UNIVERSITÉ DE MONTRÉAL

UNDERSTANDING THE IMPACT OF CLOUD COMPUTING PATTERNS ON PERFORMANCE AND ENERGY CONSUMPTION

SEYED AMIRHOSSEIN ABTAHIZADEH
DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE INFORMATIQUE)
AOÛT 2016

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé:

UNDERSTANDING THE IMPACT OF CLOUD COMPUTING PATTERNS ON PERFORMANCE AND ENERGY CONSUMPTION

présenté par: ABTAHIZADEH Seyed Amirhossein
en vue de l'obtention du diplôme de: Maîtrise ès sciences appliquées
a été dûment accepté par le jury d'examen constitué de:

ACKNOWLEDGMENT

First and foremost, I would like to thank my thesis advisor Prof. Foutse Khomh of the Computer Engineering department. The door to his office was always open whenever I ran into a trouble spot or had a question about my research or writing. He consistently allowed this thesis to be my own work, but steered me in the right direction whenever he thought I needed it. I am grateful that I've studied under his supervision and I was able to explore and obtain a variety of knowledge in Cloud Computing and related topics.

I would like to thank members of the jury, who have reviewed my thesis.

Finally, I would like to thank Amazon Inc. for supporting me through their AWS in Education Grants.

RÉSUMÉ

Les patrons infonuagiques sont des solutions abstraites à des problèmes récurrents de conception dans le domaine de l'infonuagique. Bien que des travaux antérieurs aient prouvé que ces patrons peuvent améliorer la qualité de service des applications infonuagiques, leur impact sur la consommation d'énergie reste encore inconnu. Pourtant l'efficacité énergétique est un défi majeur pour les systèmes infonuagiques. Actuellement, 10% de l'électricité mondiale est consommée par les serveurs, les ordinateurs portables, les tablettes et les téléphones intelligents. La consommation d'énergie d'un système dépend non seulement de son infrastructure matérielle, mais aussi de ses différentes couches logicielles. Le matériel, le firmware, le système d'exploitation, et les différentes composantes logicielles d'une application infonuagique, contribuent tous à déterminer son empreinte énergétique. De ce fait, pour une meilleure efficacité énergétique, il est important d'améliorer l'efficacité énergétique de toutes les couches matérielles et logicielles du système infonuagique, ce qui inclut les applications déployées dans le système infonuagique.

Dans ce mémoire, nous examinons l'impact de six patrons infonuagiques (Local Database proxy, Local Sharding Based Router, Priority Queue, Competing Consumers, Gatekeeper and Pipes and Filters) sur la consommation d'énergie de deux applications multi-traitement et multi-processus déployées dans un système infonuagique. La consommation d'énergie est mesurée avec l'outil Power-API, une interface de programmation d'application (API) écrite en Java et permettant de mesurer la consommation d'énergie au niveau du processus. Les résultats de nos analyses montrent que les patrons étudiés peuvent réduire efficacement la consommation d'énergie d'une application infonuagique, mais pas dans tous les contextes.

D'une manière générale, nous prouvons qu'il y a un compromis à faire entre performance et efficacité énergétique, lors du développement d'une application infonuagique. De plus, nos résultats montrent que la migration d'une application vers une architecture de micro-services peut améliorer les performances de l'application, tout en réduisant considérablement sa consommation d'énergie. Nous résumons nos contributions sous forme de recommandations que les développeurs et les architectes logiciels peuvent suivre lors de la conception et la mise en œuvre de leurs applications.

ABSTRACT

Cloud Patterns are abstract solutions to recurrent design problems in the cloud. Previous works have shown that these patterns can improve the Quality of Service (QoS) of cloud applications, but their impact on energy consumption is still unknown. Yet, energy consumption is the biggest challenge that cloud computing systems (the backbone of high-tech economy) face today. In fact, 10% of the world's electricity is now being consumed by servers, laptops, tablets and smart phones. Energy consumption has complex dependencies on the hardware platform, and the multiple software layers. The hardware, its firmware, the operating system, and the various software components used by a cloud application, all contribute to determining the energy footprint. Hence, increasing a data center efficiency will eventually improve energy efficiency. Similarly, software itself can affect the internal design of cloud-based applications to optimize hardware utilization to lower energy consumption.

In this work, we conduct an empirical study on two multi-processing and multi-threaded cloud-based applications deployed in the cloud, to investigate the individual and the combined impact of six cloud patterns (Local Database proxy, Local Sharding Based Router, Priority Queue, Competing Consumers, Gatekeeper and Pipes and Filters) on the energy consumption. We measure the energy consumption using Power-API, an application programming interface (API) written in Java to monitor the energy consumed at the process-level. Results show that cloud patterns can effectively reduce the energy consumption of a cloud application, but not in all cases.

In general, there appear to be a trade-off between an improved response time of the application and the energy consumption. Moreover, our findings show that migrating an application to a microservices architecture can improve the performance of the application, while significantly reducing its energy consumption. We summarize our contributions in the form of guidelines that developers and software architects can follow during the design and implementation of their applications.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABREVIATIONS

IT	Information Technology
QoS	Quality of Service
SLA	Service Level Agreement
API	Application Programming Interface
EC2	Amazon Elastic Computing
AWS	Amazon Web Services
IaaS	Infrastructure as a Service
SaaS	Software as a Service
PaaS	Platform as a Service
MS-Azure	Microsoft Azure: Cloud Computing Platform and Services
VMM	Virtual Machine Monitor
SSL	Secure Sockets Layer
TLS	Transport Layer Security
HTTPS	Hypertext Transfer Protocol over TLS
ACID	Atomicity, Consistency, Isolation, Durability
BASE	Basic Availability, Soft-state, Eventual consistency
HPC	High Performance Computing
SEE	Software Engineering Environment
SOA	Service Oriented Architecture
CLI	Command-Line Interface
MQ	Priority Message Queue pattern
SHRD	Sharding pattern
PRX	Proxy pattern
CCP	Competing Consumers pattern
P&F	Pipes and Filters pattern
GK	Gatekeeper pattern

LIST OF APPENDICES

CHAPTER 1 INTRODUCTION

Cloud computing systems are now pervasive in our society. As a consequence, the energy consumption of data centers and cloud-based applications has become an emerging topic in the software engineering research communities. Energy consumption has complex dependencies on the hardware platform, and the multiple software layers. The hardware, its firmware, the operating system, and the various software components used by a cloud application, all contribute to determining the energy footprint; making energy optimization a very challenging problem. The role of software components and coding practices has been recently investigated, and researchers have proposed energy-aware algorithms (Prosperi et al., 2012) and sensor relocation techniques (El Korbi and Zeadally, 2014) to help optimizing software energy consumption. When developing an energy efficient cloud-based application, developers must seek a compromise between the application's Quality of Service (QoS) and energy efficiency. Manually finding such a compromise is a daunting task, and developers should be supported by guidelines and–or tailored recommendations in the form of best practices.

Cloud patterns, which are general and reusable solutions to recurring design problems, have been proposed as best practices to guide developers during the development of cloud-based applications. However, although previous work (Hecht et al., 2014) has shown that these cloud patterns can improve the QoS of cloud based applications, their impact on energy consumption is still unknown. In this study, we evaluate the impact on energy consumption of six cloud patterns: Local Database Proxy, Local Sharding-Based Router, Priority Queue, Competing Consumers, and Pipes and Filters. The study is performed using two different applications exhibiting the behavior of a real cloud-based application.

The first experiment uses a RESTful multi-threaded application written in Java, and the second experiment consists of an application implemented with the Python Flask microframework, using both multi-processing and multi-threaded scenarios, deployed in the Amazon EC2 cloud. These two systems are also implemented with different combinations of the aforementioned patterns. The second application is also decomposed into seven microservices in order to enable investigating the impact of microservices design architecture in a cloud-based scheme. Energy consumption is measured using Power-API, an application programming interface (API) written in Java that can monitor the energy consumed by an application, at the process-level (Noureddine et al., 2012).

1.1 Concepts and Definitions

In this section, brief summaries of essential concepts related to the research topic are presented (Furht and Escalante, 2010):

Cloud Computing: ISO/IEC 17788:2014 (Author, 2014) defines cloud computing as a "paradigm for enabling network access to a scalable and elastic pool of shareable physical or virtual resources with self-service provisioning and administration on-demand". In fact, we can say that cloud computing is a computing model, not a technology. In cloud computing, Information Technology (IT) experts and end users can access cloud resources such as servers, networks and applications via the Internet, following a Pay-As-You-Go model in which users pay only for the amount of computing services that they consumed.

Cloud Patterns: Cloud design patterns are determined solutions to frequent design problems in cloud applications. These patterns are repeatable, and they describe the context in which they should be applied, the problem that they solve and the solution to this problem.

