported by compilers built on the GraalVM Native Image toolkit [132]. Among these, the Oracle GraalVM Native Image [89] provides enterprise-level support and advanced features such as alternative garbage collectors. The GraalVM Native Image Community Edition (CE) provides a different licensing model offering only core features. Furthermore, Mandrel [70] is a distribution specifically optimized for the Quarkus framework.

While this approach is particularly beneficial in environments like microservices, serverless computing, and containerized applications, where quick initialization and efficient resource use are crucial, it also comes with limitations. Since it produces platform-specific native images, it is essential to know the target architecture at build time. Additionally, AOT compilation is not suitable for all types of applications, particularly those that rely on

dynamic features like reflection. Since AOT compilation eliminates reflection to optimize performance, applications that require dynamic behavior or runtime flexibility may not be compatible.

5.2 Methodology

5.2.1 Methodology Overview

To answer research questions, we used an experimental methodology aimed to accurately reflect real-world use cases, where diverse workloads and interactions can introduce significant variability in systems performances. Specifically, our empirical study was conducted on an *On-Line Transaction Processing* (OLTP) system, like those used by online retail platforms. These systems are inherently complex, involving database interactions, server-side processing, user input validation and data serialization. Such characteristics provide an appropriate context for evaluating software based on web application frameworks, since they usually feature out-of-the-box solutions supporting these requirements. Therefore, the benchmarks used in this study are derived from the *Transaction Processing Performance Council benchmark C* (TPC-C) [125] already presented in Section 4.2.1 of the previous chapter. The following sections recapitulate essential elements and present the adjustments made in this contribution.

5.2.2 Web Application Benchmark

Request Profiles

In our evaluation, we used five distinct request types, namely New-Order, Payment, Delivery, Order-Status and Stock-Level, which represent the business operations defined in the TPC-C benchmark. Each request type either retrieves or updates data through dedicated database transactions, depending on the nature of the operation: New-Order places a new order in the system. Payment updates the accounting records to process a customer payment. Delivery simulates a real delivery of new orders by processing a batch of 10 undelivered orders. Order-Status reports the status of the last order placed by a customer. Stock-Level identifies items recently sold that have stock levels below a specified threshold, executing a read-only transaction.

Web Applications

Using each framework introduced in Section 5.1.1, we developed web applications that implement the TPC-C business logic. In this setup, the web application serves as a proxy, translating *HyperText Transfer Protocol* (HTTP) requests into database transactions. To adhere to standard web development practices, request parameter validation was incorporated into the applications (*i.e.*, checks for parameter types and value ranges).

To ensure consistency across implementations, all applications relied on the same core technologies and targeted Java 21, the latest *Long-Term Support* (LTS) release available at the time of the study. Database-related operations were managed using the Hibernate *Object-Relational Mapping* (ORM), with each web framework providing its own integration wrapper. Specifically, Spring Data, Micronaut Data, and Panache were employed for the Spring, Micronaut, and Quarkus frameworks, respectively. Similarly, data exchanged with clients was formatted in *JavaScript Object Notation* (JSON) using the Jackson library. It is important to note that we did not use the reactive programming approach supported by Micronaut and Quarkus. A fully asynchronous comparison would have required the use of Spring WebFlux instead of traditional Spring, as this latter does not support asynchronous processing natively. Nevertheless, when working with reactive frameworks such as Quarkus and Micronaut, we took care to avoid blocking reactive architecture threads with synchronous calls [114, 82], particularly during database communication.

For each implementation, we produced executables using the two compilation techniques discussed in Section 5.1.2. In particular, applications running on a traditional Java runtime environment are packaged in a Docker container that includes an Eclipse Temurin JRE. Although we used Native Image tools from the GraalVM project for AOT compiled applications, we did not use a GraalVM JRE for applications running on standard JVM. This decision stems from the fact that standard GraalVM distributions always include the JDK alongside the JRE, which would result in an impractically large runtime environment for comparison purposes.

For experiments exploring the effects of AOT compilation, we specifically selected the Oracle GraalVM Native Image distribution for its *Garbage-First Garbage Collector* (G1GC) support. This enables a fair comparison with applications running on traditional JREs where G1GC became default since Java 9. In addition, enabling this *Garbage Collector* (GC) is a relevant choice given our experimental setup which comprise multi-threaded applications and machines equipped with large memory capacities [88]. Finally, to further explore the effects of AOT compilation on application efficiency, we built an additional Quarkus-based application using the Mandrel compiler instead of Oracle GraalVM Native Image. Mandrel provides optimizations tailored for the Quarkus framework but only supports the *Serial* GC,

thereby offering an additional perspective on the trade-offs between compilation strategies and garbage collector configurations.

Table 5.1 Overview of web application builds characteristics.

Compilation Strategy	Compiler	Framework
Just-In-Time	Eclipse Temurin	Micronaut Quarkus Spring
Ahead-Of-Time	Oracle GraalVM Native Image	Micronaut Quarkus Spring
	Mandrel	Quarkus

The Table 5.1 summarize the 7 evaluated software variants.

Workload Generator

The workload generation for this study was based on the TPC-C standard, implemented using the BenchBase tool [29, 8]. This tool offers a fine-grained control of the workload characteristics, enabling precise adjustments to stress the system under different scenario and identify potential bottlenecks.

In contrast to the previous chapter, we adapted the BenchBase tool to send HTTP requests containing transaction details to the web applications, rather than executing business transactions directly on the database. These applications then process these requests and execute the corresponding TPC-C-defined operations against the database. As defined in the original workload generator, we configured database transactions at the SERIALIZABLE isolation level [101].

Benchmark Settings

In the context of the TPC-C benchmark, a warehouse serves as a central unit for managing stock levels, processing orders, handling customer interactions, and executing other transactional operations. This makes the number of warehouses a critical parameter, as it acts as a scaling factor that determines both the size of the dataset inserted during database initialization and the workload characteristics. That is, the number of database records for each entity is proportional to the number of warehouses—except for the *Items* table, which remains fixed at $100,000$ entries [125]. In addition, as the number of warehouses increases, requests are spread across a larger dataset, which helps reduce database contention.

To simulate different workload profiles, we varied two factors: First, the *number of warehouses*, was set to two values in our experiments: 5 and 36. The smaller scale factor (5) corresponds to half the smallest default database connection pool size in Spring and Micronaut, creating a compact dataset leading to high contention, even with a small number of database connections. Conversely, the larger scale factor (36) generates a more extensive dataset, aligning with the number of processor threads and reducing contention. Second, the *number of clients* concurrently sending requests to the web application was varied between 5 and 54, using discrete values: 5, 10, 12, 15, 18, 20, 27, 36, and 54. These specific values were chosen to reflect different scenarios. For instance, values like 10 and 20 align with frameworks' default database connection pool configurations, while 18 and 36 match the core and thread counts of the *Central Processing Unit* (CPU), respectively.

5.2.3 Collected Metrics

To answer the research questions, we captured both performance indicators and hardware resource utilization metrics.

Energy Consumption

Using the *Running Average Power Limit* (RAPL) [26] interface, a power management feature available on modern Intel processors, we captured the total energy consumed during each single run. The energy usage of both the processor *package* (PKG) and the associated *Dynamic Random Access Memory* (DRAM) was captured and reported in Joules [116].

Memory Footprint

Effective resource management, especially memory usage, is essential to optimize resource allocation. To measure memory consumption, we used Control Groups version 2 (cgroups v2) [23], a fine-grained resource management and monitoring feature of the Linux kernel. This method enables a detailed and accurate measurement of memory consumption, encompassing not only the physical memory held in DRAM, but also the memory allocated for file system cache and the memory reserved by the kernel for the process. This method provides a more complete representation of resource usage compared to traditional metrics, such as *Resident Set Size* (RSS), which only captures the physical memory held in DRAM. Furthermore, cgroups v2 captures the combined memory usage of a process and its child processes, providing a comprehensive view of the total memory consumption during program execution.

Service Reliability

To assess the systems' reliability, we consider metrics related to request processing as reported by the load generator. These include the total number of requests processed, including both successful and failed ones. In addition, we computed derived metrics such as the error rate (*i.e.*, the ratio of failed requests to total requests submitted), since an efficient system processes a high number of successful requests, close to the total number of requests, whereas an inefficient system exhibits a high rate of failed requests, indicating potential performance bottlenecks or resource limitations. These metrics help evaluate the systems' ability to handle incoming requests effectively and maintain stable performance across varying workloads.

5.2.4 Experiment Infrastructure

To ensure reliable and consistent measurements, all experiments were conducted using the same hardware infrastructure comprising 3 identical Dell PowerEdge R640 servers, interconnected via a Dell Z9264F-ON switch. Each server is equipped with an Intel Xeon Gold 5220 processor running at 2.20GHz, featuring 18 cores and 36 threads, 96GB of memory, a 480GB *Solid-State Drive* (SSD), and dual 25Gb/s network cards. The servers operate on the standard Debian 11 installation, configured solely with the specific applications and services required for this study. To streamline software deployment and execution, all machines share a common software configuration, including Docker [31] and a Python environment comprising the standard library and the pip package manager.

Due to its lightweight nature, Alpine Linux was initially considered as the base environment for the Docker images of the assessed applications. However, its use of the musl standard C library is incompatible with the Mandrel tool used for AOT compilation. Therefore, to ensure consistency across all evaluated applications—whether compiled natively or running on a standard JRE—we selected Ubuntu 22.04 LTS, which provides the popular GNU C library (glibc). Additionally, the Ansible tool was installed to automate the experimental pipeline and ease replicability.

Database Server

For each experiment, the first server hosts a single PostgreSQL 16.4 database instance, which serves as the central data repository for that experiment. PostgreSQL was chosen for its widespread use and popularity as an open-source database solution. The database is installed and managed within a Docker container based on the Alpine Linux 3.20 operating system. By dedicating a system for the database, this setup ensures that both the database and other applications involved in this experiment are isolated, thus minimizing potential measurement

variability from interleaved execution. Furthermore, we also installed the Docker image of the workload generator tool to use the device's local network during the dataset population phase.

Application Server

The second server is dedicated to hosting the application being evaluated, which interfaces with both the database and the client workloads. As with the database server, isolating the web application within its dedicated environment ensures that performance metrics are accurately captured, without interference from the database.

Orchestration Server

The final server orchestrates the experiments by managing their progress and generating the workloads that stress the web applications. It runs the workload generator Docker container, a suite of Ansible scripts, and a daemon service that executes these scripts to carry out the data collection process described in the following section.

5.2.5 Experimental Protocol

The experimental protocol consists of two main steps: deployment and data collection. These steps are designed to ensure reliable and reproducible measurements of performance and resource usage across all evaluated software variants and benchmark configurations.

Deployment Step

Before measuring performance and resource usage, the source code of each web application is deployed to the orchestration server, where the applications are built. This process ensures consistency and reliability in the results as *(i)* all web applications are built on the same device, eliminating variability from differing build environments, and *(ii)* for AOT compilation, the applications are built on a machine with identical hardware settings to the application server—*i.e.*, the runtime environment.

Data Collection Step

This step collects the metrics described in Section 5.2.3. First, a PostgreSQL database is created on the database server and populated with the TPC-C dataset using the workload generator tool. To ensure that the data loading step does not interfere with the stress test step (e.g., data cached by Docker, the *DataBase Management System* (DBMS), or both), the

database container is deleted after the data loading process is complete. A new database container is then created, and once the database is ready, the web application is started on the application server. Additionally, a daemon service running on each server begins continuously recording resource usage metrics at a one-second interval. For each benchmark configuration, 20 executions are performed using the workload generator running on the orchestration server. Each execution lasts one minute and sends a constant mix of benchmark requests, generating detailed reports on request-processing performance.

After completing all 20 iterations for a given configuration, the web application, database, and resource usage recording processes are stopped. To account for power tail states-situations where system components continue to draw elevated power even after a task has completed [116]—a 1-minute idle period was introduced before assessing the next configuration, ensuring more accurate and stable energy measurements. To maintain consistency across executions, the collected resource usage data is then transferred from each measured server to the orchestration server before the database and application servers are restored to their initial state. Then, the protocol proceeds with the next benchmark configuration. This process is repeated for all benchmark configurations and web applications under evaluation, covering all scale factors, software variants and connection pool configurations. In total, the study evaluates 7 software implementations across 2 scale factors, 2 database connection pool values, and 9 client counts, resulting in a dataset of 5040 individual runs.

5.3 Results

This section presents the results of our experiments and answers the research questions. When a change is reported—whether an improvement or a degradation—we quantify that change using the standard percentage change metric: $(\frac{V_2-V_1}{V_1}) \times 100$, where V_1 denotes the reference (baseline) value and V_2 the new value obtained under the experimental condition being evaluated.

5.3.1 RQ1: How do the performance and energy consumption of software built on different web frameworks vary under JIT compilation?

To investigate how system performance and energy consumption are influenced by the selected application framework, we developed and deployed one application for each framework: Micronaut, Quarkus, and Spring. Each application was compiled into Java bytecode,

producing programs designed to run on a standard JVM with default JIT compilation features enabled. We relied on the default configurations of each framework, with one exception: the database connection pool size for Quarkus was reduced from its default of 20 to 10, in order to align with the default configurations of the other two frameworks. This choice reflects a realistic deployment scenario, where multiple services or instances may connect to the same database. Starting with a lower default helps to avoid unnecessary connections, making it a more practical baseline configuration.

To simulate varying workloads and concurrency levels, we stressed the applications with the workloads described in Section 5.2.

Service Reliability

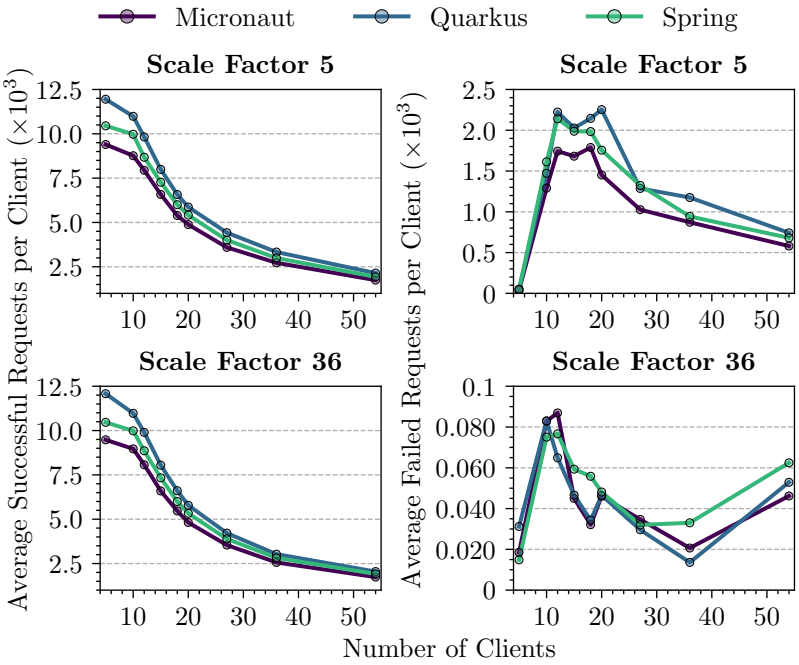


Fig. 5.1 Median of average successful and failed requests per client across frameworks and benchmarks for JIT-compiled web applications.

Figure 5.1 details how the median of average number of successful and failed requests per client varies across benchmarks. For both scale factors, the average number of successful requests per client (left sub-figures) exhibits comparable trends, decreasing steadily once workloads exceed 10 clients. This sustained downward trend in the average number of processed requests per client suggests that the system throughput becomes capped once the number of clients reaches or exceeds the size of the database connection pool. In contrast, the average number of failed requests per client (right sub-figures) rises as the client count grows

from 5 to 12, then fluctuates. This rise is a direct consequence of the heightened concurrency resulting from multiple clients simultaneously accessing the same data as workloads evolve. For example, this is also the case with 54 clients at a scale factor of 36, where the number of concurrent accesses exceeds the scale factor and leads to an increase in the failure rate.

When comparing frameworks, the Quarkus-based application initially outperforms its Spring and Micronaut counterparts in handling client requests. However, this difference narrows as the client load increases, with all frameworks eventually converging toward similar performance under high concurrency. For instance, at a scale factor of 5 with 10 clients, the Spring-based application achieves a median of 9,976 successful and 1,612 failed requests per client. The Quarkus variant performs slightly better with 10,986 successes and 1,472 failures, while the Micronaut implementation records 8,764 successes and 1,290 failures per client. Despite these variations, all three applications maintain a comparable success rate close to 87% for this benchmark.

At higher concurrency, such as with a scale factor of 36 and 54 clients, success rates increase to about 97% across all frameworks. Spring achieves medians of 1,912 successful and 62 failed requests per client, compared to 2,046 successes and 52 failures for Quarkus, and 1,735 successes and 46 failures for Micronaut.

Energy Consumption

Figure 5.2 presents the medians of total energy consumption for the web applications and their associated PostgreSQL database, shown across the different scale factors, frameworks, and client counts. The four sub-figures each exhibit a stabilization of energy consumption once the number of clients exceeds the size of the database connection pool. This plateau mirrors the throughput saturation identified in Figure 5.1 and echoes this earlier observation that the connection pool configuration may be a bottleneck.

Across all benchmarks, the Quarkus-based application consistently exhibits the lowest energy consumption, although its associated database remains the most energy-intensive system. In contrast, the Micronaut-based application demonstrates the highest energy consumption in nearly all workloads (except for the case with 5 clients at a scale factor of 5 where its footprint is lower), while its associated database presents the lowest energy consumption. Meanwhile, the Spring-based application and its database consistently fall between these two extremes, exhibiting intermediate levels of energy consumption. For instance, the web applications consumes 4,333J (Quarkus), 4,711J (Micronaut) and 4,519J (Spring), while the associated databases use 4,390J, 4,183J and 4,328J for Quarkus, Micronaut and Spring, respectively.