Vertical Scaling: Consists in increasing the capacity of existing commodity hardware or software by adding more resources such as memory or CPU when needed. In other words, enhancing an individual computing resource capabilities by providing more hardware equipments, and resizing the server with no change to the code.

Horizontal Scaling: Consists in connecting innumerable hardware or software units so that they function as a single logical monad. Clustering machines is a way of scaling out the original server by adding more compute or storage resources. The most important advantage of horizontal scaling is the ability to increase the capacity of a system on the fly, which is often referred to as elastic computing in the cloud domain.

Public Cloud: Public cloud refers to cloud computing providers that make applications, storage and resources available to the public over the Internet. This type of cloud computing is often criticized in terms of data exposure and security concerns. In this work, instances from Amazon Inc. Cloud Computing infrastructure were used. In particular, we setup the second experiment on Amazon Elastic Computing (EC2) services virtual machine instances.

Private Cloud: When a cloud provider infrastructure is distinctively used by one customer or organization and it is not shared with others. Security challenges along with the data exposure might be less likely to happen in comparison with public cloud as there is a higher level of isolation and control over data. In general, this solution is suitable for protecting sensitive and valuable data. Utilizing a private cloud (locally or hosted by a third-party) enhances an organization with a dedicated pool of IT resources that can be adjusted to requirements easily and quickly in a much secure way than the public cloud.

Hybrid Cloud: The Hybrid solution is to allow the work loads to move between private and public cloud, hence providing more resiliency to the business. Hybrid approach utilizes both advantages of public and private cloud. Sensitive data can be stored in a private cloud while less-critical computations and functions could be hosted on a public cloud. Hybrid cloud is a solution for cloud applications with dynamic or highly changeable work loads.

IaaS: Infrastructure as a Service makes available to the general public (users and IT organizations) the functions of an entire data center, by providing grids or clusters and virtualized servers, networks, and storage to name a few. The best known example is Amazon Elastic Compute Cloud (EC2) which allows users to access a variety of instances to deploy their cloud applications.

PaaS: Platform as a Service provides virtualized servers in which users can run existing applications or develop new ones on them. The key benefit is that users do not have to maintain the operating systems or hardware along with load balancing or even computing capacity and they can focus only on the development and deployment of their applications. Moreover, PaaS also provide Application Programming Interfaces (API) that developers can use during the development of their applications. Microsoft Azure is the best known example of such service in cloud.

SaaS: Software as a Service brings about all the functions of a conventional application, but through a Web browser, not a locally-installed application. SaaS tends to eliminate tasks concerning servers, storage, application development and common concerns of IT by allowing users to run applications remotely from the cloud. GMail by Google is an example of SaaS.

Elastic Computing: Elastic Computing can be defined as the ability of a cloud provider to provision flexible computing power (in terms of CPU, memory, storage, etc.) when required. Elasticity allows applications to adapt to workloads in real time. In fact, a platform maintains elasticity by scaling up and down computing resources in accordance with the demand requested by cloud application users. Unlike scalable platforms which is designed to meet some expected demands, elasticity allows to meet both expected and unexpected demands by increasing or decreasing the capacity to adjust the workloads to application requirements.

Amazon Web Services (AWS): It is a comprehensive cloud computing platform and infrastructure provided by Amazon Inc[1]. Instances from Elastic Computing services (Amazon EC2) were chosen to conduct our second experiment.

NoSQL: NoSQL or non-SQL databases are open source and horizontally scalable databases which are non-relational and distributed, supporting large numbers of concurrent users. NoSQL databases set a goal of delivering highly responsive experiences to globally distributed users. Types of such databases include: column, document, key-value, graph and multi-model. NoSQL databases focus on availability and eventual consistency by using Basic Availability, Soft-state Eventual consistency (BASE) transactions. BASE emphasizes the fact that it is enough for the database to eventually be in a consistent state, hence the NoSQL databases are more suitable for distributed data. In the context of cloud, NoSQL databases are widely used because of the distributed nature of cloud-based applications and the high demand for availability.

Hypervisor: A hypervisor or virtual machine monitor (VMM) is a computer software, firmware or hardware that enables creating and running virtual machines.

Virtualization: Virtualization means creating virtual devices or resources, such as computer server, network, storage device, or an operating system. Virtualization has a significant underlying role in cloud computing. The main benefit of Virtualization is the subdivision of commodity hardware resource into multiple execution environments. Cloud providers offer services based on virtualized resources. There exists different kind of virtualization: Full Virtualization which is the complete simulation of the hardware, Partial Virtualization in which a few but not all of the hardware is simulated, Paravirtualizaiton which provides isolated domains for the guest programs to be executed whereas the hardware is not simulated, and finally, Hardware-assisted virtualization which is a platform virtualization approach that enables full virtualization using help from hardware capabilities.

VMware ESXi: VMware ESXi, which is the latest hypervisor architecture from VMWare has been used in our study. We chose this architecture because it contains a new bar for security and authenticity which reduces the “attack surface" with less code to patch [2]. Moreover, it is used by leading server manufacturers such as Dell, IBM, HP, and Fujitsu-Siemens.

[1]`https://aws.amazon.com/`

[2]http://www.vmware.com

1.2 Research Objectives

This research aims to explore the effects of cloud patterns on the energy consumption and performance of cloud-based applications. In particular, we analyze the potential impact of cloud patterns on both energy utilization and performance, and we propose a series of architectural design guidelines to cloud developers that helps optimizing energy and performance efficiency.

1.3 Thesis Plan

Chapter 2 presents a literature review of the evaluation of cloud patterns, and discusses common software-defined power meters. Chapter 3 contains information about patterns that have been used in this research along with their use cases and benefits for cloud-base applications. Chapter 4 introduces the methodology of our experiments that aimed to understand the effect of cloud patterns on performance and energy consumption. It also summarizes the design of the two Software as a Service (SaaS) frameworks used to implement the applications of our experiments. In Chapter 5, we discuss our results and provide guidelines for developers of cloud-based applications, then we discuss threats to the validity of our experimental study. In Chapter 6, we summarize our studies, and we conclude on how this study could help the design of an architectural recommendation system for developers of cloud-based applications. This chapter also discusses some avenues for future work.

CHAPTER 2 LITERATURE REVIEW

Energy consumption is the biggest challenge that cloud computing systems face today. Pinto et al. (Pinto et al., 2014), who analyzed more than 300 questions and 550 answers on StackOverflow question-and-answer site for developers, reported that the number of questions on energy consumption increased by 183% from 2012 to 2013. The majority of those questions were related to software design, showing that developers need guidance for designing green software systems. Similarly, Pang et al. (Pang et al., 2016) found that developers lack knowledge on how to develop energy-efficient software. Nowadays, cloud applications are part of the business world as they eventually increase the benefits of high availability, scalable and managed solutions. Many efforts have been focused on modeling the energy consumption of cloud infrastructures. However, most of the foregone studies were limited to hardware optimization, neglecting the benefits that can be achieved through software optimization. When developing an energy efficient cloud-based application, developers must seek a compromise between the application's Quality of Service (QoS) and energy efficiency.

Cloud patterns, which are general and reusable solutions to recurring design problems, have been proposed as best practices to guide developers during the development of cloud-based applications. Although previous works such as (Hecht et al., 2014) and (Ampatzoglou et al., 2012) have shown that patterns can improve the QoS of cloud based applications, their impact on energy consumption is still unknown.

A design pattern describes a common problem and provides a corresponding solution, and we use patterns to document the solution in a generic template format so that it can be repeatedly applied. Our inquiry takes into account the knowledge of design patterns, not only to purvey an understanding of the plausible problems (that such designs may be subjected to), but also to provide answers as to how these problems are best tackled.

2.1 Object-Oriented Design Patterns and Software Quality

Several works in the literature have assessed the impact of design patterns on software quality (Khomh and Guéhéneuc, 2008), software maintenance (Vokáč et al., 2004) and code complexity (Prechelt et al., 2001). Overall, these studies found that design patterns do not always improve the quality of applications. Khomh and Guéhéneuc (Khomh and Guéhéneuc, 2008) claim that design patterns should be used with caution during software development because they may actually impede software quality. Object Oriented design patterns are usually not supposed to increase performance, nevertheless, Aras *et. al* (Aras et al., 2005) have found that design patterns can have a positive effect on the performance of scientific applications despite the overhead that they introduce (by adding additional classes). Of course the results of these studies cannot be directly generalized to cloud patterns, which usually focus on scalability and availability. However, they provide hints about the possible benefit and downside of cloud patterns. Clearing up the impact of cloud patterns on energy consumption as well as QoS is important to help software development teams make good design decisions.

2.2 Evaluation of Cloud Patterns

Ardagna et al. (Ardagna et al., 2012) empirically evaluated the performance of five scalability patterns for Platform as a service (PaaS): Single, Shared, Clustered, Multiple Shared and Multiple Clustered Platform Patterns. To compare the performance of these patterns, they measured the response time and the number of transactions per second. They explored the effects of the addition and the removal of virtual resources, but did not examine the impact of the patterns on energy consumption. Tudorica et al. (Tudorica and Bucur, 2011) and Burtica et al. (Burtica et al., 2012) performed a comprehensive comparison and evaluation of NoSQL databases (which make use of multiple sharding and replication strategies to increase performance), but did not examine energy consumption aspects. Along the same line of work, Cattel (Cattell, 2011) examined NoSQL and SQL data stores designed to scale by using replication and sharding. They also did not perform energy consumption evaluations. Message oriented middlewares have been benchmarked by Sachs et al. (Sachs et al., 2009), however, the energy consumption aspect was ignored.

Regarding the relationship between cloud patterns and energy consumption, Beloglazov (Beloglazov, 2013) proposed novel techniques, models, algorithms, and software for dynamic consolidation of Virtual Machines (VMs) in Cloud data centers, that support the goal of reducing the energy consumption. His design architecture is reported to improve the utilization of data center resources and reduce energy consumption, while satisfying defined QoS requirements. Ultimately, his work led to the design and implementation of OpenStack Neat[1], which is "an extension to OpenStack implementing dynamic consolidation of Virtual Machines (VMs) using live migration".

[1]http://openstack-neat.org

2.3 Green Software Engineering

Software has an indirect effect on the environment, since it intensely affects the hardware functioning. Therefore, it should be written efficiently to avoid overusing the underlying hardware. There is a lack of models/descriptions with regard to environmental sustainability in the area of computer software. In fact, we can argue that software's building process should consider energy efficiency. New green software engineering processes must be created to fulfill the constraints of energy efficiency, and developers can produce efficient code if they can understand the effects of their code on energy consumption.

We set out to investigate how applying architectural patterns can have a compelling effect on the energy consumption of cloud-based applications. Although developers can take advantage of software tools that monitor resources in order to observe energy consumption, there is still lack of programming guidelines (*i.e.*, implementation of patterns) that they can follow during the design of their application. Although there have been efforts to improve the energy efficiency of hardware components, very few approaches examined the role that architecture plays on the energy efficiency of an application. There is a lack of software process that developers can follow to minimize the energy consumed by their application, while preserving the QoS. Mahmoud et al. (Mahmoud and Ahmad, 2013) have introduced a two level software development process, based on agile principles, and which is claimed to be environmentally sustainable. Each software stage has been marked with specific metric, representing its level of sustainability. Although this process can help reduce the energy footprint of an application's development cycle, it does not necessarily improve the energy efficiency of the application. A comprehensive overview of sustainability perspectives in software engineering is presented in (Calero and Piattini, 2015). This book discusses the impact of software on environment and provides important guidelines for making software engineering “green”, i.e., reducing their environmental and energy footprints.

2.4 Software-defined Power Meters

Software-defined power meters are configurable software libraries that can estimate the power consumption of a software in real-time. Power estimation of software processes can provide critical indicators to optimize the overall energy consumed by an application. Software libraries that enable measuring the energy of cloud-based applications can help monitor and improve the design of applications, making them more energy efficient. In this section, we briefly describe some common tools used in both academia and industry, to estimate the power consumption of applications.

2.4.1 Power-API

Power-API is a system-level library that provides power information per Process ID for each system component (*e.g.,* CPU, network card, etc.) (Noureddine et al., 2012). The energy estimation is performed using analytical models that characterize the consumption of the components (*i.e.,* CPU, memory, disk). The accuracy of Power-API was evaluated using the bluetooth PowerMeter (PowerSpy) (Vereecken et al., 2010), and results revealed only minor variations between the energy estimations of Power-API, and the energy consumption measured by PowerSpy (Noureddine et al., 2012).

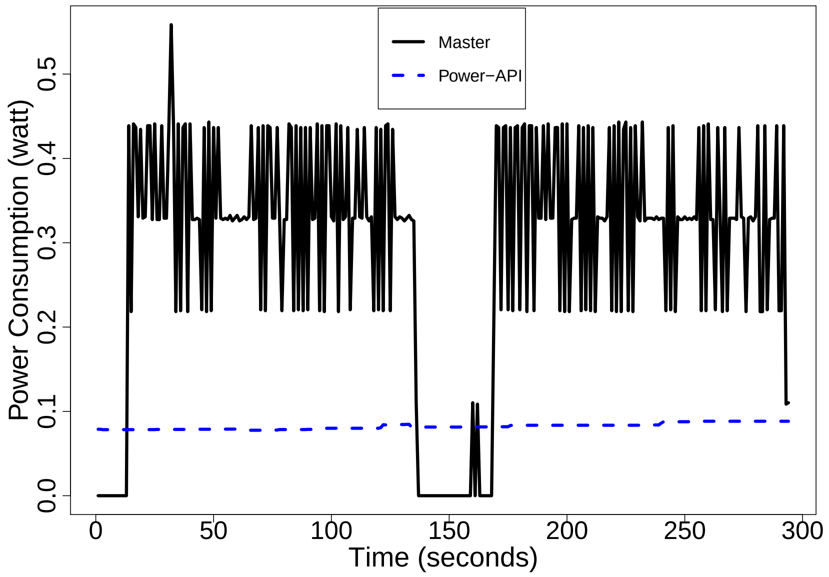


Figure 2.1 Power-API distortion test

Power-API was selected for this work because of its reported high accuracy (Noureddine et al., 2012), and the aggregate energy consumption data for CPU, Memory and Disk were collected by auditing multiple corresponding virtual machines. Power-API measures power in watts, which was converted to the unit of energy (Kilojoules). However, prior to our study, we performed a pilot procedure to examine the risk of Power-API to introduce noise in its own measurements. More explicitly, we conducted a test in which 10,000 records were inserted twice with a short gap of time into our database and both the process of Power-API and the process of the database were measured. Power-API itself runs as a process in Linux operating system, therefore, we wanted to make sure that such process does not, actually, consume energy and introduce noise in our measurements.

The result illustrated in Figure 2.1 shows that Power-API does not introduce noise in its measurements; it did not alter the energy measurements of the master server during the period of insertion. In fact, the figure shows that Power-API process itself consumes not considerable amount of energy (dashed line) and it is stable during the whole experiment, revealing that the middleware never introduces noise in measurement. The whole test duration was approximately 300 seconds with a short gap in the middle. If Power-API brought noise in energy consumption measurement, we would have expected to see a fluctuating line for its energy amount throughout the test process. In second experiment, we used a newer released version of Power-API which supports the power estimation of CPU and Memory for CPU-intensive applications. This version of Power-API provided a Command-Line Interface (CLI) that as embedded in a bash script file in order to measure energy consumption of the application during each execution of the application.

2.4.2 TPC-Energy

The TPC benchmarking tool (Nambiar et al., 2014) enables benchmarking databases performance by simulating a series of users queries against the database. TPC-Energy (newly introduced) is the specification that is used to report energy consumption in such benchmarks. It includes energy consumption of servers, disks, and other items that consume power and that are required by the benchmark specification. TPC-benchmark supports Atomicity, Consistency, Isolation, and Durability (ACID) in data queries and transactions. However, in the cloud context, most transactions are Basically Available, has a Soft-state and are Eventually consistent (i.e., they are BASE).

2.4.3 Joulemeter

Joulemeter (Goraczko et al., 2011) is a software tool that can estimate the power consumption of a virtual machine (VM). Joulemeter was developed by Microsoft. It can measure memory, disk and CPU energy consumption. Although this tool has been reportedly used by IT managers, the public download is now closed. Besides, the software works only on Windows operating system.

2.4.4 jRAPL

jRAPL[2] is an open source application framework recently introduced by Kenan Liu (Liu et al., 2015) to profile Java programs running on CPUs with Running Average Power Limit

[2]https://github.com/kliu20/jRAPL

(RAPL) support. It enables chip-level power management, and it is widely supported in today's Intel architectures. RAPL allows energy/power consumption to be reported at a fine-grained level, *e.g.,* monitoring CPU core, CPU uncore (L3 cache, on-chip GPUs, and interconnects), and DRAM separately.

2.4.5 Power Gadget

Power Gadget[3] is a software-based power monitoring tool developed by Intel Inc. Power Gadget provides libraries for the estimation and monitoring of power consumption in real-time; using the energy counters in the processor. Its latest version 3.0 enables power estimation on multi-socket systems. It also provides externally callable APIs that can be used to extract power information from source code. The limitation of this tool is that it only supports the 2nd Generation up to the 6th generation of the Intel® Core^{TM} processor. Moreover, it runs only on Windows and MAC OS X operating systems.