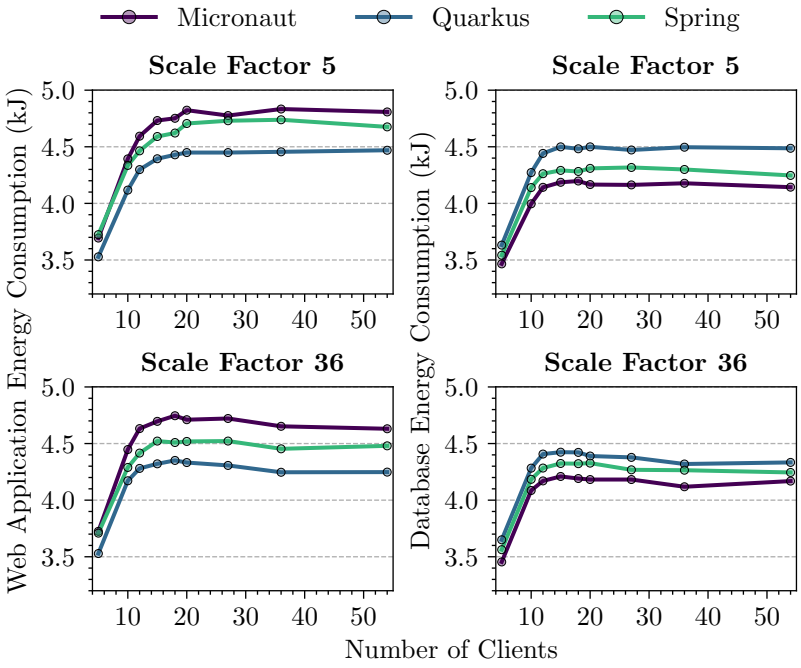


Fig. 5.2 Median energy consumption (kJ) of JIT compiled web applications and their associated PostgreSQL database across frameworks and benchmarks.

Anomalies in Database Connection Management within Quarkus

By investigating usage statistics of database instances, we identified inconsistent interactions between the Quarkus-based web application and the database. Although we fixed the number of pooled database connections via the framework settings, a profiled replay analysis highlighted that all connections are intermittently flushed from the pool and the application occasionally use more connections than expected, exploiting the PostgreSQL's default configuration (which allows a maximum of 100 concurrent connections). This finding highlights potential issues related to framework's connection management. To provide a fair and accurate basis for comparing the frameworks under identical limits, we repeated this experiment by limiting the number of connections at the database level. This ensured that no application could acquire more connections than allowed, but the Quarkus application may still attempt to open additional connections, which then fail.

The Figure 5.3 illustrates a shortcoming in the Quarkus-based application by reporting the percentage change in the total number of processed requests (including success and failures) between two experiment settings: *(i)* limiting only the connection pool size at the framework level, and *(ii)* limiting the number of allowed connection at the database level. In both settings, the Spring and Micronaut applications exhibit consistent throughput, confirming that

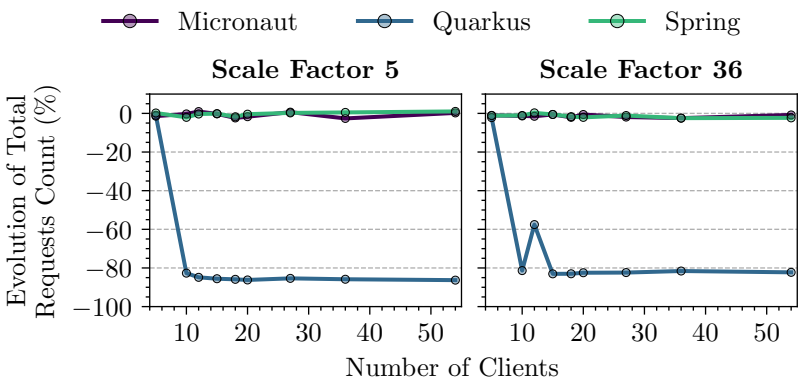


Fig. 5.3 Median change (%) in the total number of requests, with a PostgreSQL configuration limited to 10 pooled connections.

these frameworks comply with the framework-level configuration. The minor fluctuations associated with these two frameworks reflect normal workload variations. By contrast, the Quarkus-based system shows a large drop in the number of processed requests, revealing inconsistencies in how database connection are managed.

The throughput degradation of the Quarkus-based application exceeds 80% for every client count greater than 10 at both scale factors, except for the case of 12 clients with a scale factor of 36. In this particular benchmark the degradation is less pronounced, although the loss remains substantial (approximately 60%). We hypothesize that this reduced degradation is due to a more effective concurrency management (e.g., better thread scheduling) for this client count.

To understand the root cause of this issue, we profiled the Quarkus-based application by replaying the evaluation scenarios outside the main experiments, and identified that the problem can be traced back to how connections are managed within the Agroal connection pool—the default database connection pool implementation bundled with the Quarkus framework.

Our experimental setup creates the context in which two problems in the library become evident. All benchmark transactions were executed at the SERIALIZABLE isolation level [101], causing PostgreSQL to roll back a transaction when two client sessions conflict. This processing is expected and simply reflects the conditions under which we measured the system.

However, the Narayana transaction manager integrated with Agroal erroneously flushes the pool and creates a new set of connections each time the database performs an unilateral rollback. A database initiated rollback should be treated exactly like an application triggered rollback, *i.e.*, the connection should be returned to the pool for reuse.

Second, we identified a race condition that, when a connection is borrowed from the pool, can cause the pool to create more connections than the configured limit, especially after a recent flush of the pool. The race condition therefore allows the pool to temporarily exceed its configured capacity, but this does not invalidate the comparison between frameworks. Within each of Figures 5.1 and 5.2, the results remain comparable because, when a connection is returned to the pool and the total number of open connections surpasses the configured maximum, this additional connection is closed immediately.

In contrast, Hikari—the default database connection pool implementation bundled with Spring and Micronaut—never creates connections above the configured pool size limit.

Answer to RQ1: The results indicate that framework choice influences performance and energy consumption, but other factors, such as the system's configuration, could be equally or more influential.

All three frameworks demonstrated comparable levels of efficiency under low and high concurrency environments. However, our analysis also revealed that the underlying connection pool implementation is a determining factor in the observed differences.

Overall, the consistent decline in the average number of processed requests per client indicates that system throughput becomes constrained once the client count equals or exceeds the database connection pool size. This suggests that the fixed pool size of 10 connections represents a limiting factor, and that beyond this threshold, throughput is driven more by the number of available database connections than by the choice of framework.

5.3.2 RQ2: What is the impact of application framework configuration on system's efficiency?

Optimizing configuration settings within a chosen framework might yield efficiency gains beyond mere framework replacement. To answer this question, we thus evaluated the 3 web applications compiled to run on a standard JVM with JIT features enabled again. However, this time, the number of pooled database connections was fixed to 20 to align with the default configuration of the Quarkus framework.

In this section we first present the data obtained with the default PostgreSQL configuration (which may benefit the Quarkus-based application) in order to assess the effect of a larger connection pool. Considering the shortcomings described in Section 5.3.1, we also evaluate the performance of the applications when the PostgreSQL database configuration is adjusted.

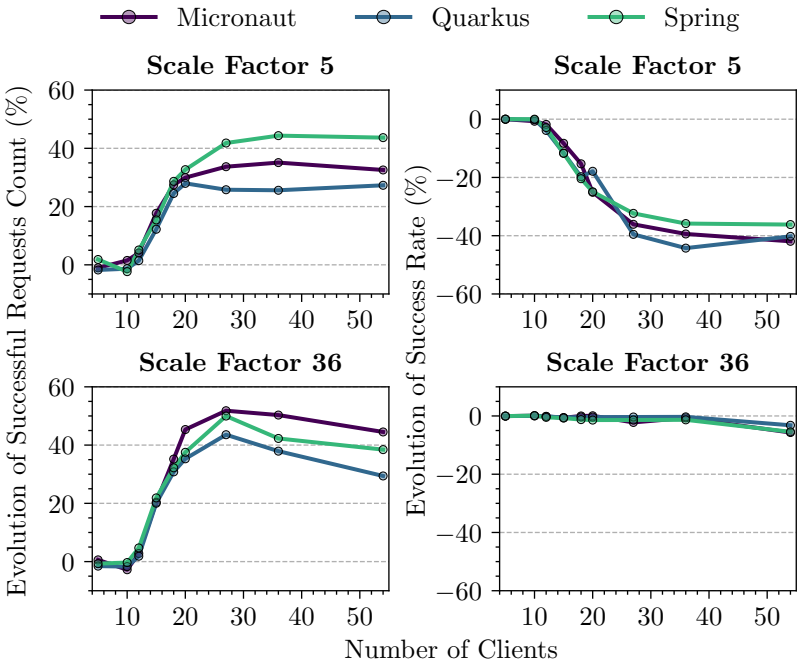


Fig. 5.4 Median change (%) in the number of successful requests and success rate for JIT-compiled web applications when adjusting frameworks configuration from 10 to 20 pooled database connections.

Service Reliability

Figure 5.4 shows the percentage changes in the total number of successful requests and success rate when the database connection pool size is increased to 20.

For both scale factors, increasing the number of pooled database connections enable more successful requests from 12 clients onward (left sub-figures).

Nevertheless, at scale factor 5, the improvement is counterbalanced by a decline in the success rate as the number of clients grows.

For example, at a scale factor of 5 with 20 clients, the Spring-based application experiences a degradation in success rate of approximately 25%, even though the count of successful requests improves by 33%. For 5 and 10 clients, the minor variations observed reflect normal workload fluctuations between the 10 and 20 connections configurations, respectively. For scale factor 36, the error rate remains roughly constant and stays below 5% across all client loads, with the worst case occurring at 54 clients.

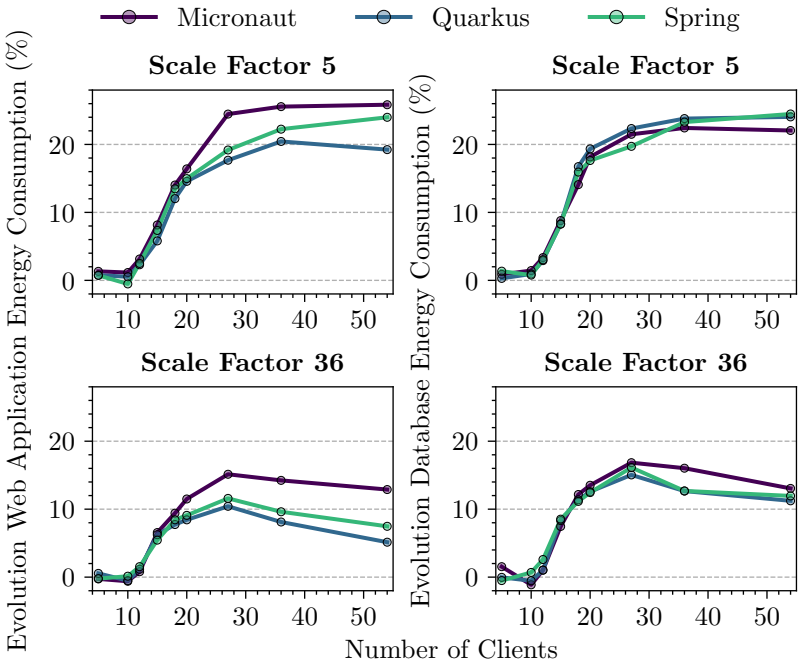


Fig. 5.5 Median change (%) in energy consumption of JIT compiled web applications and their associated PostgreSQL database when adjusting frameworks configuration from 10 to 20 pooled database connections.

Energy Consumption

Figure 5.5 presents the percentage changes in the median of total energy consumption for the web applications and their associated PostgreSQL database when the number of database connections is increased to 20, across the different scale factors, frameworks, and client counts. Interestingly, the percentage increase in energy consumption is higher at scale factor 5 compared to scale factor 36, while the rise in concurrency led to a higher rate of failed requests. With a scale factor of 36, doubling the connection pool size results in a moderate increase in energy consumption. This is expected, because the larger number of connections allows the system to process more requests, as shown in Figure 5.4.

Anomalies in Database Connection Management within Quarkus

As reported in Section 5.3.1, Quarkus exhibits inconsistencies. Figure 5.6 shows the total number of requests processed when the PostgreSQL database is limited to 20 connections.

At both scale factors, fluctuations are noticeable for workloads with fewer than 20 clients. Once the number of clients reaches the configured connection limit, the degradation in the total number of processed requests by the Quarkus application always exceeds 80%.

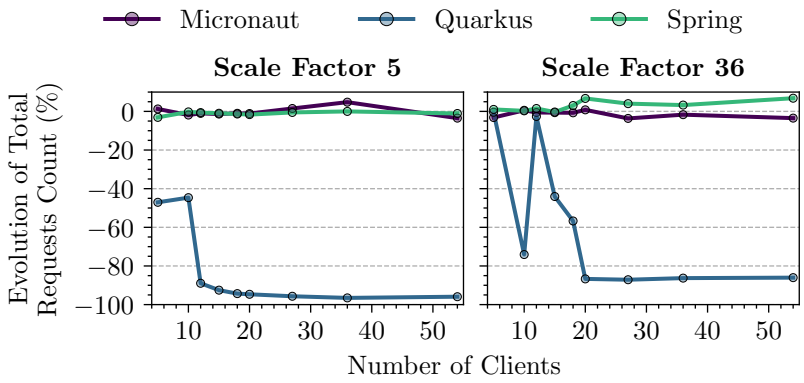


Fig. 5.6 Median change (%) in the total number of requests using JIT compilation for web applications, with a PostgreSQL configuration limited to 20 pooled connections.

At scale factor 5 (left sub-figure), the 5 and 10 clients workloads present a less pronounced degradation, but the loss still exceeds 40%. In contrast, at scale factor 36 (right sub-figure the percentage difference observed for the 5 clients workload is negligible, which is also true for the 12 clients workload. For the 10, 15, and 18 clients workloads the degradation does not reach 80% but remains substantial.

Overall, these trends confirm the issues related to the Agroal library detailed in Section 5.3.1.

Answer to RQ2: Increasing the size of the database connection pool can significantly enhance throughput, but its effectiveness depends on the workload characteristics. For a scale factor of 36, doubling the pool size resulted in a notable increase in the number of successful requests—by an average of 24% across all frameworks and workloads—without a substantial rise in the failure rate. This increase also resulted in better energy efficiency, as more requests were successfully handled with only a marginal increase in energy consumption. In contrast, for a scale factor 5, the benefits of a larger connection pool were limited. At higher client counts, the additional connections led to increased contention, which in turn caused a higher failure rate. This not only reduced service reliability but also resulted in wasted energy, as most of the consumed energy processing a higher volume of failed requests.

5.3.3 RQ3: Does the use of AOT compilation improve the performance and reduce resource usage of web services?

Ahead-Of-Time compilation offers opportunities to improve Java application performance, especially by shortening startup times and reducing memory footprint. Yet, the practical ben-

efits and trade-offs of this approach have not been thoroughly explored for web services built on application frameworks. Consequently, this section investigates how AOT compilation influences the reliability, energy consumption, and memory usage of such web applications.

In the experiments described previously we showed that, with a scale factor of 5, using 10 pooled database connections minimizes failures and reduces energy waste, whereas at a scale factor of 36 the configuration with 20 pooled connections yields the best overall efficiency.

To answer the research question, the present analysis is therefore limited to these two benchmark configurations, allowing us to assess to what extent AOT compilation can further improve system efficiency.

Service Reliability

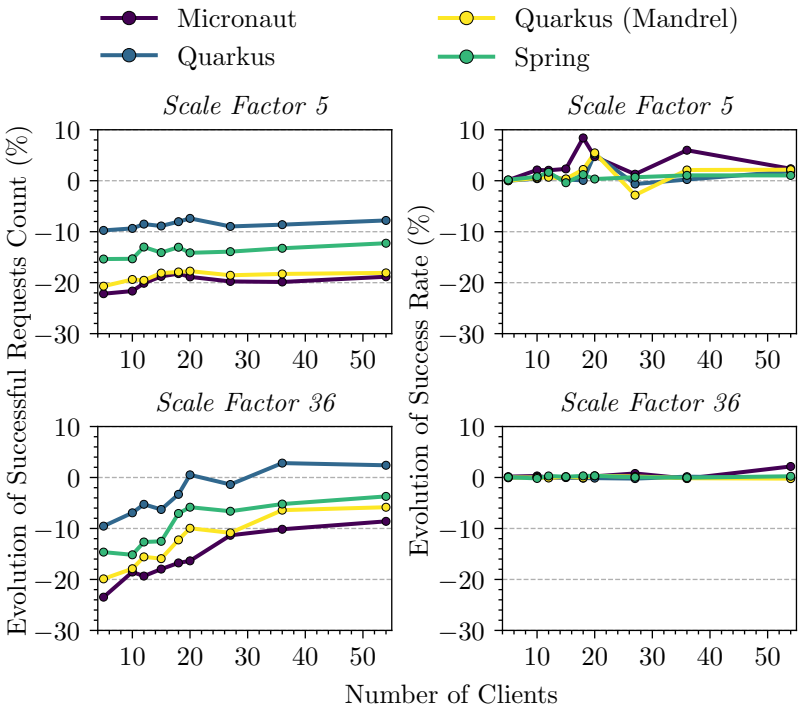


Fig. 5.7 Median change (%) in the number of successful requests and success rate when using AOT compilation compared to the JIT strategy (for 10 database connections benchmarks).

Figure 5.7 shows the percentage change in total successful requests and in the success rate for AOT compiled applications compared to their JIT-based counterparts running on a standard JVM. For the scale factor 5, every framework records fewer successful requests when AOT compilation is used. The success rate shows only small variations (less than 10%) for some workloads and remains essentially constant between the two compilation strategies. For the scale factor 36, the difference in the percentages of the successful requests

count between AOT and JIT-based applications narrows as the number of clients grows. A plateau is observed for workloads with 20 clients or more, which results from the limited number of available database connections. At 54 clients all frameworks exhibit less than 10% deterioration. For each framework the success rate stays roughly constant across all workloads.

Among the applications built with Oracle GraalVM Native Image, Quarkus shows the smallest degradation for both scale factors, followed by Spring and Micronaut. The Quarkus build compiled with Mandrel displays a higher degradation, likely because the Serial GC introduces longer application pauses during garbage-collection, compared with its Oracle GraalVM Native Image counterpart.

These observations suggest that AOT compilation yields a lower throughput than JIT compilation.