[3]https://software.intel.com/en-us/articles/intel-power-gadget-20

CHAPTER 3 CLOUD PATTERNS

Designing applications for the cloud has some challenges. In the cloud, there are no precise users requirements because we don't know all the users needs, and we don't know the devices from which users will access the application. Therefore, we are unable to predict the application workload. All we can do is to anticipate failures since the application will depend on third-party software on which we have no control. Also, in the cloud, there is a cost for every load that is executed, so there must be a way to handle that. Given the magnitude of these challenges, best practices have been proposed in the form of cloud patterns to assist and guide developers during the design and implementation of their application.

Cloud Patterns are abstract solutions to recurrent design problems in the cloud. Despite their wide adoption, little is known on the effectiveness of these solutions. In the following, we discuss several possible problems that could be tackled when such architectural design guidelines (*i.e.* patterns) are being employed.

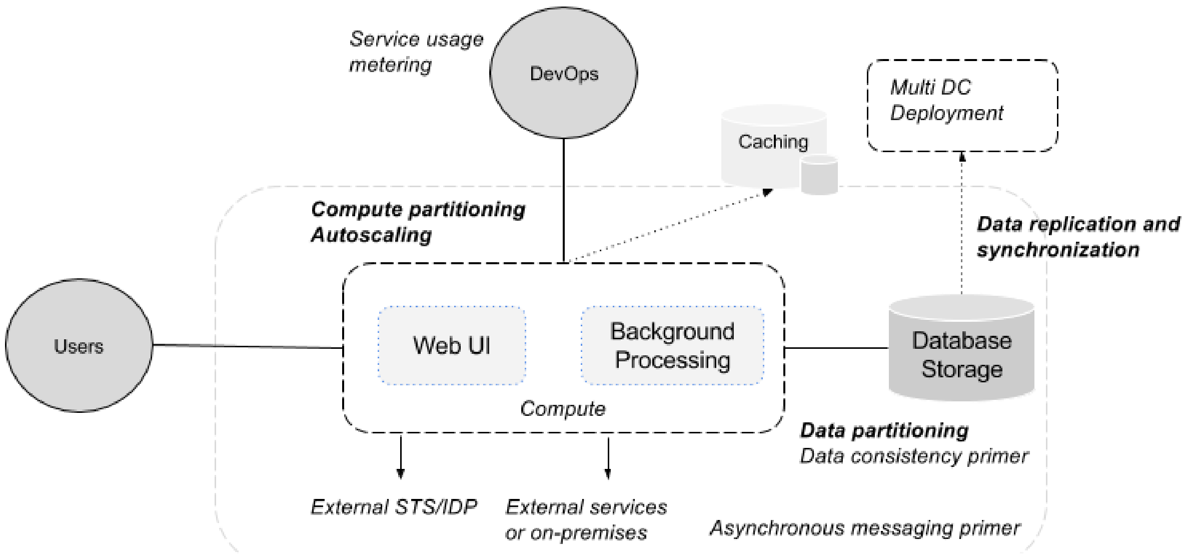


Figure 3.1 Basic orientation for developing cloud applications (Homer et al., 2014)

Figure 3.1 demonstrates exemplary contexts of cloud application development, in which different dimensions of common problems are illustrated. As one can see on Figure 3.1, there are many problems faced by developers during the development of cloud based applications. In this thesis, we focus on four of these problems:

- **Messaging:** Cloud applications are usually distributed. Hence, a messaging infrastructure is required to connect components and services to maximize scalability. Asynchronous messages are commonly used in cloud applications. However, they pose many issues such as messages ordering, poison message management, or idempotency.

- **Data Management:** When deploying applications in the Cloud, developers have to distribute databases in different locations to ensure a good performance, scalability and-or availability. This poses issues of data consistency and data synchronization across the multiple instances.

- **Performance and Scalability:** Performance measures the responsiveness of a system when executing a request or command in a given time. Scalability, on the other hand, enables a system to handle increased workloads without a change in performance, using available resources. Cloud applications are much likely to encounter sudden increases/decreases in workloads than traditional applications, hence, they require a flexible architecture that can scale out/in on demand to ensure good performance and high availability.

- **Security:** Cloud applications must defeat and prevent malicious activities and prevent disclosure and–or loss of information. These applications are exposed on the Internet. Therefore, the design and deployment of such applications should restrict access to only trusted users and protect sensitive data/information.

Table 3.1 summarizes some common problems that developers face when developing cloud-based applications, along with the corresponding pattern that can be implemented to tackle the problem. For example, the Gatekeeper pattern can be used whenever security is critical, and Sharding can be used to solve data management issues when the data is partitionable, etc.

Table 3.1 Common Problems in Cloud Area and Suggested Patterns

PROBLEM IN CLOUD AREA	SUGGESTED PATTERN
Messaging	Competing Consumers, Pipes & Filters, Priority Queue
Performance & Scalability	Competing Consumers, Priority Queue, Proxy, Sharding
Security	Gatekeeper
Data Management	Sharding

3.1 Cloud Patterns

We now discuss the six patterns studied in this thesis.

3.1.1 Local Database Proxy

This pattern uses data replication between master/slave databases and a proxy to route requests (Strauch et al., 2012). Write requests are handled by the master and replicated on its slaves, while Read requests are processed by slaves. When applying this pattern, components must use a local proxy whenever they need to retrieve or write data. The proxy distributes requests between master and slaves depending on their type and workload. Slaves may be added or removed during the execution to obtain elasticity. There could be a risk of bottleneck on the master database when there is a need to scale with write requests. This issue together with the lack of strategy for write requests are the main limitations of this pattern. The impact of this pattern on the QoS of applications has been examined by Hecht et al. (Hecht et al., 2014); however, to the best of our knowledge, no work has empirically investigated the impact of the Local Database Proxy pattern on the energy consumption of applications.

3.1.2 Local Sharding-Based Router

This pattern is useful when an application needs scalability both for read and write operations (Strauch et al., 2012). Sharding is a technique that consists in splitting data between multiple databases into functional groups called shards. Requests are processes by a local router to determine the suitable databases. Data are split horizontally i.e. , on rows, and each split must be independent as much as possible to avoid joins and to benefit from the sharding. The sharding logic is applicable through multiple strategies; a range of value, a specific shard key or hashing can be used to distribute data among the databases. It is possible to scale the system out by adding further shards running on redundant storage nodes.

Sharding reduces contentions and improves the performance of applications by balancing the workload across shards. Shards can be located close to specified clients to improve data accessibility. When combined with other patterns, Sharding can have a positive impact on the QoS of applications (specifically when experiencing heavy loads) (Hecht et al., 2014). However, the impact of Sharding on energy consumption is still unclear.

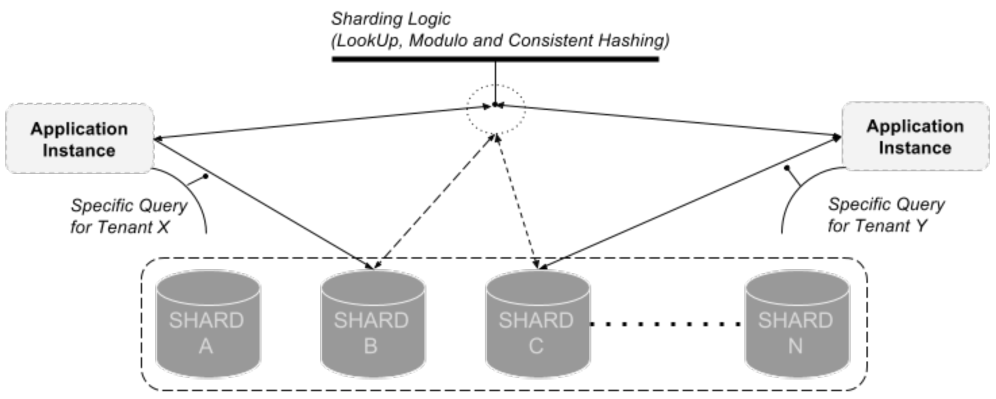

Sharding Logic
(LookUp, Modulo and Consistent Hashing)
Application Instance
Application Instance
Specific Query for Tenant X
Specific Query for Tenant Y
SHARD A
SHARD B
SHARD C
SHARD N

Figure 3.2 Local Sharding-Based Router Pattern (Homer et al., 2014)

Figure 3.2 illustrates the implementation of the Sharding pattern. Whenever the data structure allows for sharding, significant performance improvements can be achieved since the processing of requests is distributed across multiple shards and servers in the network. The main disadvantage of the sharding pattern is the fact that application developers have to write extra code to handle the sharding logic, which eventually makes the overall design process more complex.