Energy Consumption

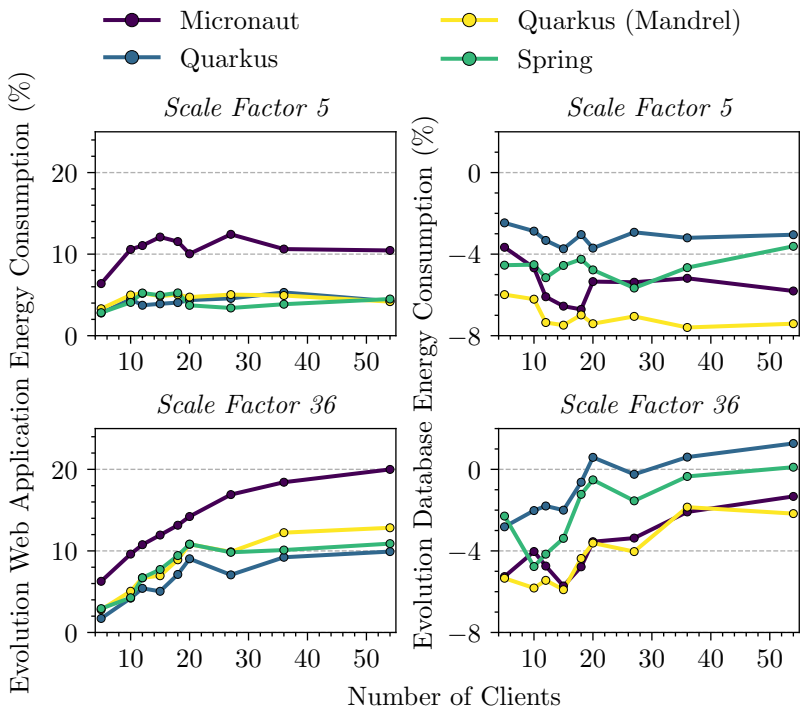
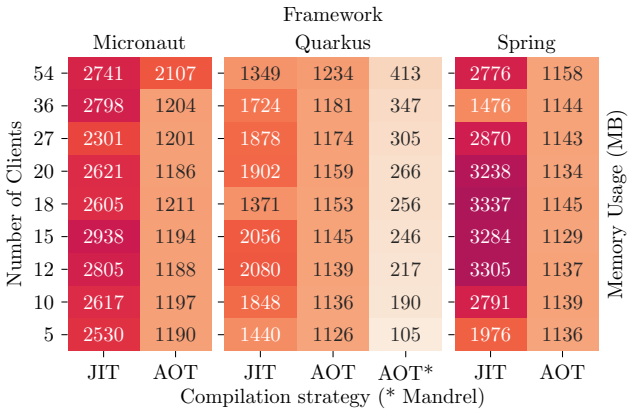


Fig. 5.8 Median change (%) in energy consumption of web applications and their associated PostgreSQL database when using AOT compilation compared to the JIT strategy (for 10 database connections benchmarks).

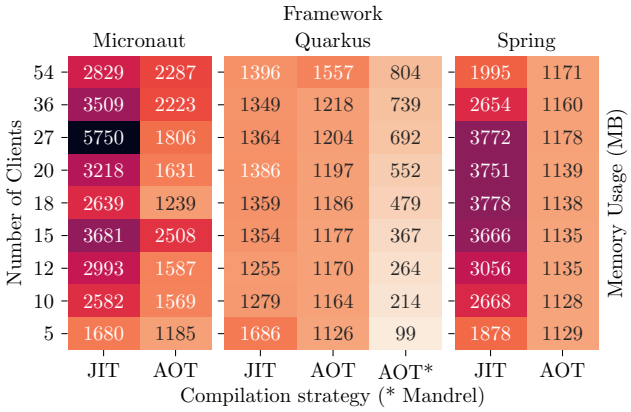
Figure 5.8 outlines the percentage change in energy consumption for AOT compiled applications compared to their JIT-based counterparts running on a standard JVM.

For both scale factors, the AOT compilation strategy results to a higher energy demand for the web application tier. The Micronaut-based web services exhibits the most pronounced increase, with energy use rising by roughly 20% compared with its JIT version. In contrast, the energy consumption of the associated databases decreases when AOT is employed, which is consistent with the reduced throughput observed in the previous section. This pattern underscores the trade-off introduced by AOT compilation between execution speed and overall energy efficiency.

Memory Usage



(a) Scale Factor 5 - 10 Pooled Database Connections



(b) Scale Factor 36 - 20 Pooled Database Connections

Fig. 5.9 95th percentile of memory usage (MB) across compilation strategies, frameworks, and benchmarks.

In Figure 5.9, we report the 95th percentile of memory usage, which represents a threshold where 95% of the measurements fall below. This metric offers insight into sustained high

memory consumption while filtering out transient, rare spikes caused by factors such as garbage collection, temporary memory allocations, or JIT operations (e.g., during code compilation). By investigating this "worst-case scenario", we aim to better understand memory usage across compilation strategies, particularly during the steady state of program execution on a standard JVM.

Several trends can be observed. First, across the various benchmark configurations, AOT compilation tends to reduce memory usage. This is highlighted by dark-colored areas, which represent higher memory consumption in non-AOT configurations. Second, among the evaluated applications, the Quarkus variant compiled with Mandrel consistently exhibits the lowest memory footprint. This difference can be attributed to the type of GC used in each setup: while Oracle GraalVM Native Image was configured with the G1GC, Mandrel relied on the *Serial* GC (as discussed in Section 5.2.2).

Aside from the Quarkus application compiled with Mandrel, at a scale factor of 5 the transition from JIT to AOT yields the greatest memory-savings for the Micronaut and Spring applications. At a scale factor of 36, among the applications built with Oracle GraalVM Native Image, the Micronaut-based application shows the highest memory usage, consuming 2508MB of RAM for the 15 clients workload. For the same workload, the Quarkus application uses about 1177MB, the Spring-based service uses 1135MB, and the Quarkus variant compiled with Mandrel requires only 367MB.

Answer to RQ3: Our findings show that AOT compilation offers memory efficiency benefits, sometimes at the cost of slightly increased energy usage and a marginal drop in throughput. These results highlight that the choice between AOT and JIT should be guided by specific application requirements and deployment contexts. Our results also show that memory usage among *native images* is influenced by the choice of garbage collection strategy, underscoring the importance of runtime configuration even in precompiled settings.

5.4 Discussion and Threats to Validity

This section discusses the implications of our findings and the potential threats to the validity of our work.

5.4.1 Implications

The findings of this empirical investigation suggest that beyond framework selection, tailoring their configuration for a given data architecture and load profiles is crucial on improving system efficiency.

By modifying the number of connections between the web application and the database for different levels of concurrency, our results show that this parameter is critical to optimize both performance and resource utilization of distributed system wide. This highlights the importance for practitioners to profile applications in realistic context and fine-tune configuration parameters rather than relying only on default settings.

5.4.2 Threats to Validity

Construct factors. In this work, energy consumption was measured using the RAPL interface, focusing on the energy usage of the processor (PKG) and DRAM. Measuring memory usage is critical for assessing resource efficiency, especially in environmental contexts where increased memory consumption can lead to the need for setting up new machines. However, our study does not take into consideration other hardware components, such as network interface cards or motherboard power consumption. Considering these factors could contribute to a broader understanding of energy usage and should be considered in future research. Additionally, the observed results may be influenced by third-party dependencies. For example, Micronaut relies on Netty as its web server, and its performance and energy consumption could be affected by how Netty integrates with AOT compilation tools. Such dependencies introduce an additional layer of variability that could be explored in future studies.

Internal factors. Accurately measuring performance and energy in the Java ecosystem can be challenging due to non-deterministic factors, such as garbage collection and JIT effects [5]. To mitigate threats arising from the variability in the measurement processes, we repeated experiments multiple times under identical conditions.

The exclusive use of a PostgreSQL database and the TPC-C benchmark represents another limitation. While the TPC-C dataset ensures reproducibility, database operations-such as query optimization or page locks—may have influenced results.

Developer expertise bias also represents a common threat in empirical software engineering studies. To mitigate this, the core business logic was implemented in a generic manner, and framework-specific code was adapted following best practices and official documentation for each framework.

External factors. The use of the TPC-C benchmark represents a potential limitation. While TPC-C is a widely recognized standard for transaction processing systems, it may not fully represent the diversity of application processing logic and architectures encountered in real-world scenarios. Alternative benchmarks, such as the Yahoo! Cloud Serving Benchmark [24], TPC-W, and TPC-App were also considered, but the latter two were ultimately excluded due to their deprecated status [63]. We finally selected TPC-C because of its active use in research and the availability of tools for dataset generation.

Another limitation stems from our focus on the Java platform. While this decision was motivated by its widespread use in enterprise environments and its rich ecosystem of frameworks and build tools, our findings may not directly translate to other programming languages, especially those not offering a differentiated compilation approach. Finally, while the evaluated application frameworks—Spring, Quarkus, and Micronaut—are commonly used for developing web services, they can also be applied to other types of software. As such, our findings provide developers guidelines on factors to consider for energy efficiency and performance improvements, even outside the context of web services.

5.5 Chapter Conclusion

In this chapter we examined the impact of Java web application frameworks, configuration settings, and compilation strategies on performance and energy efficiency. Through an empirical evaluation of Micronaut, Quarkus, and Spring, we analyzed how framework design, database connection pooling, and compilation techniques influence systems efficiency under varying workloads.

Although the Quarkus-based applications achieved the highest efficiency in most scenarios, the experiments also revealed inconsistencies that can lead to performance degradation under certain conditions. Applications built with the Micronaut framework proved reliable across the experiments, but they consistently consumed more energy and maintained a higher memory footprint in several workloads. Finally, the Spring framework emerged as a balanced alternative, offering reliable performance, appropriate memory management, average energy consumption, and good throughput, thereby providing a practical trade-off for applications that need both stability and reasonable resource use.

Furthermore, increasing database connection pools also improved throughput in some cases, but led to increased contention and failure rates under high loads. Finally, our results show that AOT compilation reduces memory usage across all frameworks. However, the

trade-off comes in the form of slightly increased energy consumption and, in some cases, reduced request processing rates.

These observations stress the need to carefully consider *Service-Level Objectives* (SLOs) and deployment context when developing web applications. In particular, when it comes to web framework parameters, like the connection pools, we recommend practitioners to properly benchmark their application under realistic workloads (database and requests) to appropriately tune their values, and eventually update them along the life-cycle of a system in production. Interestingly, enabling AOT can offer an indirect and complementary impact on energy consumption by reducing the pressure on the DRAM and increasing the consolidation of services on a reduced number of servers, hence saving critical resources.

Chapter 6

Exploring Performance of Configurable Software Systems: the JHipster Case Study

As discussed in Chapter 4 and Chapter 5, the performance of software systems remains a key concern in software engineering. Modern software systems must balance strict performance requirements and provide an optimal user experience, such as low response latency while also ensuring resource and energy efficiency. Consequently, developers are encouraged to adopt technologies capable of meeting these performance goals [91]. However, identifying the most suitable technologies for achieving these objectives is challenging, as performance assessments are influenced by runtime environments, input workloads, and the exponential growth of configuration possibilities in variable systems. In addition, some performance indicators are significantly more complex to measure than others. For instance, assessing power usage requires dedicated benchmarking setups, while metrics like binary size or the number of services are easier to obtain. As a result, developers often rely on default settings or explore only a subset of configurations [81, 96, 40].

Therefore, in this chapter we investigate the impact of configuration choices on system performance. By exploring metrics such as binary size, execution time, and power usage, we aim to help stakeholders determine *(i)* relevant performance indicators, *(ii)* their interrelations, and *(iii)* strategies for designing high-performance configurations.

We conduct our study on JHipster [45, 44], an open-source platform for developing web applications, which offers over $100,000$ possible configurations [76]. As such, it serves as an ideal case study for examining variability in software systems. We selected JHipster due to its alignment with key criteria: open-source accessibility, automation capabilities, and comprehensive evaluation metrics. Widely adopted in industry, JHipster exhibits over $56,000$

commits, 700+ contributors, and 21,700+ GitHub stars, ensuring that its performance has a significant real-world impact.

In this chapter, we aim to answer the following research questions:

RQ1: *How are the different performance indicators related?*

RQ2: *Do the different options of JHipster impact the performance of generated systems in different ways?*

RQ3: *How does the default configuration perform compared to all valid configurations?*

The main objective of **RQ1** is to uncover the relationships among performance indicators that are routinely collected for software evaluation. Given that measuring a large number of indicators is time-consuming and error-prone, we hypothesize that a smaller subset can provide the same insight when some indicators are strongly correlated, thereby enabling developers to monitor systems more efficiently and simplifying the identification of configurations that best fit performance requirements. Through **RQ2**, we aim to determine how the different options available in JHipster influence the performance of the systems it generates. As variability is known to affect performance [96], the selection of a particular option may produce measurable differences in performance indicators. By systematically generating and measuring applications using distinct configurations, we intend to quantify the trade-offs associated with the various options provided by JHipster and help practitioners select the setting that best meets their performance needs. Earlier research shows that users often rely on the default configuration of their systems [133]. However, performance issues related to default settings can affect many deployments. Therefore, **RQ3** aims to compare the performance of the default JHipster configuration with that of all valid configurations, and to quantify the performance gap that may exist between the default and alternative configurations.

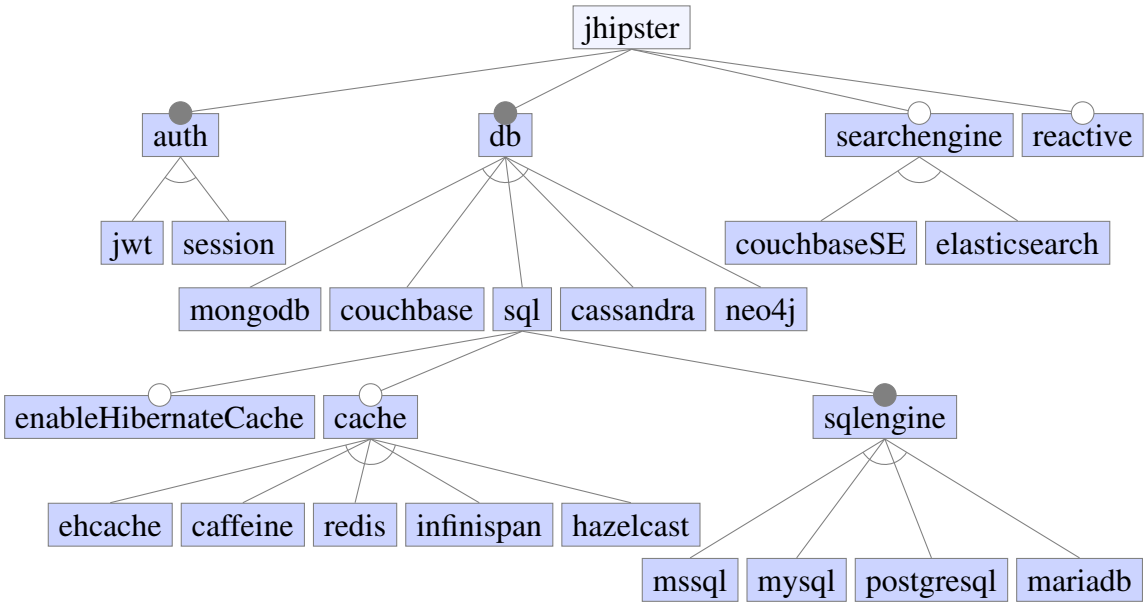
The remainder of this chapter is organized as follows. In Section 6.1 we describe our experimental methodology, present an overview of JHipster's configuration options, and detail the software indicators that were investigated. Section 6.2 explores the correlation between software indicators across 118 JHipster configurations. Section 6.3 uncovers the impact of JHipster configuration options on system performance and investigates how to create efficient configurations through option selection. Section 6.4 discusses the threats to validity and the lessons learned. Finally, Section 6.5 concludes the chapter.

6.1 Experimental Methodology

This section presents our experimental setup, the monitored performance indicators, and our data collection process.

6.1.1 Case Study

In this work, we do not exhaustively test all 100,000 JHipster configurations, as our analysis consists in assessing the performance of the software stack in production. That is, our evaluation excludes options related to *(i)* building tools, such as Maven or Gradle, *(ii)* development databases (H2disk, H2mem, etc.), and *(iii)* virtualization and orchestration tools, as we assume such options do not relate to nor impact software performance in production.



$\text{couchbaseSE} \Rightarrow \text{couchbase}$
$\text{elasticsearch} \Rightarrow \neg\text{couchbase}$
$\text{couchbase} \vee \text{neo4j} \Rightarrow \text{jwt}$
$\text{cassandra} \Rightarrow \neg\text{searchengine}$
$\text{cache} \Rightarrow \text{enableHibernateCache}$
$\text{enableHibernateCache} \Rightarrow \text{cache}$
$\text{reactive} \Rightarrow \neg(\text{cassandra} \vee \text{neo4j} \vee \text{session} \vee \text{enableHibernateCache})$

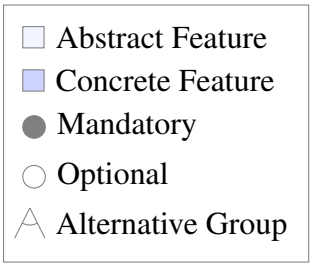


Fig. 6.1 Feature model and constraints capturing the JHipster configurations explored in this chapter

As depicted by Figure 6.1, the JHipster variability under study covers options related to the database system, the cache system, the search engine, the authentication method, and the usage of reactive processing. This feature model was built based on the official documentation of JHipster, by extending and updating previously published JHipster feature models and through source-code mining on the JHipster GitHub repository [53]. This model defines 118 valid configurations, which were all run and tested to ensure that they actually work and that the selected options are properly included. While it does not represent the entire configuration space of JHipster, prior work showed that measuring performance of multiple products at once was feasible by sampling and analyzing a minimal set of products [39]. To answer our research questions, we empirically assessed the performances of all 118 configurations, built using version 7.9.3 of JHipster.

6.1.2 Measured Performance Indicators

The configurations were analyzed *wrt.* a set of performance indicators related to response time, energy consumption, and various static metrics were measured and analyzed. To ensure realistic simulations, response time and power usage are not limited to a single metric. Instead, multiple indicators that cover different usage scenarios of JHipster were considered. This approach allowed for evaluating JHipster's performance under various conditions and workloads.

Response time Response times were measured through Gatling, a benchmarking tool designed to simulate heavy loads on web applications and measure their performance and stability. In particular, we monitored requests, such as the authentication request (`authentication`), the first authenticated request (`authenticated`), the retrieval of all objects belonging to a certain entity (`getall`), the creation of an object of a given entity (`create`), the retrieval of this object (`get`), and the deletion of this object (`delete`). For each indicator, we measured the mean response time.