A performance degradation can be observed when the capacity of a shard is exceeded. This can happen when a customer interacts with the database more, hence this customer has more allocation of read/write requests than the others. Overall, sharding can help improve the number of queries that an application can handle, resulting in higher scalability and availability. Also, a good sharding logic creates a good data locality, which improve the response time of the application; local data are accessed faster.

3.1.3 Priority Message Queue

This pattern which implements a First In First Out (FIFO) queue is typically used to surrogate tasks to background processing or to allow asynchronous communications between components. Priority Message Queue is recommended when there are different types of messages. Basically, messages with high priority values are received and processed more quickly than those with lower priority values. Message Queues enable designing loosely coupled components and improve the scalability of applications (Hecht et al., 2014). Figure 3.3 depicts the overall architecture of this pattern.

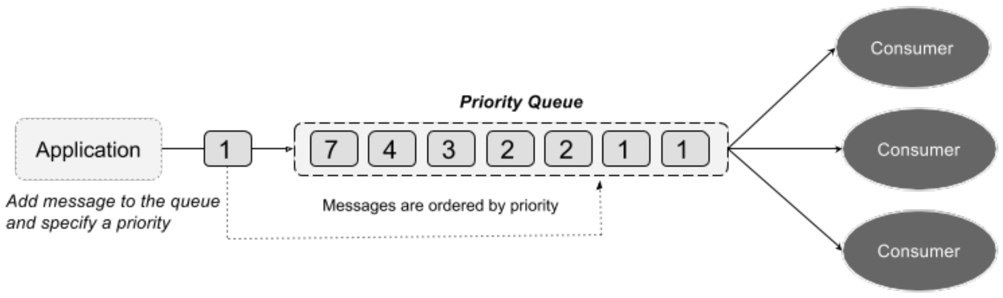

Priority Queue
Application
1
7
4
3
2
2
1
1
Add message to the queue and specify a priority
Messages are ordered by priority
Consumer
Consumer
Consumer

Figure 3.3 Priority Message Queue Pattern (Homer et al., 2014)

3.1.4 Competing Consumers

A cloud-based application is likely to face a large number of requests sent by the users. A common technique to provide responses is to pass them through a *messaging system* to another service that processes them asynchronously rather than synchronously ensuring that the business logic of the application is not blocked. In addition, requests in a cloud domain may increase significantly over time and unpredictable workloads might be exposed to the application. The Competing Consumers pattern allows applications to handle fluctuating workloads (from idle times to peak times), by deploying multiple instances of the consumer service coordinated to ensure that a message is only delivered to a single consumer. The architecture recommends to use a message queue as the communication channel (and as a buffer) between the application and the instances of the consumer service. Figure 3.4 illustrates the architecture of this pattern.

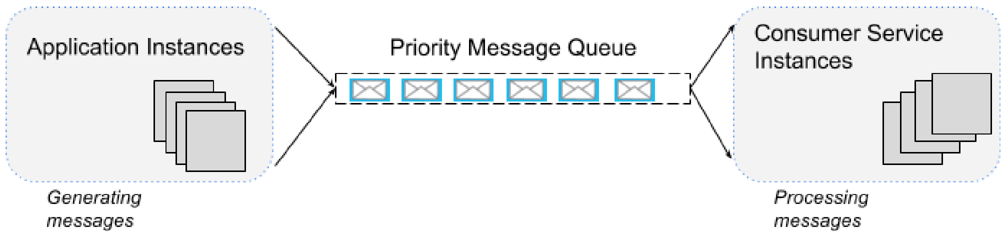


Figure 3.4 Competing Consumer Pattern (Homer et al., 2014)

When using this pattern, multiple concurrent consumers can process messages received on the same messaging channel, which can result in a system that processes multiple messages concurrently with an optimized throughput. In other words, this pattern would potentially improve scalability and availability, and can help balance the workload. The Competing Consumers pattern enables handling wide variations in the volume of requests sent by application instances, and improves reliability. It guarantees that a failed service instance will not result in blocking a producer, and messages can be processed by other working services.

This pattern eliminates complex coordination between the consumers, or between the producer and the consumer instances, therefore increasing maintainability. Instances of a consumer service can be dynamically added or removed as the volume of messages change. This pattern will be implemented and investigate in terms of performance and energy optimization.

3.1.5 The Gatekeeper

This pattern describes a way of brokering access to data and storage. In cloud applications, usually, instances directly connect to the storage, maximizing the risk of exposing sensitive data. If a hacker gains access to the host environment, the security mechanisms and keys will be compromised, therefore, the data itself can be divulged.

The gatekeeper is a web service designed to handle requests from clients. It is the key entry point of the system. It processes and directs requests to trusted messages on another instance(s) called *Trusted Host*. The *Trusted Host* holds the necessary code and security measures required to retrieve data from the storage.

In fact, we tend to decouple application instances from the storage retrieving instances. The goal of this pattern is to bring about control validation, limited risk and exposure, and appropriate security to cloud applications. The architecture of this pattern is presented in Figure 3.5. The main issue is to ensure that trusted host only connects to the gatekeeper, therefore, a secure communication channel (HTTPS, SSL, or TLS) must be employed. Moreover, as the gatekeeper could be the single point of failure in such design, we must consider deploying additional instances along with auto-scaling mechanisms. This pattern is suitable for applications that handle sensitive/protected information or data, and whenever the validation of requests must be performed separately from the tasks inside a system.

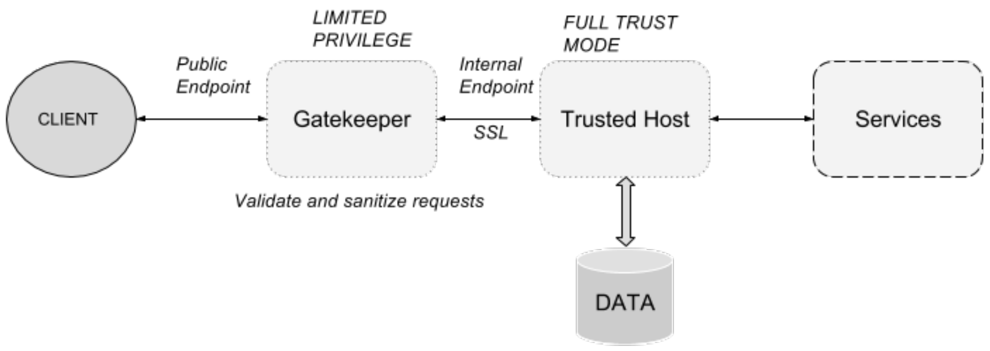


Figure 3.5 Gatekeeper Pattern (Homer et al., 2014)

3.1.6 Pipes and Filters

Cloud-based applications often perform a variety of tasks of varying complexity. The Pipes and filters pattern recommends to decompose processing into a set of discrete components i.e., *filters*. A communication channel i.e., *pipes* is required to convey the transformed data (from each step) between the filters. This pattern is used to ensure performance and scalability. It also improves resiliency because if a task is failed, it can be rescheduled on another instance. Failure of an instance (filter) does not necessarily result in the failure of the entire pipeline. Figure 3.6 presents an illustration of the pipes and filters pattern.

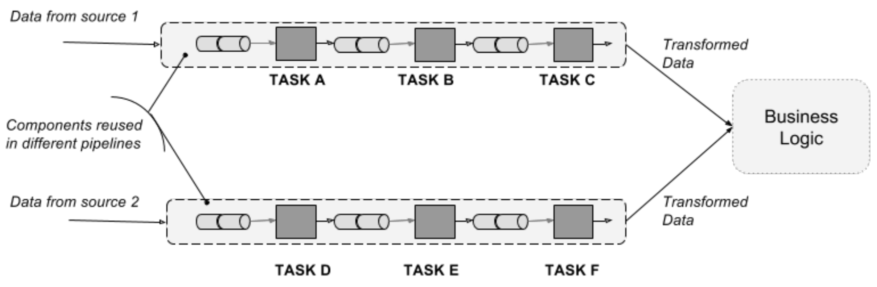


Figure 3.6 Pipes and Filters Pattern (Homer et al., 2014)

The main disadvantage of this pattern is the fact that the time required to process a single request depends on the speed of the slowest filter in the pipeline. There is also a risk that one or more filters form a bottleneck. The deployment automation and testing of pipe and filters architectures can be complex because developers and testers might have to deal with a variety of technologies, and isolated data sources. In this thesis, we examine the impact of this pattern on the energy efficiency of applications.

CHAPTER 4 METHODOLOGY AND DESIGN

4.1 Context and Problem

Previous work has shown that cloud patterns can improve the Quality of Service (QoS) of cloud applications (Hecht et al., 2014), but their impact on energy consumption is still unknown. This study sets out to empirically investigate the impact of six patterns on the performance (response time) and energy consumption of cloud-based applications. The goal of this study is to provide architectural design guidelines to developers and raise their awareness about the trade-offs between performance and energy optimization, during the development of cloud-based applications.

4.2 Study Definition and Design

We set out to empirically evaluate the impact of Local Database Proxy, Local Sharding-Based Router, Priority Message Queue, Competing Consumers, Gatekeeper and Pipes and Filters on the energy consumption of cloud-based applications. We select these cloud patterns because they are used in previous studies (Hecht et al., 2014; Tudorica and Bucur, 2011) (allowing us to compare our results with these previous works), and because they are described as good design practices by both academic and practitioners. This section presents our research questions, and describes the two experimental setups of our study, as well as our design and procedure.

4.2.1 Research Questions

Our study aims to answer the following research questions:

(RQ1) Does the implementation of Cloud patterns affect the energy consumption of cloud-based applications?

(RQ2) Do interactions among patterns affect the energy consumption of cloud-based applications?

4.2.2 Experimental Setup

To answer these research questions, we conducted the following two experiments.

Experiment 1

A multi-threaded distributed application, which communicates through REST calls was implemented and deployed on a GlassFish 4 application server. We chose MySQL as the database management system because it is one of the most popular databases for Cloud applications. Sakila database [1] provided by MySQL is used as it is initially developed to provide a standard schema, containing a large number of records, making it interesting for experimentations. Sakila also serves to highlight the latest features of MySQL such as Views, Stored Procedures, and Triggers, and it is consistent with existing databases. The test application was fully developed with the Java Development Kit 1.7 and it is composed of about 3,800 lines of code and its size is 6 MB. The master node has the following characteristics: 2 virtual processors (CPU: Intel Xeon X5650) with 8GB RAM and 40GB disk space. This node is a virtual machine of a server located on a separate network. We have 8 slave database nodes: 4 on one server, each one has a virtual processor (CPU: Intel QuadCore i5) with 1 GB RAM and 24 GB disk space. The other four database nodes are on a second server with the following characteristics: each Virtual Machine has one virtual processor (CPU: Intel Core 2 Duo), 1 GB RAM and 24 GB disk space. All the hardware is connected on a private network behind a switch. All the virtual machines are running on VMware ESXi and all the servers are running Ubuntu 14.04 LTS 64-bit as operating system. The architectural design of Experiment 1 is shown in Figure 4.1.

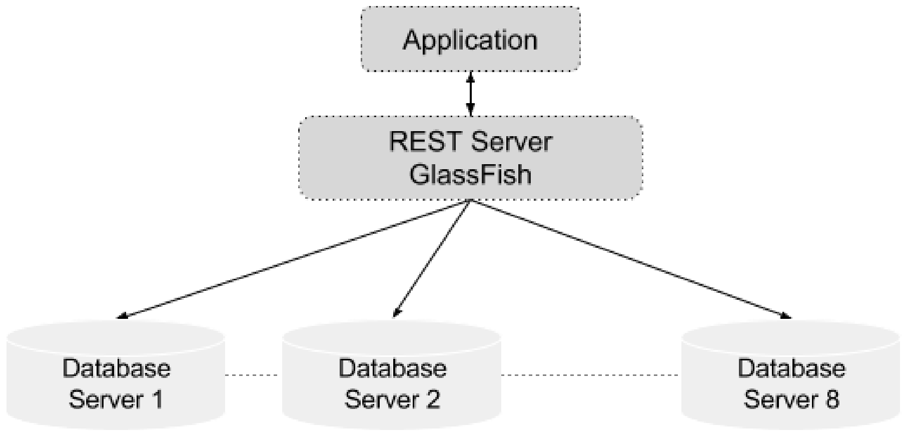


Figure 4.1 Architectural Design of the First Experiment

[1]http://dev.mysql.com/doc/sakila/en/

A scenario was designed in which the client is a thread generated on the client side of our cloud-based application. This client establishes a connection to the server, and then performs a series of actions in a certain amount of time (see Figure 4.2). Each client sends *100 select requests* at the peak of the scenario workload, and we measure the response time of the application at this point by taking the average of the response time to all clients. This will represent the performance of the application.

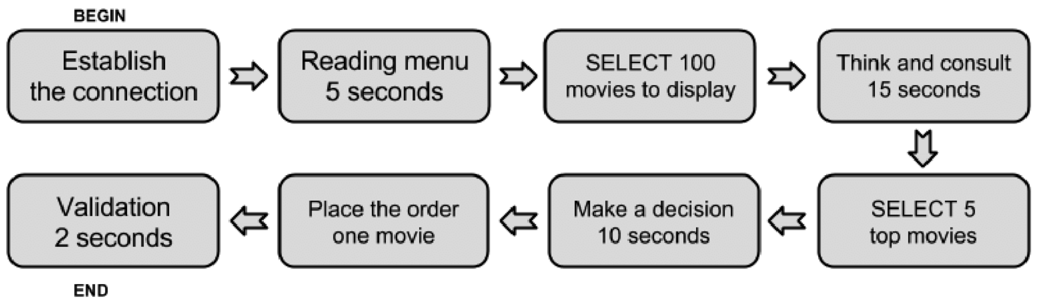

BEGIN
Establish the connection
Reading menu 5 seconds
SELECT 100 movies to display
Think and consult 15 seconds
SELECT 5 top movies
Make a decision 10 seconds
Place the order one movie
Validation 2 seconds
END

Figure 4.2 First Experiment Test Scenario

We repeat this scenario using different numbers of clients. The requests performed by the clients are: *select* or *write* to *display* or *place the order* respectively. The application generates concurrent threads to simulate clients propagating these requests. The number of clients used in our experiments are: 100, 250, 500, 1000 and 1500. Power-API is located on the database node and measures the amount of energy consumed by the MySQL process. Because of the variability observed during multiple executions of an application (caused for example by optimization mechanisms like caching), we repeated each experiment five times and took the average.

Experiment 2

Our second application was implemented using Python Flask microframework. This micro web framework is highly scalable and extensible. Cloud computing services are commonly multi-language in the real world, however, Java and Python are two most often used programming languages in Cloud applications. Therefore, we set up our second experiment using Python in order to achieve diversity and compare results for two different programming languages. The application reports the average response time of clients by measuring the response time to deliver results to each client during the execution. It includes 17 modules and 7 microservices (the list of microservices is presented in Table 4.1) that are communicating with each other using REST web services. 10 stored procedures were designed to make the response when querying the database machines.

The microservices architectural style enables developing a single application as a suite of small services, each running in its own process, having its own data storage. Microservices are communicating with lightweight mechanisms, often an HTTP resource API and they are built around business capabilities. Moreover, the deployment of microservices can be fully automated and the services can be deployed independently. In order to fully benefit from the microservice architectural style, each service should be elastic, resilient, composable, minimal, and complete. Monolithic applications, on the other hand, are built on a single unit, thus any changes to the system involves building and deploying a new version of the server-side application. Change cycles are tied together, meaning that a change made to a small part of the application requires the entire monolith application to be rebuilt and deployed.

Over time it can become hard to maintain a good modular structure in monolithic applications, making it harder to keep changes that ought to only affect one module within that module. Scaling requires scaling of the entire application rather than parts of it, which consumes more resources. In this work, we have implemented both a monolithic and a microservice version for our application.

The application was deployed on 15 virtual machines, i.e., **m3.xlarge** instances from Amazon EC2 which provides a balance of compute, memory, and network resources. Each instance had 4 CPU cores with 15 Gb memory available and 2 SSD 40 Gb hard disks running Ubuntu 14.04 LTS 64-bit. Requests were sent from an application machine simulating the behavior of clients to the interconnected virtual instances, through REST services. Microservices were deployed on separate instances to simulate the communication of microservices through a network. As for the monolithic version of the application, the clients requests were processed sequentially in one separate module. In a microservice architecture, there is no clear leading service that drives the rest of the services. In other words, each service module of our application controls its business function and data, and they all have an equal contribution to the application. However, the microservice 1 of the application had to be called constantly by the other services since this service was in charge of security checks and was responsible for providing authorizations to clients. The architectural design of Experiment 2 is shown in Figure 4.3.