Power usage To account for execution time variations between configurations, the energy efficiency of a configuration is defined as a consumption rate, *i.e.*, power usage, expressed in Watts. The power usage is captured during a fixed idle period `idle-total` once the stack is booted, and then during the Gatling workload, reported as `wl-total`. To increase the granularity of power indicators, the total power usage is further distributed into two indicators, CPU power usage, and RAM power usage. Thus, six indicators are monitored: `idle-total`, `idle-cpu`, `idle-ram`, `wl-total`, `wl-cpu`, and `wl-cpu`.

Static indicators Some JHipster configurations include external services, such as Redis or Elasticsearch, which increase the number of concurrently running containers. Thus, the number of Docker images `service` and their `size` evolve with configurations, which may impact other performance indicators. The number of selected features `features`, which is different from the number of images, is also monitored. Finally, the stack's boot time `boot-time` is also accounted for in this category, as it is independent of the workload.

6.1.3 Experimental Setup

To ensure reproducibility, all experiments were conducted in the same technical setup. The experiments were performed on the Grid'5000 computing grid [15], on a single-socket server equipped with a 2.20 GHz CPU and 96 GB of memory, running Ubuntu 20-04. The entire server was allocated for this experiment, preventing alterations in performances caused by other users. The specific version of each software component is listed in our open data package. All valid configurations were built as docker images and stored on a repository for reuse.

The assessment of each configuration is automatically performed by the following process: First, the idle power usage of the device is monitored over a 30-second period. Then, the generated stack is started, and the startup duration is logged. Once the stack is ready to accept requests, the power usage is monitored over a 30-second period to compute the `idle-total` indicator. The system is warmed up with synthetic workload. Then, the stack is stressed with the load testing tool Gatling, while the power usage is monitored to compute the `wl-total` indicator. The Gatling workload is the default scenario generated by JHipster to prevent user bias. Specifically, Gatling generated 100 users evenly distributed over a one-minute time window, with each user performing a scenario composed of 19 operations. The response-time indicators are extracted from the reports generated by Gatling.

Energy consumption is measured using *Running Average Power Limit* (RAPL) [61]. This tool provides the total power usage of the computing package, and the specific power usage of the *Central Processing Unit* (CPU) and *Dynamic Random Access Memory* (DRAM) [28]. All the power usages discussed in this work are cleaned of the idle power usage of the device. The idle power usage of the device is measured before each workload to ensure the finest granularity. All experiments were performed after a warmup of the device, to avoid heat-related variations in the power usage [130]. Finally, the performance of each configuration is estimated as the median value over five replications of the measurements.

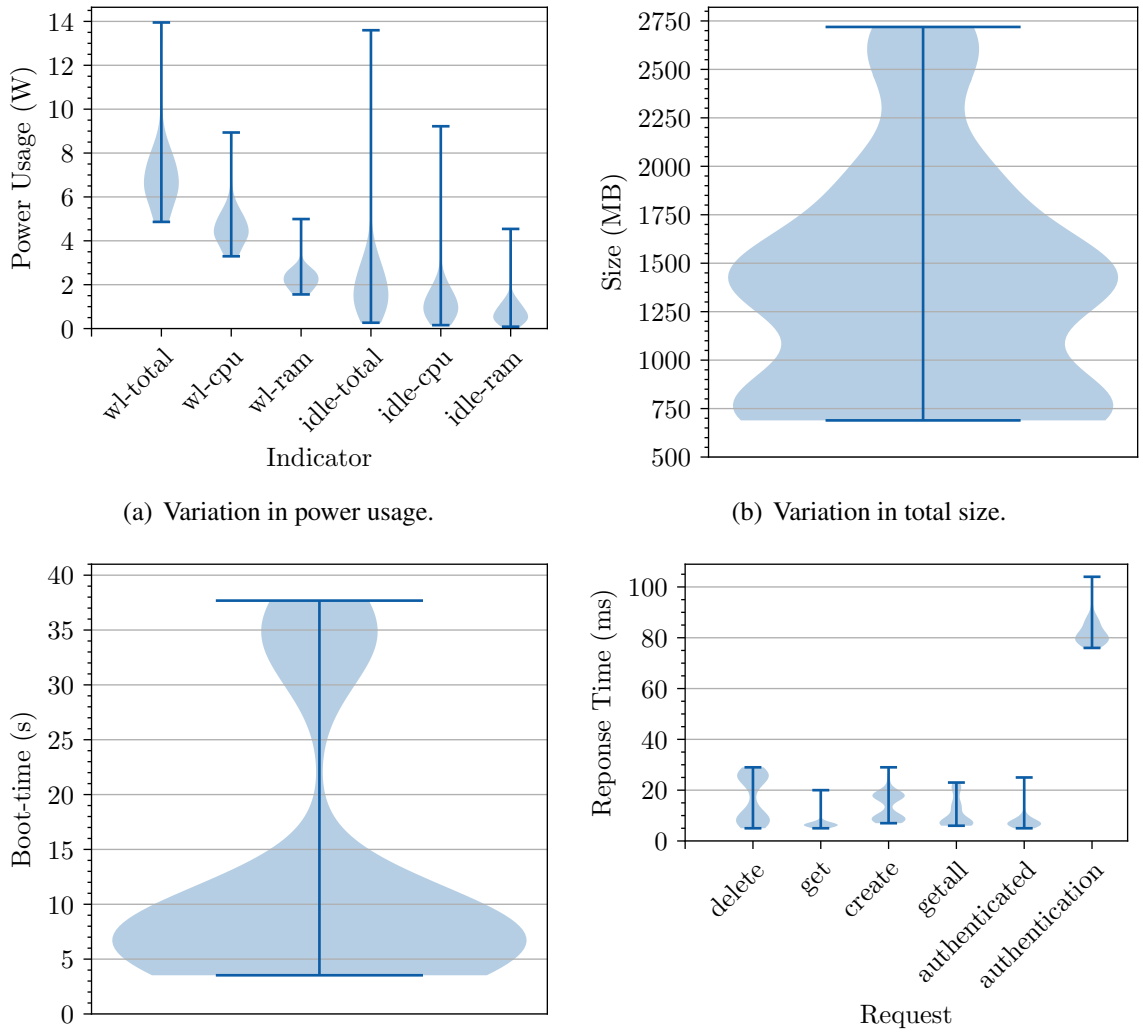
6.2 Performance of Configurations

Table 6.1 Overview of the measured performance indicators.

Indicator	Unit	mean	std	min	25%	50%	75%	max	factor	Description
size	MB	1364.71	605.44	689.00	766.25	1373.50	1687.00	2719.00	3.95	Total size of all containers
services	∅	2.63	0.61	2.00	2.00	3.00	3.00	4.00	2.00	Number of containers
boot-time	s	12.96	11.85	3.53	5.78	6.98	9.80	37.68	10.68	Delay before stack is operational
features	∅	3.95	0.99	2.00	3.00	4.00	5.00	5.00	2.50	Number of selected features
authentication	ms	82.36	5.33	76.00	79.00	81.00	85.00	104.00	1.37	User authentication
authenticated	ms	7.98	3.68	5.00	6.00	7.00	8.00	25.00	5.00	First authenticated request
getall	ms	11.07	5.56	6.00	7.00	8.00	14.00	23.00	3.83	Fetching all objects of an entity
create	ms	13.59	5.20	7.00	9.00	13.50	18.00	29.00	4.14	Creating an object
get	ms	6.91	2.41	5.00	6.00	6.00	7.00	20.00	4.00	Fetching the created object
delete	ms	16.53	8.84	5.00	8.00	12.50	25.75	29.00	5.80	Deleting the creating object
idle-cpu	W	1.34	1.55	0.16	0.69	0.96	1.54	9.22	59.00	CPU when booted and idle
idle-ram	W	0.80	0.78	0.09	0.43	0.63	1.01	4.54	52.59	RAM when booted and idle
idle-total	W	2.14	2.32	0.27	1.10	1.54	2.52	13.60	50.96	CPU & RAM when booted and idle
wl-cpu	W	4.70	1.10	3.30	4.08	4.44	5.07	8.94	2.71	CPU under load
wl-ram	W	2.31	0.63	1.56	1.98	2.27	2.60	4.99	3.21	RAM under load
wl-total	W	7.01	1.72	4.86	6.14	6.76	7.67	13.95	2.87	CPU & RAM under load

This section analyzes the performance of the 118 configurations of JHipster and look into the correlation between performance indicators. Table 6.1 depicts the variation in performances across configurations, for all the measured indicators. This table summarizes, for each indicator, its unit, mean, standard deviation, minimum, maximum, and quartiles. The factor between the maximum value and minimum value is also displayed. While some performance indicators such as `authentication` response times are consistent across configurations, most exhibit substantial variations. In particular, the workload power usage (`wl-total`) varies by a factor of 3, the size by a factor of 4, and some boot times are 11 times longer than others. The total idle power usage, *i.e.*, the power usage when the configuration is running with no workload, varies by factors as high as 51. Therefore, one can conclude that the appropriate configuration of JHipster is crucial as it can have a significant impact on the performance of the generated software system.

Beyond the difference in performances across configurations, some indicators exhibit clustering patterns, as depicted in Figure 6.2. Specifically, in Figure 6.2(c) the boot time follows a bimodal distribution: being either below 10 seconds, or more than 30. Similar behavior is visible in Figure 6.2(b), with `size`, in Figure 6.2(d) with `create` and `delete` response times. Finally, in Figure 6.2(a), most configurations are grouped around the lower values, while some outliers appear to have substantially higher power usage. This distribution raises some questions. *Are the fastest configuration of "create" and "delete" the same configuration? Are they also the fastest to boot? The most power-efficient?* Such questions can be generalized to performance indicators with no obvious clusters, such as `get`.



(a) Variation in power usage.

(b) Variation in total size.

(c) Variation in boot time.

(d) Variation in response time.

Fig. 6.2 Variations in the measured indicators across configurations.

Answering such questions requires a deeper analysis of the correlation between all the measured indicators: if some or all of these performance indicators are strongly correlated, then the assessment and optimization of performance would be substantially simplified.

Finally, it would be beneficial to identify if such behaviors are caused by specific options.

6.2.1 Correlation in Indicator Groups

We analyzed correlations within each group of indicators—*i.e.*, between static indicators, energetic indicators, and response times. The objective is to identify indicators that can be estimated from others, and used as proxies to simplify experimental setup and results

analysis. As shown in Figure 6.2, some indicators exhibit a bimodal distribution. Therefore, all correlations reported in Figure 6.3 are Spearman Correlation coefficients, computed using the median value of five replications for each of the 118 configurations.

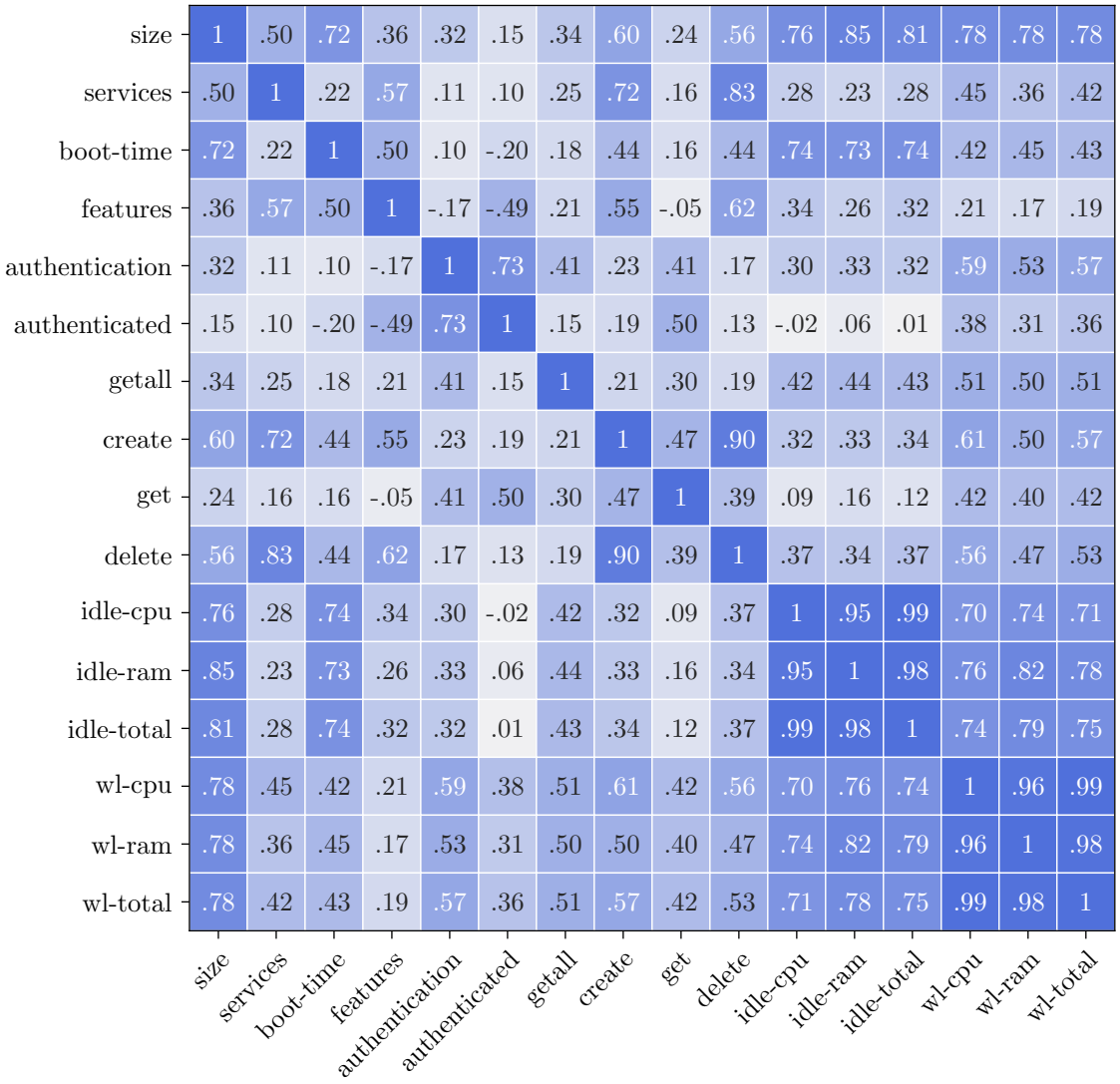


Fig. 6.3 Correlations between measured indicators.

To account for the increased risk of Type I errors (false positives) due to multiple correlation tests performed on the same dataset, we applied the *Benjamini-Hochberg* procedure to control the *False Discovery Rate* (FDR). The FDR-adjusted p-values are detailed in Figure 6.4. Correlations coefficients higher than 0.75 are considered as absolutely strong, and correlations lower than 0.5 are considered as absolutely weak. Only the correlations with adjusted p-values below 0.05 are considered significant.

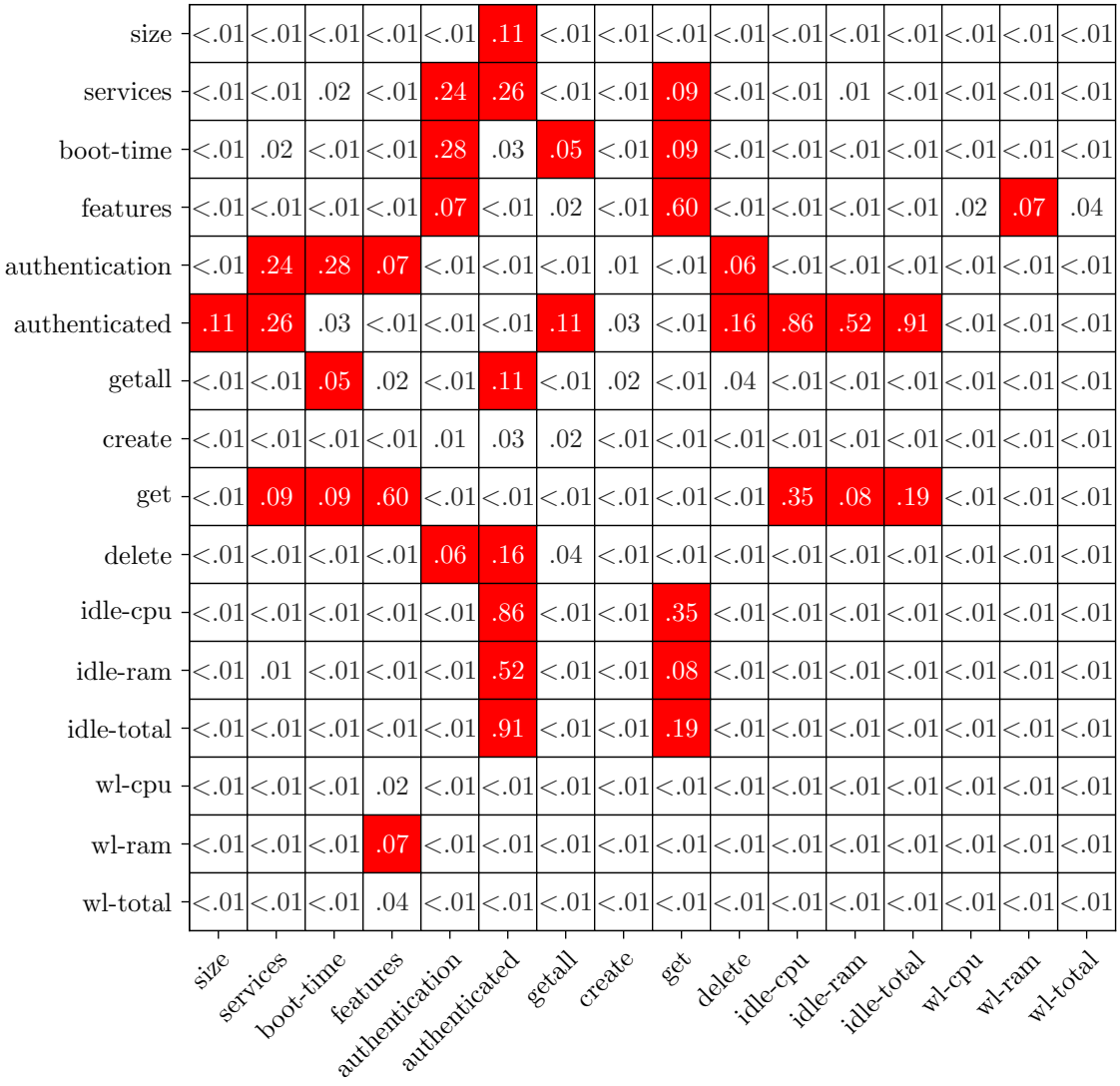


Fig. 6.4 FDR-corrected p-values of the correlations.