The Northwind sample database[2] was chosen for the second experiment. We choose this database because of its rich set of functionalities. The Northwind database is about a company named "Northwind Traders". The database includes all the sales transactions that

[2]https://northwinddatabase.codeplex.com/

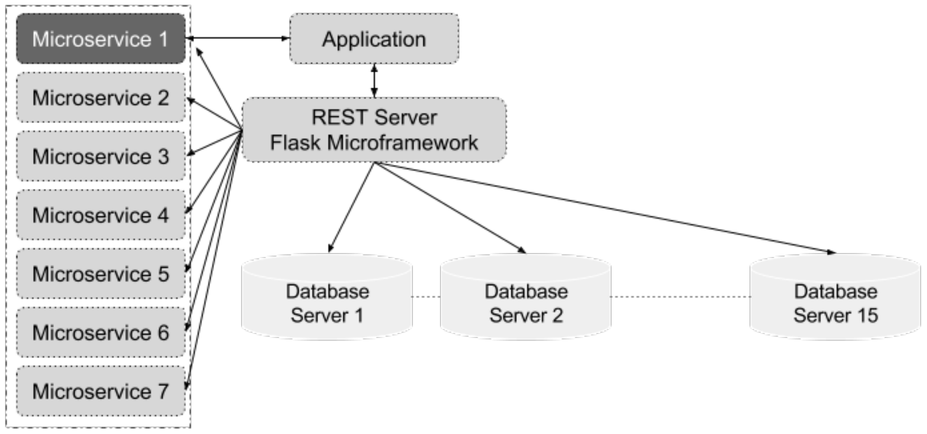


Figure 4.3 Architectural Design of the Second Experiment

occured between the company i.e., Northwind traders and its customers as well as the purchase transactions between Northwind and its suppliers. We installed a MySQL version[3] of this database to conduct our experiment. All the information contained in this database is anonymized. All the views, triggers and stored procedures used in this experiment were written from scratch for the purpose of the experiment.

Table 4.1 List of Microservices and their Tasks

Number	**Microservice Name**	**Microservice Task**
1	pf_micro1	Login service
2	pf_micro2	Searching a customer and display information
3	pf_micro3	List products
4	pf_micro4	View customer shipping information
5	pf_micro5	Add/Edit shipping information
6	pf_micro6	Preview order and calculate subtotal
7	pf_micro7	Submit an order

The application represents a typical SaaS application that simulates sales representatives of the Northwind Trading Company using a Pocket PC to take orders from customers and synchronizing the new orders back to the database concurrently. The scenario is depicted in Figure 4.4. Similar to Experiment 1, we repeated this scenario using different number of clients. The requests performed by the clients are: *select* or *write* to *display* or *place the order* respectively. The application generates concurrent threads on the server side, and multiprocessing in the client side to simulate clients propagating their requests.

[3]http://github.com/dalers/mywind

The number of clients used in our experiments are: 100, 250, 500, 1000, 1500, 2000, 2500 and 3000. The Power-API is located on the database node and REST server nodes, and measures the amount of energy consumed by MySQL processes and the REST server accepting REST calls. Because of the variability observed during multiple executions of an application, we repeated each experiment five times and took the average.

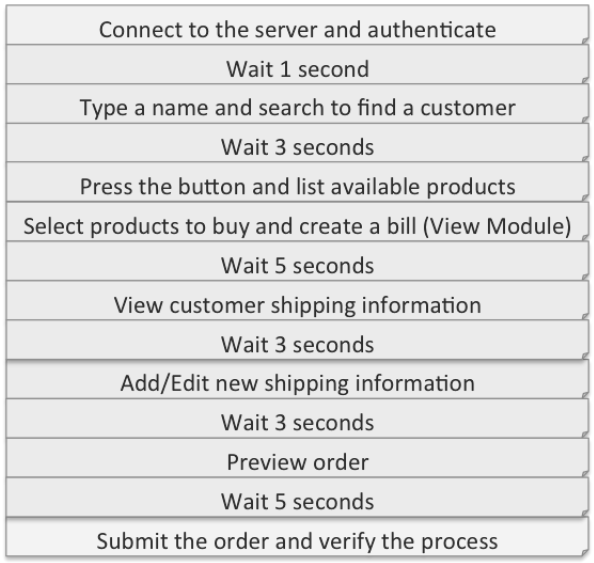

Connect to the server and authenticate
Wait 1 second
Type a name and search to find a customer
Wait 3 seconds
Press the button and list available products
Select products to buy and create a bill (View Module)
Wait 5 seconds
View customer shipping information
Wait 3 seconds
Add/Edit new shipping information
Wait 3 seconds
Preview order
Wait 5 seconds
Submit the order and verify the process

Figure 4.4 Second Experiment Test Scenario

To better understand the impact of combinations of Cloud patterns, we designed two approaches (a Lightweight and a Heavyweight approach) mimicking the behavior of ordinary customers and VIPs passing orders:

- Lightweight approach (Simple queries / small sized company customer):

- MQ + Proxy
- MQ + Sharding
- Sharding + Pipes & Filters

- Heavyweight approach (Complex queries / VIP company customer):

- Sharding + Competing Consumers + Pipes & Filters
- Sharding + Competing Consumers + Gatekeeper + Pipes & Filters

4.2.3 Implementation of the Patterns

In order to evaluate the benefits and the trade-offs between the Local Database Proxy, the Local Sharding-Based Router and the Priority Message Queue design patterns, we implemented these patterns in the applications described in Section 4.2.2 and examined them through the mentioned scenarios. The NoProxy/NoSharding version E0 does not use any pattern. Versions E1 to E3 implement Local Database Proxy with Random Allocation, Round-Robin and a Custom load balancing algorithm. Versions E4 to E6 correspond to Local Sharding-Based Router with three sharding algorithms: Modulo, Lookup and Consistent Hashing. Two different implementations of the Gatekeeper pattern have two version E7 for the decomposed architecture with microservices employed to communicate and deliver messages and E8, that is the monolith and performs all the tasks in a series of orders. The Competing Consumers pattern also has the two version E9 and E10, which refer to the microservices and monolithic style of this pattern respectively. The Pipes and Filter pattern has only the microservices architecture style marked with E11 version in our experiments.

Version E12 implements the Priority Message Queue pattern. Since there is only one implementation of Priority Message Queue pattern in this study, we investigate its impact on energy consumption only in combination with the other patterns. Table 4.2 summarizes the different versions of the applications that were produced.

Table 4.2 Patterns chosen for the experiments

Pattern	**Abbreviation**	**Code**
No Proxy, direct hit	PRX-NO	E0
Proxy Random	PRX-RND	E1
Proxy Round-Robin	PRX-RR	E2
Proxy Custom	PRX-CU	E3
Sharding LookUp	SHRD-LU	E4
Sharding Modulo	SHRD-MD	E5
Sharding Consistent	SHRD-CN	E6
Gatekeeper with Microservices	GK-Micro	E7
Gatekeeper Monolithic	GK-Mono	E8
Competing Consumers with Microservices	CCP-Micro	E9
Competing Consumers Monolithic	CCP-Mono	E10
Pipes and Filters	P&F	E11
Priority Message Queue	MQ	E12

To ensure lower variance between maximum and average values and hence increase the precision of our energy measurements, we eliminated the values of the first and last executions. In total, our application was able to simulate a maximum of 150,000 concurrent requests, enabling us to establish cloud-based applications for our experiment that are capable of responding to thousands of coincidental requests from clients. This level of load is reflective of real-world cloud applications.

In our study, each experiment is independent with regard to others, and simulations were terminated at a fixed time. Even large requests and heavy loads never lasted more than a certain amount of time predefined as the maximum, which is *180 seconds* for the first application and *75 seconds* for the second one. In the following, we explain in details the specific algorithms that were implemented in each pattern.

Local Database Proxy: Three implementations of this pattern were considered in our research; Random allocation strategy, Round-Robin allocation strategy, and Custom Load balancing strategy. The proxy is placed between the server and the clients. The basic approach, *NoProxy* REST web service, exposes a set of methods that are hitting the database directly without load balancing. This method has been implemented to test the local database proxy pattern. It is the baseline used to compare the results of our proxy implementations. The queries are built using parameters such as the ID of a *select* passed over the REST call from each client (thread) concurrently during each scenario.

The random approach is implemented by choosing randomly an instance from the pool. The round-robin chooses the next instance that has not yet been used in the “round", *i.e.* the first, then the second, then the third,. . . , finally the first and so on. The custom algorithm is more reactive, and it uses two metrics to evaluate the best slave node to pick: the ping response time between the server and slave, and the number of active connections on the slaves. A thread is started every 500 ms with the purpose of monitoring such metrics. After choosing the corresponding slave, the query is executed and the result is sent back to the function that was called. To simplify the tests, only IDs (number identifiers) were sent back, so there was no need to serialize any data. Eventually, if the result is null, the response sent to the client has the http *no content* status. Otherwise, the result is sent back to the client using the http *ok response* status.

Local Sharding-Based Router: To test this pattern, we used multiple shards hosted separately. Each shard had the same database schema and structure as suggested by Sharding Algorithms[4]. Two tables of a modified version of the Sakila database were used. All the relationships in both the “rental" and “film" tables were removed since the sharding is adapted only for independent data.