Correlation among static indicators Static indicators include size, services, boot-time, and features. While all the correlations are positive, some are weaker: in particular, the correlations boot-time and services or features are 0.22 and 0.5, respectively. By contrast, boot-time is strongly correlated to size, by 0.72. The indicators services and features are correlated by 0.57. This can be explained by the design of the variability: the services are represented as features in the feature model, but not all features are services. Finally, services and features appear weakly correlated to size, by 0.5 and 0.36, respectively. Thus, a higher number of features or services tends to increase size, which in turn affects boot-time. Therefore, size may be a relevant indicator to indirectly optimize services, boot-time, and features metrics.

Correlation between Response Times The correlations between response times indicators are all positive, suggesting that faster configurations tend to be faster in all indicators. However, the correlations range from 0.13 to 0.9. The `authentication` and `authenticated` requests are strongly correlated—*i.e.*, 0.73. They are only weakly correlated to other requests, such as `get` request (0.41 and 0.50) and `create` (0.23 and 0.19). The `get` and `getall` indicators only have correlations of lower magnitude, below 0.5, with the other response times. Finally, the `create` and `delete` requests are strongly correlated (0.9). Such a correlation tends to confirm the hypotheses formulated about the clustering of values in Figure 6.2(d): the slowest configurations on `create` are also the slowest in `delete`. The `delete` request is only weakly correlated to other response times, with a correlation lower than 0.39. The `create` request has weaker correlations to the other response times, between 0.3 and 0.5.

Correlation between energetic indicators All the energy-related indicators are strongly correlated, with correlations ranging from 0.7 to 1. The CPU and RAM power usage are strongly correlated, both in idle (0.95) and under load (0.96). Across idle and workload, the performance of the CPU and RAM are also correlated: their respective idle and workload power usages are correlated at 0.7 and 0.82, respectively. Consequently, the total power usage in idle is strongly correlated to the power usage under load (0.75). Since the total power usage is computed as the sum of the CPU and RAM power usage, these indicators are not independent and their correlations to the total power usage are not relevant in both idle and workload states.

6.2.2 Correlations Across Indicators Groups

While some proxy indicators appeared inside each indicator group, assessing the performance of a configuration still requires a dedicated analysis for each indicator group. Therefore,

to further simplify performance assessment and optimization, it appears relevant to explore proxy indicators that enable the estimation of performance across different groups of indicators.

Static indicators and time The `create` or `delete` requests response time have the strongest correlation to static indicators, in particular the number of services (0.72 for `create` and 0.83 for `delete`). The correlations between `create` and `delete` and other static indicators may be the consequence of the internal correlations between static indicators. Aside from the `create` and `delete` indicators, the response times are only weakly correlated with static indicators. The correlations range from -0.05 to -0.49. Thus, only `services` is a potential proxy indicator for `create` and `delete`, and static indicators do not provide a means to assess the response time of the other operations.

Static indicators and power The correlation between power indicators and `size` ranges from 0.76 to 0.85. In particular, `size` is strongly correlated to all energetic indicators (between 0.76 and 0.85), while `boot-time` is strongly correlated to the idle power usage (between 0.73 and 0.74). The correlation between the remaining static indicators `service` and `features` and the energetic indicators are weaker, between 0.17 and 0.45. However, `service` is overall more correlated to energetic under load (0.36 to 0.45) than in idle (0.23 to 0.28), whereas `features` exhibits the opposite behavior (0.26 to 0.34 in idle, and 0.17 to 0.21 under load). Therefore, `size` may be a proxy for idle and load power usages, and `boot-time` may be a proxy only for idle power usage.

Time and power The response times are systematically more strongly correlated to the power usage under load (between 0.31 and 0.61) than to the one in idle (between 0.01 and 0.44). The behavior may be explained by the fact that both response times and power usage underquantify the behavior of the system at runtime.

Furthermore, the power usage of the RAM is less correlated to response time than the power usage of the CPU. The power usage under load is more correlated to `authentication`, `getall`, `create`, and `delete` (between 0.47 and 0.61) than to `authenticated` and `get` (between 0.31 and 0.42).

6.2.3 Proxy Indicators

Some of the analyzed indicators exhibit strong correlations. In particular, all the energetic indicators are strongly correlated with each other. The idle CPU and RAM power usage are strong predictors of the total idle power usage, and the CPU and RAM usage under load

are strong predictors of the total power usage under load (with R^2 of respectively 0.99, 0.98, 0.98, and 0.96 in linear regression). The total power usage in idle is also a good predictor of the total power usage under load ($R^2 = 0.71$). Static indicators are also correlated with each other, and in particular `size` is a good proxy indicator for `boot-time` ($R^2 = 0.68$). Proxy indicators also appear between response times, in particular with `authentication` and `authenticated` ($R^2 = 0.71$) and between `create` and `delete` ($R^2 = 0.84$).

Table 6.2 Mean Absolute Error (MAE) of each indicator, predicted from metrics from all groups versus metrics from other groups.

Indicators	MAE (%) (All groups)	MAE (%) (Other groups)	CV (%)
size	6	5	44
services	7	6	23
boot-time	12	17	91
features	19	17	25
authentication	3	4	6
authenticated	13	21	46
getall	32	31	50
create	7	13	38
get	10	15	35
delete	10	17	53
idle-CPU	9	32	115
idle-RAM	11	27	97
idle-total	8	30	108
wl-CPU	3	7	23
wl-RAM	3	8	27
wl-total	2	8	24

However, proxy indicators within the same group of indicators have a limited interest. Indeed, except for `boot-time`, the ability to measure a given indicator of a group implies the ability to measure all indicators of that group. Across indicator scopes, only a limited set of indicators are correlated: `size` and power usages, and some response times with `service` or the power indicators under load, but such correlations do not convert into strong predictors. In particular, `size` can only approximate the total power usage under load ($R^2 = 0.29$). In idle, `size` is a better predictor of the RAM power usage than the total power usage (R^2 of respectively 0.24 and 0.16). `service` provides better estimates of the responses times of `create` and `delete` (R^2 of respectively 0.51 and 0.68). Among response times, only `authentication` and `getall` allow for estimating the total power usage under load (R^2 of respectively 0.32 and 0.26). Other response times perform worse in such a task ($R^2 < 0.1$).

Thus, predicting an indicator using a proxy indicator remains a challenge across indicator groups. However, using a set of proxy indicators, rather than a single one, may compensate for such correlations. In particular, we explore predicting a given indicator using only indicator from other groups, e.g., predicting the total power usage of the system from the static and response time indicators.

Table 6.2 compares prediction accuracy from each indicator, when using either all other metrics (*i.e.*, metrics from all groups except the target) versus using only metrics from other groups. In addition, the *Coefficient of Variation* (CV) for such indicators is provided, to assess whether such accuracy can be imputed to random changes. Predictions were performed using a random forest regressor. We observe that while most indicators achieve accuracy significantly better than random chance, some exceptions arise. For instance, `authentication` and `features` indicators have a low *Mean Absolute Error* (MAE) but also a low CV, as the collected metrics show little variations, which reduces the practical value of predicting these indicators. Conversely, the idle power usage indicators exhibit higher MAE and CV, reflecting greater variability but also increased prediction difficulty. `authenticated` and `getall` tend to have higher MAE despite limited CV, indicating that such metrics are harder to model for the regressor. Overall, limiting the training to indicators from other groups as a cost ranging from 1 to 22 percentage points in MAE. Nonetheless, the resulting predictions still substantially outperform random chance, enabling the prediction of harder-to-predict indicators by leveraging other indicator groups.

Answer to RQ 1 Some indicators exhibit strong correlations, in particular when they quantify measurements of similar natures, allow the use of proxy indicators to simplify performance assessment and optimization. Such proxies can serve as warning signs that a system might exhibit poor performance in unmeasured indicators.

However, they do not allow for quantifying precisely such a performance without specific measure. Correlations also exist across indicators of different natures, but they tend to be weaker and less frequent. Nonetheless, it is possible to predict such indicators using a compositing of proxies from other indicator groups.

6.3 Performance of Options

This section aims to understand the impact of options of JHipster on its performance and to discover means to create efficient configurations through relevant option selection. As the previous section highlighted that the CPU, the RAM, and the total power usage are strongly correlated (more than 0.96), only the latter is discussed in the remainder of this chapter.

6.3.1 Performance of Individual Options

Table 6.3 Average performance of configurations containing each option.

		size (MB)	services	boot-time (s)	features	authentication (ms)	authenticated (ms)	getall (ms)	create (ms)	get (ms)	delete (ms)	idle-total (W)	wl-total (W)
Database	**Cassandra**	907.00	3.00	4.05	2.00	80.00	8.00	8.00	**7.00**	6.00	**5.00**	1.72	7.37
	Couchbase	1730.00	**2.00**	6.69	2.00	86.00	**6.00**	17.00	**7.00**	**5.00**	**5.00**	13.59	13.51
	MariaDB	695.00	**2.00**	5.13	2.00	80.00	8.00	**6.00**	8.00	6.00	7.00	0.49	**4.93**
	MongoDB	740.00	**2.00**	**3.73**	2.00	**77.00**	7.00	**6.00**	7.00	6.00	6.00	0.58	5.15
	MSsql	1925.00	**2.00**	33.76	2.00	81.00	9.00	7.00	9.00	7.00	8.00	2.41	6.77
	MySQL	765.00	**2.00**	5.39	2.00	82.00	10.00	7.00	9.00	7.00	9.00	1.07	6.82
	Neo4j	869.00	**2.00**	6.68	2.00	86.00	12.00	10.00	12.00	9.00	9.00	1.19	6.45
	PostgreSQL	**689.00**	**2.00**	5.16	2.00	80.00	8.00	7.00	8.00	7.00	7.00	**0.27**	5.04
Authentication	**JWT**	1358.88	2.67	**13.60**	4.06	**78.69**	**6.51**	10.57	13.29	6.39	16.98	**1.78**	**6.61**
	Session	**1358.71**	2.67	13.64	4.06	83.39	7.43	**10.02**	**13.12**	**6.31**	**16.86**	1.82	6.66
Cache	**No cache**	**1353.62**	**2.50**	**12.94**	**2.50**	82.31	9.19	**6.62**	**13.00**	6.62	**16.75**	**1.51**	**6.49**
	Any cache	1385.29	2.70	14.46	4.50	**80.96**	**6.49**	11.30	13.46	**6.34**	17.35	1.90	6.69
	Caffeine	**1355.19**	**2.50**	13.22	4.50	79.25	6.12	7.25	**13.06**	6.25	16.62	**1.46**	6.31
	Ehcache	1355.38	**2.50**	**13.10**	4.50	**79.06**	**6.06**	7.25	**13.06**	**6.12**	**16.38**	1.58	**6.29**
	Hazelcast	1371.31	**2.50**	16.10	4.50	82.31	6.81	13.62	13.50	6.50	17.25	2.96	7.35
	Infinispan	1367.88	**2.50**	16.36	4.50	80.25	6.12	**7.19**	13.81	6.19	17.12	1.63	6.36
	Redis	1476.69	3.50	13.53	4.50	83.94	7.31	21.19	13.88	6.62	19.38	1.88	7.13
CouchbaseSE	**No**	1733.50	2.00	6.43	**2.50**	**86.50**	7.00	19.50	7.00	5.00	5.00	**13.49**	**13.34**
	Yes	1733.50	2.00	**6.35**	3.50	87.00	7.00	**19.00**	7.00	5.00	5.00	13.59	13.85
Elasticsearch	**No**	**1024.48**	**2.14**	**12.84**	**3.52**	**81.96**	**7.95**	10.86	**9.39**	**6.98**	**8.86**	**1.45**	**6.16**
	Yes	1694.95	3.14	13.87	4.52	82.50	8.11	**10.80**	18.50	7.02	25.45	2.04	7.38
Reactive	**No**	**1370.08**	2.42	10.42	**2.50**	**81.08**	**8.08**	**8.33**	**11.83**	**6.25**	**14.50**	3.44	**7.58**
	Yes	1380.83	2.42	**9.37**	3.50	92.42	16.17	16.83	17.25	11.67	14.92	**3.24**	9.03

This section compares the performance of various options available in JHipster. Table 6.3 presents the performance of each option.

Authentication In our experiment, we analyzed two authentication methods: session and *JSON Web Token* (JWT). There is no significant difference *wrt.* consumption and static indicators between the two authentication methods, but some variations can be observed regarding response time. Specifically, the authentication and first authentication requests are slower with `session` than with `JWT` by 4% and 12%, respectively. Then, `session` is slightly faster on the remaining operations `getall` (5%), and less than 2% faster on the remaining operations `create`, `get`, and `delete`.

Databases JHipster supports 8 databases: MySQL, Microsoft SQL Server (MSSQL), Postgres, MariaDB, Mongo, Cassandra, Neo4j, and Couchbase. The total size of the stack differs depending on the database. While most database footprints are between 690 Mb and 900 Mb, the average size of configurations is 1730 Mb for the ones containing Couchbase and 1925 Mb for the ones containing MSSQL. Thus, the choice of database appears to have a strong impact on the size of the system, and can partially explain the spread in size in

Figure 6.2(b). Finally, while most databases have boot times between 3.7 and 6.7 seconds, MSSQL increases this value to 34 seconds. It thus appears that MSSQL is responsible for the clustering visible in Figure 6.2(c). The selected database also affects response times. For instance, Neo4j is among the slowest across all response times, except for `getall`.

Contrarily, Couchbase is among the fastest across all response times, but the slowest for `getall`. As for energy indicators, Couchbase exhibits an outlying behavior: its power usage is higher in idle (13,59 W in total) than under load (13,51 W in total). This behavior is due to the design of Couchbase, which performs opportunistic indexation in idle to optimize response times under load. All other databases have idle power usage between 0.5 W and 2.4 W, except PostgreSQL with 0.27 W, and load power usage between 4.9 W and 6.8 W, except for Cassandra with 7 W.

Hibernate cache JHipster supports the activation of Hibernate cache for SQL databases. This analysis only accounts for configurations where Hibernate cache is available and activated, while configurations where the Hibernate cache is available and disabled serve as a baseline. Activating the Hibernate cache appears to increase the boot time by 12% and the response time of `create` and `delete` requests by 3.5%. The response time for `getall` is increased by 71%. Such overhead is leveraged in reading operations: the response times for `get` and `authenticated` are reduced by 4% and 29%, respectively. While activating the Hibernate cache substantially increases the idle power usage (+26%), it has a limited impact on the power usage under load (+3%).

Cache provider When the Hibernate cache is enabled, a cache provider must be selected among the five offered by JHipster. In Table 6.3, the "No cache" line refers to configurations where a cache system was available, but was not selected and thus acts as a baseline. The choice of cache system affects static indicators. Specifically, Redis increases the service count and the `size` of configurations. Infinispan and Hazelcast increase the boot time by 26% and 24%, respectively, whereas other providers increase such boot time by less than 5%. The choice of the cache system also impacts the response times. Some operations are consistently affected. For instance, response times for `authenticated` are reduced by up to 35%, and `create` are increased by up to 7%.

However, some cache systems have outlying behavior. For instance, Hazelcast and Redis increase `getall` by respectively 106% and 220% while other systems only increase it by 10%. Similarly, Caffeine and Ehcache do not affect `delete` response times while other systems increase it by up 16%. Ehcache appears to have the best performance across response times, while Caffeine is a close second. Contrarily, Redis appears to provide the worst

performance gains. Finally, the choice of cache systems also impacts the energy efficiency of the stack. Hazelcast and Redis substantially increase the idle power usage, by 96% and 25%, respectively Under load, such a difference is reduced to $+13\%$ and $+10\%$, respectively. Ehcache and Infinispan increase the idle power usage by 5% and 8%, respectively, but reduce the power usage under load by -3% and -2%. Finally, Caffeine reduces the power usage by 3%, both in idle and under load. Overall, configurations containing Ehcache or Caffeine exhibit the lowest power usage, both idle and under load.

Search engine JHipster supports Elasticsearch or Couchbase as search engines (S. E.), providing support to search content in the database. Each search engine is compared to its respective baseline of configurations that could contain them but do not. Configurations with CouchbaseSE are thus compared to configurations with Couchbase and not CouchbaseSE, while configurations with Elasticsearch are compared to configurations without Elasticsearch, Couchbase, or Cassandra. CouchbaseSE is embedded in Couchbase, and thus does not affect `size` and `boot time`, whereas Elasticsearch increases them by 65% and 8%, respectively. Thus, Elasticsearch can also be partially responsible for the spread in `size` visible in Figure 6.2(b). While both search engines have a limited impact on response times, Elasticsearch increases the response time of `create` by 97% and `delete` by 187%. Finally, enabling the Couchbase search engine has only a limited impact on power usage: $+1\%$ in idle and $+4\%$ under load. Contrarily, enabling Elasticsearch substantially increases power usage, by 41% in idle and 20% under load.

Reactive The Reactive option enables reactive programming for the application. Reactive allows for creating an asynchronous and non-blocking *Application Programming Interface* (API), in order to improve the scalability of the system. Reactive has a limited impact on static indicators, by reducing the boot time by 10%. However, it substantially increases the response times. In particular, `getall` and `get` are increased by 102% and 87%, respectively. This difference is more limited for `create`, 46%. Only `delete` is unaltered. Furthermore, reactive reduces the power usage in idle by 6%, but increases the power usage under load by 19%.

Answer to RQ 2 The available options have different impacts on the performance of the generated system. This impact is not consistent across indicators: the same option can have a negligible impact on a given indicator, while substantially affecting another one. Thus, an informed feature selection per indicator is needed to achieve efficient configurations tailored to specific requirements.

6.3.2 Performance of Default Configuration

In JHipster, the default configuration is the one generated when selecting the default option for each prompt in the JHipster *Command Line Interface* (CLI) tool. This configuration includes features like JWT, PostgreSQL, and Ehcache, while the search engine and the reactive database are disabled. As the default configuration, it may be the most used configuration of JHipster [133].

Table 6.4 Ranks and percentage rank of the default configuration and candidates configurations *wrt.* all valid configurations.

	Default		**Candidate 1**		**Candidate 2**		**Candidate 3**		**Candidate 4**	
	Rank	**Percentage**	**Rank**	**Percentage**	**Rank**	**Percentage**	**Rank**	**Percentage**	**Rank**	**Percentage**
size	6	5.1	**23**	**19.5**	**9**	**7.6**	**3**	**2.5**	**9**	**7.6**
services	1	0.8	1	0.8	1	0.8	1	0.8	1	0.8
boot-time	24	20.3	**3**	**2.5**	**21**	**17.8**	**22**	**18.6**	**23**	**19.5**
features	33	28.0	**1**	**0.8**	33	28	33	28.0	33	28.0
authentication	1	0.8	**6**	**5.1**	1	0.8	1	0.8	1	0.8
authenticated	1	0.8	**39**	**33.1**	1	0.8	1	0.8	1	0.8
getall	12	10.2	**1**	**0.8**	12	10.2	12	10.2	12	10.2
create	9	7.6	**1**	**0.8**	9	7.6	9	7.6	9	7.6
get	6	5.1	6	5.1	6	5.1	6	5.1	6	5.1
delete	9	7.6	**7**	**5.9**	9	7.6	9	7.6	9	7.6
idle-total	17	14.4	**10**	**8.5**	**8**	**6.8**	**12**	**10.2**	**18**	**15.3**
wl-total	7	5.9	**17**	**14.4**	**9**	**7.6**	**5**	**4.2**	**1**	**0.8**
Average	10.5	8.9	9.6	8.1	9.9	8.4	9.5	8	10.3	8.7

To compare this with all other valid configurations, their overall performance is aggregated into a single value. This value is calculated as the average of the configuration's percentile across all monitored performance indicators, where each indicator is given equal weight in determining the overall performance.

The ranks of the default configuration regarding each indicator are detailed in Table 6.4. When evaluating the overall rank of each configuration, with all metrics equally weighted, the default configuration performs well compared to other valid configurations. Its average percentage rank is 8.9%, indicating that, on average, it outperforms 91.1% of all other configurations across the monitored indicators. However, four configurations achieve better overall ranks, making them potential candidates for improving the default configuration's performance. These candidate configurations share similarities with the current default configuration, as shown in Table 6.4. The first candidate uses MongoDB instead of PostgreSQL and thus has no cache system, while candidates 2, 3, and 4 replace PostgreSQL with MariaDB, Ehcache with Caffeine, or both.

Candidate 1 exhibits worse ranking in terms of `size`, `authenticated` response time, and `wl-total` power usage. However, these regressions are outweighed by better ranks in 6 other indicators, resulting in an average percentage rank of 8.1%. Candidates 2, 3, and 4 are

closer to the default configuration and only affect the `size`, `boot-time`, `idle-total`, and `wl-total` indicators. All three configurations improve `boot-time`. Additionally, candidate 2 improves `idle-total` but worsens `size` and `wl-total`, while candidate 4 improves `wl-total` and worsens `size` and `idle-total`. Candidate 3 stands out by improving all four indicators without any regressions, allowing it to achieve an average percentage rank of 8%. It thus make it the top-ranked configuration in our evaluation and therefore, the best candidate for replacing the default configuration. Specifically, this improvement would be implemented by only replacing Ehcache with Caffeine. This results in a significant reduction of `idle-total` by 20.9%, while the benefits of this replacement are more modest regarding `wl-total`, `size`, and `boot-time`, by respectively 1.2%, 0.1% and 0.7% (other indicators remain unaffected). While this optimization aimed to maximize a single objective (optimizing all 12 indicators equally) it can also be adapted to focus on different sets of indicators or assign varying weights to each, allowing for the maximization of specific performance profiles. This approach can be applied not only to the default configuration but also to any custom configuration defined by the practitioner. For instance, when optimizing specifically for `boot-time`, `getall`, and `wl-total`, the default configuration achieves an average percentage rank of 20% for such indicators but is outperformed by 10 configurations. The best-performing candidate configuration has an average percentage rank of 10% and outperforms the default configuration with improvements of 2.1%, 14.3%, and 7.4% in `boot-time`, `getall`, and `wl-total`, respectively.

Answer to RQ 3 The default configuration of JHipster offers strong performances as a balanced configuration. However, when comparing its average ranking across the considered indicators, it is outperformed by several other candidates. This approach can be generalized to optimize the default configuration for any chosen set of indicators and shows that it can be more relevant to select configurations based on indicators to optimize rather than the preferred technologies.

6.4 Discussion and Threats to Validity

In this section, we discuss the lessons learned from using JHipster as a case study and outline the potential issues related to the validity of our work.

6.4.1 JHipster as Case Study

This experiment investigates the suitability of JHipster as a use case for research on the performance of configurable systems at the component level. JHipster offers several features that proved useful in this experiment, particularly its toolchain, which simplifies converting configurations files into running systems with little technical overhead. Additionally, the built-in testing scenario generation, including performance tests via Gatling, allowed for an effective evaluation of the system's usability and performance. However, different configuration options offer distinct features, meaning not all configurations are functionally equivalent. Specifically, search engines offer an additional API endpoint for content lookups, which was not evaluated in this experiment.

Thus, the metrics reported in this work reflect the impact of each option on the overall system performance, rather than their individual performance for their respective responsibility. Such an experimental approach would require designing configuration-specific test scenarios based on the options they contain. However, the simplicity of JHipster's tool chain does not apply to all configuration options. For instance, replacing the default docker-compose option with more complex alternatives such as micro-services or Kubernetes requires additional technical effort. Such alternatives were not explored in this experiment, which reduces the considered configuration space. Thus, no performance data was produced for these options, leaving their potential impact on overall system performance unknown.

In addition to these limitations, certain options or pairs of options were not considered in this experiment. The Cassandra database system is not compatible with Elasticsearch due to errors in the code generation system. Similarly, Couchbase, Neo4J, and Reactive cannot be combined with the JWT option, and Reactive is incompatible with configurations containing Cassandra, Neo4j, Session, or a cache mechanism. The Oracle database was excluded due to licensing restrictions, and community-developed options such as GRPC and JavaEE were excluded as they are not officially supported. As JHipster lacks a comprehensive feature model, such constraints were discovered either through documentation or trial and error. While some models, such as the one used in this work, or by Halin *et al.* [44] are available, the JHipster community would benefit from an official and exhaustive feature model.

6.4.2 Threats to Validity

The results of this study cannot be generalized as a ground truth about the overall performance of each analyzed services, as the configurations were assessed on a specific, synthetic workload automatically generated by JHipster during the creation of the project. This may

affect the applicability of our findings to other contexts and workloads, in particular regarding the performance of individual options discussed in Section 6.3.

Moreover, the performance model for each option was derived from this specific workload, aiming to explore the potential for inferring high-performance configurations for a given workload and performance indicator. This approach may limit how our results apply to other data models [75]. Additionally, all services were hosted on a single device, which may not accurately reflect production environments, especially in terms of response times and power usage. To address these limitations, future work will expand the scope by evaluating additional usage scenarios and adding additional execution contexts in the feature model.

6.5 Chapter Conclusion

This chapter presents an empirical analysis of the variability and performance of the configurable stack generator JHipster, demonstrating its relevance as a use case for variability analysis. The study shows how JHipster's diverse configurations significantly impact performance across various indicators, with strong correlations observed among similar indicators, enabling proxy-based optimization. Despite weaker correlations between different types of indicators, it is still possible to predict them by leveraging a combination of proxy indicators rather than relying on a single one. This approach thus enables the estimation of complex-to-measure indicators using more easily obtainable ones. Performance variations are mainly driven by specific configuration options, which may affect indicators unevenly-an option with minimal impact on one may significantly influence another.

Chapter 7

Conclusion and Perspectives

In this chapter, we summarize the contributions of our thesis and we discuss our short and long-term perspectives.

7.1 Summary of Contributions

In this thesis, we made three contributions that directly address the research question: *How can we reconcile performance with efficient use of hardware resources in the context of highly configurable software services?*.

In **Chapter 4**, we presented an empirical study focusing on two *Object-Relational Mapping* (ORM) frameworks and their configuration options to assess whether they constitute energy-efficient solutions. Across a range of workloads based on the *Transaction Processing Performance Council benchmark C* (TPC-C), we measured the energy overhead introduced by each ORM compared to a data access strategy that does not use ORM abstractions. EclipseLink proved to be the most energy-efficient ORM, while fine-tuning the configuration of Hibernate produced measurable effects on both performance and energy draw, illustrating how careful configuration can balance the two objectives.

In **Chapter 5**, we examined how Java web-application frameworks, deployments settings, and compilation strategies influence performance and resource efficiency. An empirical evaluation of Micronaut, Quarkus and Spring revealed that framework selection influences overall system efficiency, but other factors, such as how database connections are managed, can be equally or even more influential. We also compared compilation approaches, finding that *Ahead-Of-Time* (AOT) compilation consistently reduces memory usage across all frameworks while incurring a modest increase in energy consumption and a slight throughput penalty.

In **Chapter 6**, we explored both the relationships between different performance indicators and how different configuration options affect the performance of deployed systems. Using 118 distinct configurations from the JHipster tool, used as a case study, we obtained a detailed picture of how individual options influence performance metrics. The results indicated that the configurations exhibit distinct performance characteristics. highlighting the importance of informed choices in selecting the configuration to use. Furthermore, the results showed that performance indicators do not necessarily correlate, but combining multiple indicators may serve as a proxy to simplify software evaluation.

Together, these three contribution demonstrated how software abstractions, configuration parameters, and compilation strategies can be leveraged to reconcile performance with optimal use of hardware resource.

7.2 Perspectives

7.2.1 Short-Term Perspective

In this section we present short-term perspectives that emerges from this thesis.

Understanding Energy Transfer Between Distributed Systems

As we observed in Chapter 5, the energy consumption induced by distributed software services may evolve in an interdependent manner. For instance, replacing an application framework may reduce the power draw of a server hosting a web service while simultaneously increasing the consumption of a second server that runs another service (e.g., a database system) with which data are exchanged. In this context, an energy transfer can be defined as the consequence of reducing the power drain of one server by optimizing the efficiency of a software application running on it, yet simultaneously increasing the power usage of another server hosting a distinct application involved in distributed communications. The key research challenge is to understand how each system can achieve optimal energy usage without compromising overall service quality. To tackle this challenge, it will be important to consider all hardware components, such as processors, network interfaces, and storage devices, since each of them is driven in different ways depending on application characteristics and configurations.

Sustainable Source-Code Architectural Patterns

Over the past decades, numerous architectural patterns have been proposed to structure and manage the source code of software projects [2, 72]. A classic example is the three-tier architecture, whose logical layers—presentation, business logic, and data-persistence—directly reflect the physical three-tier deployment: a client machine interacts with an intermediate server, which in turn communicates with a database machine to deliver services.

More recent approaches include the hexagonal (*a.k.a.* Ports & Adapters) architecture and its variants (e.g., Clean Architecture [72]), which promote decoupling the core business rules (the domain) from technical concerns by defining simple interface definitions (ports) that the domain layer depends on, and by providing interchangeable concrete implementations (adapters). This approach facilitates the replacement of technical dependencies without affecting business logic. Although such styles promote a clean separation of concerns that could be beneficial for transitioning to more efficient libraries, they also introduce additional indirection, which can increase the cognitive load on developers, make navigation across a large code base more difficult, and potentially obscure performance-critical paths.

Consequently, exploring how source-code architectural patterns contribute to building sustainable software represents a valuable research direction. To this end, we have identified at least three aspects that can be examined.

The first aspect concerns the direct impact on runtime resources: a particular architectural decomposition may affect performance, memory usage, and energy consumption when the system is exercised with typical workloads. Architectural patterns often rely on language mechanisms such as object-oriented polymorphism or dynamic dispatch, and these mechanisms can introduce measurable runtime overhead that manifests as higher CPU utilisation, increased memory usage, and larger energy draws.

The second aspect addresses the cost and feasibility of dependency replacement. It asks how much effort is required to substitute a library or external service with a more efficient alternative, and how the chosen architecture either amplifies or mitigates that effort. An architecture that isolates technical concerns behind well-defined interfaces can make swapping implementations relatively straightforward, whereas tightly-coupled designs may force extensive code changes and testing, thereby raising the overall migration cost.

The third aspect relates to the ease of constructing lightweight software variants. It examines how difficult it is to extract a subset of an application's features in order to build a lightweight version that implements only the most frequently used functionality. A concrete illustration of this direction is the *Vertical Slice Architecture* (VSA) [107], which promotes structuring the system by features rather than by technical layers. Using VSA, each business

operation is implemented in its own "slice" that encompasses all classical layers (presentation, business logic, and data access) in contrast to the three-tier architecture.

Considering the trade-offs introduced by each architecture, the results of such research effort could enable the creation of a comprehensive map of the architecture landscape.

7.2.2 Long-Term Perspectives

In this section, we present the long-term perspectives of this thesis.

Towards Green Software Supply Chains: Sustainable Development via Enhanced Software Bills of Materials

A software supply chain comprises the components, tools, and processes used to develop, build, and deliver a software artifact. In this context, *Software Bills of Materials* (SBOMs) are manifests that enumerate every component, library, and dependency along with relevant metadata such as version and provenance, thereby enhancing traceability, vulnerability detection, and regulatory compliance [4, 90, 37]. Building on these insights, a promising research direction is to investigate how *Software Bills of Materials* (SBOMs) can be leveraged to enhance the sustainability of software applications. In particular, we believe that SBOMs can be used like repairability index[111] for physical goods, to communicate the engineering effort a development team has invested in improving software products over time.

Configuration Recommendation System for Sustainable Software Development

In this thesis we have shown that software configuration spans multiple layers of the stack, from runtime environments to library-level options.

At the same time, exhaustively enumerating and testing every possible configuration is impractical, a source of resource waste, and often misleading, because a setting that is optimal for the current requirements can quickly become suboptimal as those requirements evolve. Moreover, many configuration options are poorly documented or difficult to discover, especially in legacy code bases.

Consequently, it would be valuable to investigate how a configuration recommendation system can be developed to guide developers in *(i)* identifying the parameters that maximize sustainability objectives and *(ii)* estimating the potential trade-offs of applying the proposed configuration to reduce unnecessary deployments.

References

[1] B. Acun, B. Lee, F. Kazhamiaka, K. Maeng, U. Gupta, M. Chakkaravarthy, D. Brooks, and C.-J. Wu. Carbon explorer: A holistic framework for designing carbon aware datacenters. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, page 118–132, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9781450399166. doi: 10.1145/3575693.3575754. URL https://doi.org/10.1145/3575693.3575754.

[2] M. Aniche, G. Bavota, C. Treude, A. Van Deursen, and M. A. Gerosa. A Validated Set of Smells in Model-View-Controller Architectures . In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 233–243, Los Alamitos, CA, USA, Oct. 2016. IEEE Computer Society. doi: 10.1109/ICSME.2016.12. URL https://doi.ieeecomputersociety.org/10.1109/ICSME.2016.12.

[3] P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Feral concurrency control: An empirical investigation of modern application integrity. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, page 1327–1342, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450327589. doi: 10.1145/2723372.2737784. URL https://doi.org/10.1145/2723372.2737784.

[4] M. Balliu, B. Baudry, S. Bobadilla, M. Ekstedt, M. Monperrus, J. Ron, A. Sharma, G. Skoglund, C. Soto-Valero, and M. Wittlinger. Challenges of Producing Software Bill of Materials for Java . *IEEE Security & Privacy*, 21(06):12–23, Nov. 2023. ISSN 1558-4046. doi: 10.1109/MSEC.2023.3302956. URL https://doi.ieeecomputersociety.org/10.1109/MSEC.2023.3302956.

[5] E. Barrett, C. F. Bolz-Tereick, R. Killick, S. Mount, and L. Tratt. Virtual machine warmup blows hot and cold. *Proc. ACM Program. Lang.*, 1(OOPSLA), oct 2017. doi: 10.1145/3133876. URL https://doi.org/10.1145/3133876.

[6] A. Barrière, S. Blazy, O. Flückiger, D. Pichardie, and J. Vitek. Formally verified speculation and deoptimization in a jit compiler. *Proc. ACM Program. Lang.*, 5(POPL), 2021. doi: 10.1145/3434327.

[7] S. A. Baset, L. Wang, and C. Tang. Towards an understanding of oversubscription in cloud. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services*, Hot-ICE'12, page 7, USA, 2012. USENIX Association.

[8] BenchBase a Multi-DBMS SQL Benchmarking Framework. Benchbase a multi-dbms sql benchmarking framework, 2025. https://db.cs.cmu.edu/projects/benchbase/ [Accessed: Apr. 22, 2025].

[9] A. Bonvoisin, C. Quinton, and R. Rouvoy. Understanding the performance-energy trade-offs of orm frameworks - data and materials package, 2024. URL https://doi.org/10.5281/zenodo.10061193.

[10] A. Bonvoisin, V. Amand, C. Quinton, and R. Rouvoy. Optimizing performance and resource efficiency in java-based cloud native services - materials and data package, 2025. URL https://doi.org/10.5281/zenodo.17861521.

[11] Broadcom Inc. and/or its subsidiaries. Spring framework. https://spring.io/, 2025. Accessed: Apr. 22, 2025.

[12] Broadleaf Commerce, LLC. Broadleaf commerce community edition - github repository, 2010. URL https://github.com/BroadleafCommerce/BroadleafCommerce.

[13] C. Calero, M. Polo, and M. Ángeles Moraga. Investigating the impact on execution time and energy consumption of developing with spring. *Sustainable Computing: Informatics and Systems*, 32:100603, 2021. ISSN 2210-5379. doi: https://doi.org/10.1016/j.suscom.2021.100603. URL https://www.sciencedirect.com/science/article/pii/S2210537921000913.

[14] Z. Cao, X. Zhou, H. Hu, Z. Wang, and Y. Wen. Toward a Systematic Survey for Carbon Neutral Data Centers. *IEEE Communications Surveys & Tutorials*, 24(2): 895–936, 2022. ISSN 1553-877X. doi: 10.1109/COMST.2022.3161275. URL https://ieeexplore.ieee.org/abstract/document/9739671.

[15] F. Cappello, E. Caron, M. Dayde, F. Desprez, Y. Jegou, P. Primet, E. Jeannot, S. Lanteri, J. Leduc, N. Melab, G. Mornet, R. Namyst, B. Quetier, and O. Richard. Grid'5000: a large scale and highly reconfigurable grid experimental testbed. In *The 6th IEEE/ACM International Workshop on Grid Computing, 2005.*, pages 8 pp.–, Seattle, WA, USA, 2005. IEEE. doi: 10.1109/GRID.2005.1542730.

[16] R. Chattaraj and S. Chimalakonda. Rjoules: An energy measurement tool for r. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 2026–2029, 2023. doi: 10.1109/ASE56229.2023.00207.

[17] T.-H. Chen, W. Shang, Z. M. Jiang, A. E. Hassan, M. Nasser, and P. Flora. Detecting performance anti-patterns for applications developed using object-relational mapping. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, page 1001–1012. ACM, 2014. ISBN 9781450327565. doi: 10.1145/2568225.2568259. URL https://doi.org/10.1145/2568225.2568259.

[18] T.-H. Chen, W. Shang, A. E. Hassan, M. Nasser, and P. Flora. Cacheoptimizer: helping developers configure caching frameworks for hibernate-based database-centric web applications. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, page 666–677, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342186. doi: 10.1145/2950290.2950303. URL https://doi.org/10.1145/2950290.2950303.

[19] T.-H. Chen, W. Shang, A. E. Hassan, M. Nasser, and P. Flora. Detecting problems in the database access code of large scale systems - an industrial experience report. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, pages 71–80, 2016.

[20] T.-H. Chen, W. Shang, Z. M. Jiang, A. E. Hassan, M. Nasser, and P. Flora. Finding and evaluating the performance impact of redundant data access for applications that are developed using object-relational mapping frameworks. *IEEE Transactions on Software Engineering*, 42(12):1148–1161, 2016. doi: 10.1109/TSE.2016.2553039.

[21] Civilized Discourse Construction Kit, Inc. Discourse - github repository, 2014. URL https://github.com/discourse/discourse.

[22] D. Connolly Bree and M. Ó Cinnéide. Energy efficiency of the visitor pattern: contrasting java and c++ implementations. *Empirical Software Engineering*, 28 (6):145, Oct 2023. ISSN 1573-7616. doi: 10.1007/s10664-023-10387-8. URL https://doi.org/10.1007/s10664-023-10387-8.

[23] Control Group v2 Documentation. Control group v2 documentation, 2025. https://www.kernel.org/doc/Documentation/cgroup-v2.txt [Accessed: Apr. 22, 2025].

[24] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, page 143–154, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781450300360. doi: 10.1145/1807128.1807152. URL https://doi.org/10.1145/1807128.1807152.

[25] B. Danglot, J.-R. Falleri, and R. Rouvoy. Can we spot energy regressions using developers tests? *Empirical Software Engineering*, 29(5):121, 2024. ISSN 1573-7616. doi: 10.1007/s10664-023-10429-1. URL https://doi.org/10.1007/s10664-023-10429-1.

[26] H. David, E. Gorbatov, U. R. Hanebutte, R. Khanna, and C. Le. Rapl: memory power estimation and capping. In *Proceedings of the 16th ACM/IEEE International Symposium on Low Power ElectCarbon Explorer: A Holistic Framework for Designing Carbon Aware Datacentersronics and Design*, ISLPED '10, page 189–194, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781450301466. doi: 10.1145/1840845.1840883. URL https://doi.org/10.1145/1840845.1840883.

[27] S. Delamare and L. Nussbaum. Kwollect: Metrics collection for experiments at scale. In *IEEE INFOCOM 2021 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 1–6, 2021. doi: 10.1109/INFOCOMWKSHPS51825.2021.9484540.

[28] S. Desrochers, C. Paradis, and V. M. Weaver. A validation of dram rapl power measurements. In *Proceedings of the Second International Symposium on Memory Systems*, MEMSYS '16, page 455–470. ACM, 2016. ISBN 9781450343053. doi: 10.1145/2989081.2989088. URL https://doi.org/10.1145/2989081.2989088.

[29] D. E. Difallah, A. Pavlo, C. Curino, and P. Cudré-Mauroux. Oltp-bench: an extensible testbed for benchmarking relational databases. *Proc. VLDB Endow.*, 7(4):277–288, dec 2013. ISSN 2150-8097. doi: 10.14778/2732240.2732246. URL http://www.vldb.org/pvldb/vol7/p277-difallah.pdf.

[30] M. E. M. Diouri, M. F. Dolz, O. Glück, L. Lefèvre, P. Alonso, S. Catalán, R. Mayo, and E. S. Quintana-Ortí. Assessing power monitoring approaches for energy and power analysis of computers. *Sustainable Computing: Informatics and Systems*, 4(2): 68–82, 2014. ISSN 2210-5379. doi: https://doi.org/10.1016/j.suscom.2014.03.006. URL https://www.sciencedirect.com/science/article/pii/S2210537914000171. Special Issue on Selected papers from EE-LSDS2013 Conference.

[31] Docker. Docker, 2025. https://www.docker.com/ [Accessed: Apr. 22, 2025].

[32] Ebean ORM. Ebean orm, 2025. https://ebean.io/ [Accessed: Apr. 22, 2025].

[33] EclipseLink. Eclipselink, 2025. https://eclipse.dev/eclipselink/ [Accessed: Apr. 22, 2025].

[34] Eugen Rochko & other Mastodon contributors. Mastodon - github repository, 2016. URL https://github.com/mastodon/mastodon.

[35] M. Fayad and D. C. Schmidt. Object-oriented application frameworks. *Commun. ACM*, 40(10):32–38, Oct. 1997. ISSN 0001-0782. doi: 10.1145/262793.262798. URL https://doi.org/10.1145/262793.262798.

[36] G. Fieni, R. Rouvoy, and L. Seinturier. SmartWatts: Self-Calibrating Software-Defined Power Meter for Containers. In *CCGRID 2020 - 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing*, May 2020. doi: 10.1109/CCGrid49817.2020.00-45. URL https://hal.inria.fr/hal-02470128.

[37] Y. Gamage, N. G. Fernandez, M. Monperrus, and B. Baudry. Software Bills of Materials in Maven Central . In *2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR)*, pages 339–343, Los Alamitos, CA, USA, Apr. 2025. IEEE Computer Society. doi: 10.1109/MSR66628.2025.00062. URL https://doi.ieeecomputersociety.org/10.1109/MSR66628.2025.00062.

[38] S. Georgiou, M. Kechagia, P. Louridas, and D. Spinellis. What are your programming language's energy-delay implications? In *Proceedings of the 15th International Conference on Mining Software Repositories*, MSR '18, page 303–313, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450357166. doi: 10.1145/3196398.3196414. URL https://doi.org/10.1145/3196398.3196414.

[39] E. Guégain, C. Quinton, and R. Rouvoy. On reducing the energy consumption of software product lines. In *Proceedings of the 25th ACM International Systems and Software Product Line Conference - Volume A*, SPLC '21, page 89–99. ACM, 2021. ISBN 9781450384698. doi: 10.1145/3461001.3471142. URL https://doi.org/10.1145/3461001.3471142.

[40] E. Guégain, A. Taherkordi, and C. Quinton. Configuration optimization with limited functional impact. In *Advanced Information Systems Engineering: 35th International Conference, CAiSE 2023, Zaragoza, Spain, June 12–16, 2023, Proceedings*, page 53–68, Cham, 2023. Springer Nature Switzerland. ISBN 978-3-031-34559-3. doi: 10.1007/978-3-031-34560-9_4. URL https://doi.org/10.1007/978-3-031-34560-9_4.

[41] E. Guégain, A. Bonvoisin, C. Mathieu, Acher Quinton, and R. Rouvoy. Exploring performance of configurable software systems: the jhipster case study - replication package, 2025. URL https://doi.org/10.5281/zenodo.8140599.

[42] J. Guo, Z. Chang, S. Wang, H. Ding, Y. Feng, L. Mao, and Y. Bao. Who limits the resource efficiency of my datacenter: An analysis of alibaba datacenter traces. In *2019 IEEE/ACM 27th International Symposium on Quality of Service (IWQoS)*, pages 1–10, Phoenix, AZ, USA, 2019. IEEE. doi: 10.1145/3326285.3329074.

[43] U. Gupta, Y. G. Kim, S. Lee, J. Tse, H.-H. S. Lee, G.-Y. Wei, D. Brooks, and C.-J. Wu. Chasing Carbon: The Elusive Environmental Footprint of Computing. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 854–867, Feb. 2021. doi: 10.1109/HPCA51647.2021.00076. URL https://ieeexplore.ieee.org/document/9407142. ISSN: 2378-203X.

[44] A. Halin, A. Nuttinck, M. Acher, X. Devroey, G. Perrouin, and P. Heymans. Yo variability! jhipster: a playground for web-apps analyses. In *Proceedings of the 11th International Workshop on Variability Modelling of Software-Intensive Systems*, VaMoS '17, page 44–51, 2017. ISBN 9781450348119. doi: 10.1145/3023956.3023963. URL https://doi.org/10.1145/3023956.3023963.

[45] A. Halin, A. Nuttinck, M. Acher, X. Devroey, G. Perrouin, and B. Baudry. Test them all, is it worth it? assessing configuration sampling on the jhipster web development stack. *Empirical Software Engineering*, 24:674–717, 2019.

[46] S. Hasan, Z. King, M. Hafiz, M. Sayagh, B. Adams, and A. Hindle. Energy profiles of java collections classes. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 225–236, 2016. doi: 10.1145/2884781.2884869.

[47] Hibernate ORM. Hibernate orm, 2025. https://hibernate.org/orm/ [Accessed: Apr. 22, 2025].

[48] R. M. Hierons, M. Li, X. Liu, S. Segura, and W. Zheng. Sip: Optimal product selection from feature models using many-objective evolutionary optimization. *ACM Trans. Softw. Eng. Methodol.*, 25(2), 2016. ISSN 1049-331X.

[49] R. Hutcheson, A. Blanchard, N. Lambaria, J. Hale, D. Kozak, A. S. Abdelfattah, and T. Cerny. Software architecture reconstruction for microservice systems using static analysis via graalvm native image. In *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 12–22, Rovaniemi, Finland, 2024. IEEE. doi: 10.1109/SANER60148.2024.00008.

[50] P. Jacquet, T. Ledoux, and R. Rouvoy. SCROOGEVM: Boosting Cloud Resource Utilization With Dynamic Oversubscription . *IEEE Transactions on Sustainable Computing*, 9(05):754–765, Sept. 2024. ISSN 2377-3782. doi: 10.1109/TSUSC.2024.3369333. URL https://doi.ieeecomputersociety.org/10.1109/TSUSC.2024.3369333.

[51] Jakarta Persistence 3.1. Jakarta persistence 3.1, 2025. https://jakarta.ee/specifications/persistence/3.1/ [Accessed: Apr. 22, 2025].

[52] Jdbi. Jdbi, 2025. https://jdbi.org/ [Accessed: Apr. 22, 2025].

[53] JHipster Development Team. Jhipster - github repository, 2013. URL https://github.com/jhipster.

[54] JHipster Development Team. Jhipster: Full stack platform for the modern developer!, 2013. URL https://www.jhipster.tech/.

[55] R. Johnson and B. Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1:22–35, 06 1988.

[56] JOOQ. Jooq, 2025. https://www.jooq.org/ [Accessed: Apr. 22, 2025].

[57] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 1–12, New York, NY, USA, June 2017. Association for Computing Machinery. ISBN 978-1-4503-4892-8. doi: 10.1145/3079856.3080246. URL https://dl.acm.org/doi/10.1145/3079856.3080246.

[58] A. Katal, S. Dahiya, and T. Choudhury. Energy efficiency in cloud computing data center: a survey on hardware technologies. *Cluster Computing*, 25(1):675–705, Feb. 2022. ISSN 1573-7543. doi: 10.1007/s10586-021-03431-z. URL https://doi.org/10.1007/s10586-021-03431-z.

[59] A. Katal, S. Dahiya, and T. Choudhury. Energy efficiency in cloud computing data centers: a survey on software technologies. *Cluster Computing*, 26(3):1845–1875, June 2023. ISSN 1573-7543. doi: 10.1007/s10586-022-03713-0. URL https://doi.org/10.1007/s10586-022-03713-0.

[60] S. P. R. Katamreddy and S. S. Upadhyayula. *Working with JDBC*, pages 101–118. Apress, 2023. ISBN 978-1-4842-8792-7. doi: 10.1007/978-1-4842-8792-7_5. URL https://doi.org/10.1007/978-1-4842-8792-7_5.

[61] K. N. Khan, M. Hirki, T. Niemi, J. K. Nurminen, and Z. Ou. Rapl in action: Experiences in using rapl for power measurements. *ACM Trans. Model. Perform. Eval. Comput. Syst.*, 3(2), mar 2018. ISSN 2376-3639. doi: 10.1145/3177754. URL https://doi.org/10.1145/3177754.

[62] M. Koot and F. Wijnhoven. Usage impact on data center electricity needs: A system dynamic forecasting model. *Applied Energy*, 291:116798, June 2021. ISSN 0306-2619. doi: 10.1016/j.apenergy.2021.116798. URL https://www.sciencedirect.com/science/article/pii/S0306261921003019.

[63] D. Kossmann, T. Kraska, and S. Loesing. An evaluation of alternative architectures for transaction processing in the cloud. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, page 579–590, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781450300322. doi: 10.1145/1807167.1807231. URL https://doi.org/10.1145/1807167.1807231.

[64] M. Kumar, Y. Li, and W. Shi. Energy consumption in java: An early experience. In *2017 Eighth International Green and Sustainable Computing Conference (IGSC)*, pages 1–8, 2017. doi: 10.1109/IGCC.2017.8323579.

[65] A. G. Kumbhare, R. Azimi, I. Manousakis, A. Bonde, F. Frujeri, N. Mahalingam, P. A. Misra, S. A. Javadi, B. Schroeder, M. Fontoura, and R. Bianchini. Prediction-Based power oversubscription in cloud platforms. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 473–487. USENIX Association, July 2021. ISBN 978-1-939133-23-6. URL https://www.usenix.org/conference/atc21/presentation/kumbhare.

[66] M. LeBeane, J. H. Ryoo, R. Panda, and L. K. John. Watt watcher: Fine-grained power estimation for emerging workloads. In *2015 27th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 106–113. IEEE, 2015. doi: 10.1109/SBAC-PAD.2015.26.

[67] P. Leboulanger and A.-C. Orgerie. Power limits in data centers: what can we expect from improving energy efficiency and refreshing servers? . In *2025 IEEE International Conference on Cloud Engineering (IC2E)*, pages 114–125, Los Alamitos, CA, USA, Sept. 2025. IEEE Computer Society. doi: 10.1109/IC2E65552.2025.00027. URL https://doi.ieeecomputersociety.org/10.1109/IC2E65552.2025.00027.

[68] L. Lefèvre and A.-C. Orgerie. Designing and evaluating an energy efficient cloud. *J. Supercomput.*, 51(3):352–373, Mar. 2010. ISSN 0920-8542. doi: 10.1007/s11227-010-0414-2. URL https://doi.org/10.1007/s11227-010-0414-2.

[69] K. Liu, G. Pinto, and Y. D. Liu. Data-oriented characterization of application-level energy optimization. In A. Egyed and I. Schaefer, editors, *Fundamental Approaches to Software Engineering*, volume 9033, pages 316–331, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg. ISBN 978-3-662-46675-9. URL http://link.springer.com/10.1007/978-3-662-46675-9_21. Series Title: Lecture Notes in Computer Science.

[70] Mandrel source code repository. Mandrel source code repository, 2025. https://github.com/graalvm/mandrel [Accessed: Apr. 22, 2025].

[71] I. Manotas, C. Bird, R. Zhang, D. Shepherd, C. Jaspan, C. Sadowski, L. Pollock, and J. Clause. An empirical study of practitioners' perspectives on green software engineering. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, page 237–248, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450339001. doi: 10.1145/2884781.2884810. URL https://doi.org/10.1145/2884781.2884810.

[72] R. C. Martin. *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Prentice Hall Press, USA, 1st edition, 2017. ISBN 0134494164.

[73] S. D. Meglio and L. Libero Lucio Starace. Evaluating performance and resource consumption of rest frameworks and execution environments: Insights and guidelines for developers and companies. *IEEE Access*, 12:161649–161669, 2024. doi: 10.1109/ACCESS.2024.3489892.

[74] Micronaut Framework. Micronaut framework, 2025. https://micronaut.io/ [Accessed: Apr. 22, 2025].

[75] S. Muhlbauer, F. Sattler, C. Kaltenecker, J. Dorn, S. Apel, and N. Siegmund. Analysing the Impact of Workloads on Modeling the Performance of Configurable Software Systems . In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 2085–2097, 2023. doi: 10.1109/ICSE48619.2023.00176. URL https://doi.ieeecomputersociety.org/10.1109/ICSE48619.2023.00176.

[76] D.-J. Munoz, M. Pinto, and L. Fuentes. Quality-aware analysis and optimisation of virtual network functions. In *Proceedings of the 26th ACM International Systems and Software Product Line Conference - Volume A*, SPLC '22, page 210–221, 2022. ISBN 9781450394437.

[77] MyBatis. Mybatis, 2025. https://mybatis.org/mybatis-3/ [Accessed: Apr. 22, 2025].

[78] D. Mytton. Data centre water consumption. *npj Clean Water*, 4(1):11, Feb. 2021. ISSN 2059-7037. doi: 10.1038/s41545-021-00101-w. URL https://www.nature.com/articles/s41545-021-00101-w. Publisher: Nature Publishing Group.

[79] S. Naffziger, N. Beck, T. Burd, K. Lepak, G. H. Loh, M. Subramony, and S. White. Pioneering chiplet technology and design for the amd epyc™ and ryzen™ processor families. In *Proceedings of the 48th Annual International Symposium on Computer Architecture*, ISCA '21, page 57–70. IEEE Press, 2021. ISBN 9781450390866. doi: 10.1109/ISCA52012.2021.00014. URL https://doi.org/10.1109/ISCA52012.2021.00014.

[80] F. Nahrstedt, M. Karmouche, K. Bargieł, P. Banijamali, A. Nalini Pradeep Kumar, and I. Malavolta. An empirical study on the energy usage and performance of pandas and polars data analysis python libraries. In *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*, EASE '24, page 58–68, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400717017. doi: 10.1145/3661167.3661203. URL https://doi.org/10.1145/3661167.3661203.

[81] V. Nair, Z. Yu, T. Menzies, N. Siegmund, and S. Apel. Finding faster configurations using flash. *IEEE Transactions on Software Engineering*, 46(7):794–811, 2020.

[82] A. Navarro, J. Ponge, F. Le Mouël, and C. Escoffier. Considerations for integrating virtual threads in a java framework: a quarkus example in a resource-constrained environment. In *Proceedings of the 17th ACM International Conference on Distributed and Event-Based Systems*, DEBS '23, page 103–114, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400701221. doi: 10.1145/3583678.3596895. URL https://doi.org/10.1145/3583678.3596895.

[83] A. Noureddine. Powerjoular and joularjx: Multi-platform software power monitoring tools. In *2022 18th International Conference on Intelligent Environments (IE)*, pages 1–4, 2022. doi: 10.1109/IE54923.2022.9826760.

[84] A. Noureddine and O. Le Goaer. Investigating the Impact of Software Design Patterns on Energy Consumption . In *2025 IEEE 22nd International Conference on Software Architecture (ICSA)*, pages 153–163, Los Alamitos, CA, USA, Apr. 2025. IEEE Computer Society. doi: 10.1109/ICSA65012.2025.00024. URL https://doi.ieeecomputersociety.org/10.1109/ICSA65012.2025.00024.

[85] A. Noureddine and A. Rajan. Optimising energy consumption of design patterns. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 623–626, 2015. doi: 10.1109/ICSE.2015.208.

[86] R. Olaechea, S. Stewart, K. Czarnecki, and D. Rayside. Modelling and multi-objective optimization of quality attributes in variability-rich software. In *Proceedings of the Fourth International Workshop on Nonfunctional System Properties in Domain Specific Modeling Languages*, NFPinDSML '12, 2012. ISBN 9781450318075. doi: 10.1145/2420942.2420944. URL https://doi.org/10.1145/2420942.2420944.

[87] W. Oliveira, R. Oliveira, F. Castor, B. Fernandes, and G. Pinto. Recommending energy-efficient java collections. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 160–170, Montreal, QC, Canada, 2019. IEEE. doi: 10.1109/MSR.2019.00033.

[88] Oracle Corporation and/or its affiliates. Java platform, standard edition hotspot virtual machine garbage collection tuning guide, release 21. https://docs.oracle.com/en/java/javase/21/gctuning/introduction-garbage-collection-tuning.html, 2025. HotSpot Virtual Machine Garbage Collection Tuning Guide, Accessed: Apr. 22, 2025.

[89] Oracle GraalVM for JDK 21 Documentation. Oracle GraalVM for JDK 21 documentation, 2025. https://docs.oracle.com/en/graalvm/jdk/21/docs/reference-manual/native-image/ [Accessed: Apr. 22, 2025].

[90] W. Otoda, T. Kanda, Y. Manabe, K. Inoue, and Y. Higo. SBOM Challenges for Developers: From Analysis of Stack Overflow Questions . In *2024 IEEE/ACIS 22nd International Conference on Software Engineering Research, Management and Applications (SERA)*, pages 43–46, Los Alamitos, CA, USA, June 2024. IEEE Computer Society. doi: 10.1109/SERA61261.2024.10685624. URL https://doi.ieeecomputersociety.org/10.1109/SERA61261.2024.10685624.

[91] Z. Ournani, R. Rouvoy, P. Rust, and J. Penhoat. On reducing the energy consumption of software: From hurdles to requirements. In *Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, ESEM '20, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450375801. doi: 10.1145/3382494.3410678. URL https://doi.org/10.1145/3382494.3410678.

[92] Z. Ournani, M. C. Belgaid, R. Rouvoy, P. Rust, and J. Penhoat. Evaluating the impact of java virtual machines on energy consumption. In *Proceedings of the 15th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, ESEM '21, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450386654. doi: 10.1145/3475716.3475774. URL https://doi.org/10.1145/3475716.3475774.

[93] Z. Ournani, R. Rouvoy, P. Rust, and J. Penhoat. Evaluating The Energy Consumption of Java I/O APIs. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 1–11, Luxembourg, Sept. 2021. IEEE. ISBN 978-1-66542-882-8. doi: 10.1109/ICSME52107.2021.00007. URL https://ieeexplore.ieee.org/document/9609210/.

[94] Z. Ournani, R. Rouvoy, P. Rust, and J. Penhoat. Tales from the code #1: The effective impact of code refactorings on software energy consumption. In *ICSOFT*, pages 34–46. SCITEPRESS, 2021.

[95] Z. Ournani, R. Rouvoy, P. Rust, and J. Penhoat. Tales from the code #2: A detailed assessment of code refactoring's impact on energy consumption. In *ICSOFT (Selected Papers)*, volume 1622 of *Communications in Computer and Information Science*, pages 94–116. Springer, 2021.

[96] J. A. Pereira, M. Acher, H. Martin, J.-M. Jézéquel, G. Botterweck, and A. Ventresque. Learning software configuration spaces: A systematic literature review. *Journal of Systems and Software*, 182:111044, 2021.

[97] R. Pereira, M. Couto, J. a. Saraiva, J. Cunha, and J. a. P. Fernandes. The influence of the java collection framework on overall energy consumption. In *Proceedings of the 5th International Workshop on Green and Sustainable Software*, GREENS '16, page 15–21. ACM, 2016. ISBN 9781450341615. doi: 10.1145/2896967.2896968. URL https://doi.org/10.1145/2896967.2896968.

[98] R. Pereira, M. Couto, F. Ribeiro, R. Rua, J. Cunha, J. P. Fernandes, and J. Saraiva. Ranking programming languages by energy efficiency. *Science of Computer Programming*, 205:102609, 2021. ISSN 0167-6423. doi: https://doi.org/10.1016/j.scico.2021.102609. URL https://www.sciencedirect.com/science/article/pii/S0167642321000022.

[99] G. Pinto, F. Castor, and Y. D. Liu. Understanding energy behaviors of thread management constructs. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, page 345–360, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450325851. doi: 10.1145/2660193.2660235. URL https://doi.org/10.1145/2660193.2660235.

[100] G. Pinto, K. Liu, F. Castor, and Y. D. Liu. A comprehensive study on the energy efficiency of java's thread-safe collections. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 20–31, Raleigh, NC, USA, 2016. IEEE. doi: 10.1109/ICSME.2016.34.

[101] D. R. K. Ports and K. Grittner. Serializable snapshot isolation in postgresql. *Proc. VLDB Endow.*, 5(12):1850–1861, Aug. 2012. ISSN 2150-8097. doi: 10.14778/2367502.2367523. URL https://doi.org/10.14778/2367502.2367523.

[102] O. Poy, M. A. Moraga, F. García, and C. Calero. Impact on energy consumption of design patterns, code smells and refactoring techniques: A systematic mapping study. *Journal of Systems and Software*, 222:112303, 2025. ISSN 0164-1212. doi: 10.1016/j.jss.2024.112303. URL https://www.sciencedirect.com/science/article/pii/S0164121224003479.

[103] G. Procaccianti, P. Lago, and W. Diesveld. Energy efficiency of orm approaches: an empirical evaluation. In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, ESEM '16, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450344272. doi: 10.1145/2961111.2962586. URL https://doi.org/10.1145/2961111.2962586.

[104] Quarkus Framework. Quarkus framework, 2025. https://quarkus.io/ [Accessed: Apr. 22, 2025].

[105] A. Radovanović, R. Koningstein, I. Schneider, B. Chen, A. Duarte, B. Roy, D. Xiao, M. Haridasan, P. Hung, N. Care, S. Talukdar, E. Mullen, K. Smith, M. Cottman, and W. Cirne. Carbon-Aware Computing for Datacenters. *IEEE Transactions on Power Systems*, 38(2):1270–1280, Mar. 2023. ISSN 1558-0679. doi: 10.1109/TPWRS.2022.3173250. URL https://ieeexplore.ieee.org/abstract/document/9770383.

[106] P. Rani, J. Zellweger, V. Kousadianos, L. Cruz, T. Kehrer, and A. Bacchelli. Energy patterns for web: An exploratory study. In *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Society*, ICSE-SEIS'24, page 12–22, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400704994. doi: 10.1145/3639475.3640110. URL https://doi.org/10.1145/3639475.3640110.

[107] I. M. Ratner and J. Harvey. Vertical Slicing: Smaller is Better . In *2011 Agile Conference*, pages 240–245, Los Alamitos, CA, USA, Aug. 2011. IEEE Computer Society. doi: 10.1109/AGILE.2011.46. URL https://doi.ieeecomputersociety.org/10.1109/AGILE.2011.46.

[108] F. Rieger and C. Bockisch. Survey of approaches for assessing software energy consumption. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Comprehension of Complex Systems*, CoCoS 2017, page 19–24. ACM, 2017. ISBN 9781450355216. doi: 10.1145/3141842.3141846. URL https://doi.org/10.1145/3141842.3141846.

[109] G. Rocha, F. Castor, and G. Pinto. Comprehending Energy Behaviors of Java I/O APIs. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–12, 2019. doi: 10.1109/ESEM.2019.8870158.

[110] S. M. Rozehkhani, F. Mahan, and W. Pedrycz. Efficient cloud data center: An adaptive framework for dynamic Virtual Machine Consolidation. *Journal of Network and Computer Applications*, 226:103885, June 2024. ISSN 1084-8045. doi: 10.1016/j.jnca.2024.103885. URL https://www.sciencedirect.com/science/article/pii/S1084804524000626.

[111] L. Ruiz-Pastor and J. A. Mesa. Proposing an integrated indicator to measure product repairability. *Journal of Cleaner Production*, 395:136434, 2023. ISSN 0959-6526. doi: https://doi.org/10.1016/j.jclepro.2023.136434.

[112] E. A. Santos, C. McLean, C. Solinas, and A. Hindle. How does docker affect energy consumption? evaluating workloads in and out of docker containers. *Journal of Systems and Software*, 146:14–25, 2018. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2018.07.077.

[113] H. A. Schmid. Systematic framework design by generalization. *Commun. ACM*, 40(10):48–51, Oct. 1997. ISSN 0001-0782. doi: 10.1145/262793.262803. URL https://doi.org/10.1145/262793.262803.

[114] A. Shahoor, J. Yi, and D. Kim. Preserving reactiveness: Understanding and improving the debugging practice of blocking-call bugs. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2024, page 768–780, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400706127. doi: 10.1145/3650212.3680319. URL https://doi.org/10.1145/3650212.3680319.

[115] T. C. Shan and W. W. Hua. Taxonomy of java web application frameworks. In *Proceedings. IEEE International Conference on e-Business Engineering*, pages 378–385, 2006. doi: 10.1109/ICEBE.2006.98.

[116] S. Shanbhag and S. Chimalakonda. An exploratory study on energy consumption of dataframe processing libraries. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, pages 284–295, Melbourne, Australia, 2023. IEEE. doi: 10.1109/MSR59073.2023.00048.

[117] N. Siegmund, M. Rosenmüller, M. Kuhlemann, C. Kästner, S. Apel, and G. Saake. Spl conqueror: Toward optimization of non-functional properties in software product lines. *Software Quality Journal*, 20(3–4):487–517, 2012. ISSN 0963-9314. doi: 10.1007/s11219-011-9152-9. URL https://doi.org/10.1007/s11219-011-9152-9.

[118] V. Stoico, A. C. Dragomir, and P. Lago. An empirical study on the performance and energy usage of compiled python code, 2025. URL https://arxiv.org/abs/2505.02346. Accepted to the International Conference on Evaluation and Assessment in Software Engineering (EASE) 2025.

[119] T. Sukprasert, A. Souza, N. Bashir, D. Irwin, and P. Shenoy. On the limitations of carbon-aware temporal and spatial workload shifting in the cloud. In *Proceedings of the Nineteenth European Conference on Computer Systems*, EuroSys '24, page 924–941, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400704376. doi: 10.1145/3627703.3650079. URL https://doi.org/10.1145/3627703.3650079.

[120] C. Tang, Z. Wang, X. Zhang, Q. Yu, B. Zang, H. Guan, and H. Chen. Ad hoc transactions in web applications: The good, the bad, and the ugly. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, page 4–18, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392495. doi: 10.1145/3514221.3526120. URL https://doi.org/10.1145/3514221.3526120.

[121] N. Tirel, P. Roose, S. Ilarri, A. Noureddine, and O. Le Goaër. Workload shifting techniques: From digital inebriation to sobriety. *ACM Comput. Surv.*, Oct. 2025. ISSN 0360-0300. doi: 10.1145/3769301. URL https://doi.org/10.1145/3769301.

[122] A. Tomlinson and G. Porter. Something old, something new: Extending the life of cpus in datacenters. *SIGENERGY Energy Inform. Rev.*, 3(3):59–63, Oct. 2023. doi: 10.1145/3630614.3630625. URL https://doi.org/10.1145/3630614.3630625.

[123] A. Torres, R. Galante, M. S. Pimenta, and A. J. B. Martins. Twenty years of object-relational mapping: A survey on patterns, solutions, and their implications on application design. *Information and Software Technology*, 82:1–18, 2017. ISSN 0950-5849. doi: https://doi.org/10.1016/j.infsof.2016.09.009.

[124] P. Tözün, I. Pandis, C. Kaynak, D. Jevdjic, and A. Ailamaki. From a to e: Analyzing tpc's oltp benchmarks: The obsolete, the ubiquitous, the unexplored. In *Proceedings of the 16th International Conference on Extending Database Technology*, EDBT '13, page 17–28. ACM, 2013. ISBN 9781450315975. doi: 10.1145/2452376.2452380. URL https://doi.org/10.1145/2452376.2452380.

[125] Transaction Processing Performance Council (TPC). TPC benchmark C (TPC-C) standard specification, revision 5.11, 2010. http://www.tpc.org/tpcc/.

[126] Transaction Processing Performance Council (TPC). TPC benchmark H (TPC-H) standard specification, revision 3.0.1, 2022. https://www.tpc.org/tpch/.

[127] A. Turcotte, M. W. Aldrich, and F. Tip. reformulator: Automated refactoring of the n+1 problem in database-backed applications. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ASE '22, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9781450394758. doi: 10.1145/3551349.3556911. URL https://doi.org/10.1145/3551349.3556911.

[128] R. Verdecchia, R. A. Saez, G. Procaccianti, and P. Lago. Empirical evaluation of the energy impact of refactoring code smells. In B. Penzenstadler, S. Easterbrook, C. Venters, and S. I. Ahmed, editors, *ICT4S2018. 5th International Conference on Information and Communication Technology for Sustainability*, volume 52 of *EPiC Series in Computing*, pages 365–383. EasyChair, 2018. doi: 10.29007/dz83. URL https://easychair.org/publications/paper/MxpT.

[129] J. Wang, D. S. Berger, F. Kazhamiaka, C. Irvene, C. Zhang, E. Choukse, K. Frost, R. Fonseca, B. Warrier, C. Bansal, J. Stern, R. Bianchini, and A. Sriraman. Designing cloud servers for lower carbon. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pages 452–470, 2024. doi: 10.1109/ISCA59077.2024.00041.

[130] Y. Wang, D. Nörtershäuser, S. Le Masson, and J.-M. Menaud. Potential effects on server power metering and modeling. *Wireless Networks*, 29(3):1077–1084, 2018. ISSN 1022-0038. doi: 10.1007/s11276-018-1882-1. URL https://doi.org/10.1007/s11276-018-1882-1.

[131] P. Wiesner, I. Behnke, D. Scheinert, K. Gontarska, and L. Thamsen. Let's wait awhile: how temporal workload shifting can reduce carbon emissions in the cloud. In *Proceedings of the 22nd International Middleware Conference*, Middleware '21, page 260–272, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450385343. doi: 10.1145/3464298.3493399. URL https://doi.org/10.1145/3464298.3493399.

[132] C. Wimmer, C. Stancu, P. Hofer, V. Jovanovic, P. Wögerer, P. B. Kessler, O. Pliss, and T. Würthinger. Initialize once, start fast: application initialization at build time. *Proc. ACM Program. Lang.*, 3(OOPSLA), 2019. doi: 10.1145/3360610.

[133] T. Xu, L. Jin, X. Fan, Y. Zhou, S. Pasupathy, and R. Talwadker. Hey, you have given me too many knobs!: Understanding and dealing with over-designed configuration in system software. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, page 307–319, 2015. ISBN 9781450336758.

[134] C. Yan, A. Cheung, J. Yang, and S. Lu. Understanding database performance inefficiencies in real-world web applications. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*, CIKM '17, page 1299–1308. ACM, 2017. ISBN 9781450349185. doi: 10.1145/3132847.3132954. URL https://doi.org/10.1145/3132847.3132954.

[135] T. Yang, H. Jiang, Y. Hou, and Y. Geng. Carbon Management of Multi-Datacenter Based On Spatio-Temporal Task Migration. *IEEE Transactions on Cloud Computing*, 11(1):1078–1090, Jan. 2023. ISSN 2168-7161. doi: 10.1109/TCC.2021.3130644. URL https://ieeexplore.ieee.org/abstract/document/9627522.

[136] B. Zhang, Y. A. Dhuraibi, R. Rouvoy, F. Paraiso, and L. Seinturier. CloudGC: Recycling Idle Virtual Machines in the Cloud . In *2017 IEEE International Conference on Cloud Engineering (IC2E)*, pages 105–115, Los Alamitos, CA, USA, Apr. 2017. IEEE Computer Society. doi: 10.1109/IC2E.2017.26. URL https://doi.ieeecomputersociety.org/10.1109/IC2E.2017.26.

[137] C. Zhang, A. G. Kumbhare, I. Manousakis, D. Zhang, P. A. Misra, R. Assis, K. Woolcock, N. Mahalingam, B. Warrier, D. Gauthier, L. Kunnath, S. Solomon, O. Morales, M. Fontoura, and R. Bianchini. Flex: high-availability datacenters with zero reserved power. In *Proceedings of the 48th Annual International Symposium on Computer Architecture*, ISCA '21, page 319–332. IEEE Press, 2021. ISBN 9781450390866. doi: 10.1109/ISCA52012.2021.00033. URL https://doi.org/10.1109/ISCA52012.2021.00033.

[138] I. Švogor, I. Crnković, and N. Vrček. An extensible framework for software configuration optimization on heterogeneous computing systems: Time and energy case study. *Information and Software Technology*, 105:30–42, 2019. ISSN 0950-5849.