Three commonly known sharding algorithms were studied in our research: Modulo algorithm, Look-up algorithm and the Consistent Hashing algorithm. The modulo algorithm divides the request primary key by the number of running shards, the remainder is the number of the server which will handle the request. The second sharding algorithm used is the Look-up strategy. This algorithm implements an array with a larger amount of elements than the number of server nodes available. References to the server node are randomly placed in this array such that every node receives the same share of slots. To determine which node should be used, the key is divided by the number of slots and the remainder is used as index in the array. The third sharding algorithm used is the Consistent Hashing. For each request, a value is computed for each node. This value is composed of the hash of the key and the node. Then, the server with the longest hash value processes the request. The hash algorithms recommended for this sharding algorithm are MD5 and SHA-1.

Priority Message Queue: Requests are annotated with different priority numbers and sent in the priority message queue of our test application. All requests are ordered according to their priority and processed by the database services in this order.

Competing Consumers: This pattern puts the request of each client into a message queue, then delivers the requests to the database by a simple mechanism to preserve the load balance: The requests include a hash ID of the priority, which will be converted to a *bigInt* number and is divided by the number of instances and the remainder is the server to hit. We have conducted a series of experiments to confirm that this approach maintains load balancing among the clients queries.

The Gatekeeper: our set up for this pattern consists of 5 gatekeepers, 5 trusted hosts and 5 databases. Each query from a client is being redirected to the trusted host if it passes the security check. The security process tries to find similarity between the query and 10 predefined SQL injection rules. If a match is found, the query will be discarded, otherwise the query will be transferred to the corresponding trusted host, by using the hash ID of the

[4]http://kennethxu.blogspot.fr/2012/11/sharding-algorithm.html

message and turning it into a *bigInt* number and dividing it by 5 to obtain the remainder (i.e., the same approach as for the competing consumers pattern) to ensure load balancing. Then, the selected trusted host will query the database and re-transfer the results to the gatekeeper and, ultimately, this result will be delivered back to the client. The security mechanism will take place for every query, no matter the priority and the sender identity. However, if the query was suspicious and considered harmful, the gatekeeper would block the client, and the clients identification (including the query and IP address) will be saved in a black list. The gatekeeper will reject any further queries sent by this specific client and provide a message explaining the reason why the client cannot access the requested data. Since SQL injection rules are subject to changes, we opted in this work for simple rules, to demonstrate the behavior of this pattern. Table 4.3 summarizes the SQL injection rules applied to this pattern in our study.

Table 4.3 SQL injection rules applied in the Gatekeeper pattern

Type	**Rule Manifest**
type-1	SELECT col1 FROM table1 WHERE col1 > 1 OR 1=1; The where clause has been short-circuited by the addition of a line such as ‘a’=’a’ or 1=1
type-2	SELECT col1 FROM table1 WHERE col1 > 1 AND 1=2; The where clause has been truncated with a comment
type-3	SELECT col1 FROM table1 WHERE col1 > 1; The addition of a union has enabled the reading of a second table or view
type-4	SELECT col1 FROM table1 WHERE col1 > 1; UPDATE table2 set col1=1; Stacking Queries, executing more than one query in one transaction
type-5	SELECT col1 FROM table1 WHERE col1 > 1 UNION SELECT col2 FROM table2; An unintentional SQL statement has been added
type-6	SELECT col1 FROM table1 WHERE col1 > 1; drop table t1; SQL where an unintentional sub-select has been added
type-7	SELECT fieldlist FROM table WHERE field = ’x’ AND email IS NULL; –’; Schema mapping

Pipes and Filters: The objective of this pattern is to decompose a task that performs complex processing into a series of discrete elements that can be reused. This pattern can improve performance, scalability, and reusability by allowing task elements that perform the processing to be deployed and scaled independently. This pattern was deployed by implementing a pipeline and using a message queue. In the monolithic version, tasks are just series of REST calls performed one by one through the execution, which is time consuming and a single change might result in change in other services. On the other hand, in the case of microservices, the application decomposes the tasks into several interconnected microservices communicating by REST calls.

Combined Patterns: The combination of patterns was driven by the functionalities of the application. For example, when Proxy/Sharding is combined with Message Queue, this means that the application uses the message queue to prioritize queries, then hits the database using Proxy/Sharding pattern implementations. In our light-weight approach, the Pipes and Filters pattern has been applied to decompose the request into microservices, then the Sharding pattern is used to access the databases.

In total, twelve versions of the application were analyzed and summarized in Table 4.2. To collect the energy measurements required to answer our research questions, a series of scenarios were executed. These scenarios were designed specifically to simulate the characteristics of a real-world cloud-based application.

4.2.4 Hypotheses

To answer our research questions, we formulate the following null hypotheses, where E0, Ex ($x \in \{1 \ldots 11\}$), and E12 are the different versions of the applications, as described in Table 4.2:

- H_x^1 : There is no difference between the average amount of energy consumed by design Ex and design E0.
- H_{x12}^1 : The average amount of energy consumed by the combination of designs Ex ($x \in \{1 \ldots 6\}$) and E12 is not different from the average amount of energy consumed by each design taken separately.
- H_{x11}^1 : The average amount of energy consumed by the combination of designs Ex ($x \in \{4 \ldots 6\}$) and E11 is not different from the average amount of energy consumed by each design taken separately.
- H_{x9}^1 : The average amount of energy consumed by the combination of designs Ex ($x \in \{4 \ldots 6\}$) and E11 and E9 is not different from the average amount of energy consumed by each design taken separately.
- H_{x7}^1 : The average amount of energy consumed by the combination of designs Ex ($x \in \{4 \ldots 6\}$) and E11 and E9 and E7 is not different from the average amount of energy consumed by each design taken separately.

To better understand the trade-offs between the energy consumption and the performance, we formulate the following null hypotheses:

- H^2_x : There is no difference between the average response time by design Ex and design E0.
- H^2_{x12} : The average response time of the combination of designs Ex ($x \in \{1 \dots 11\}$) and E12 is not different from the average response time of each design taken separately.
- H^2_{x11} : The average response time of the combination of designs Ex ($x \in \{4 \dots 6\}$) and E11 is not different from the average response time of each design taken separately.
- H^2_{x9} : The average response time of the combination of designs Ex ($x \in \{4 \dots 6\}$) and E11 and E9 is not different from the average response time of each design taken separately.
- H^2_{x7} : The average response time of the combination of designs Ex ($x \in \{4 \dots 6\}$) and E11 and E9 and E7 is not different from the average response time of each design taken separately.

4.2.5 Analysis Method

Since the observations of each scenario were independent of those of the other scenarios, we perform the Mann-Whitney U test (Sheskin, 2003) to test H^1_x, H^2_x, H^1_{x7}, H^2_{x7}. Moreover, we computed the Cliff's δ effect size (Romano et al., 2006) to quantify the importance of the difference between the metric values. Cliff's δ effect size is reported to be more robust and reliable than the Cohen's d effect size (Cohen, 1977). We perform all our tests using a 95% confidence level (*i.e.* p-value < 0.05). The Mann-Whitney U test is a non-parametric statistical test that examines whether two independent distributions are the same or if one distribution tends to have higher values. Non-parametric statistical tests make no assumptions about the distributions of the metrics. Cliff's δ is a non-parametric effect size measure which represents the degree of overlap between two sample distributions (Romano et al., 2006). Cliff's δ effect size ranges from -1 (if all selected values in the first group are larger than the second group) to +1 (if all selected values in the first group are smaller than the second group), and it is zero when two sample distributions are identical (Cliff, 1993). A Cliff's δ effect size is considered negligible if it is < 0.147, small if it is < 0.33, medium if it is < 0.474, and large if it is $>= 0.474$.

4.2.6 Independent Variables

The Local Database Proxy, Local Sharding-Based Router, Priority Message Queue, Competing Consumers, The Gatekeeper and Pipes and Filters patterns, as well as the algorithms presented in Table 4.2 are the independent variables of our study.

4.2.7 Dependent Variables

The dependent variables measure the quality of service in terms of response time to select queries dispatched by the clients and the energy consumption measured by Power-API during each scenario. The result is a two-dimensional comparison between response time and the amount of energy consumed. The response time is measured in nanoseconds and then converted to milliseconds. We choose this metric because it reflects the capacity of the applications to scale with the number of clients at peak, with maximum number of requests.

The other metric is the power consumption provided by Power-API in watts, which is converted to kilojoules(kJ) using the equation:
$E_{kJ} = (P_{watt} \text{ x } t_s)/\ 1000$.

CHAPTER 5 THE IMPACT OF CLOUD PATTERNS ON PERFORMANCE AND ENERGY CONSUMPTION

This chapter presents and discusses the results of our two experiments, answering the research questions described in Section 4. The chapter is organized in two sections corresponding to our two experiments.

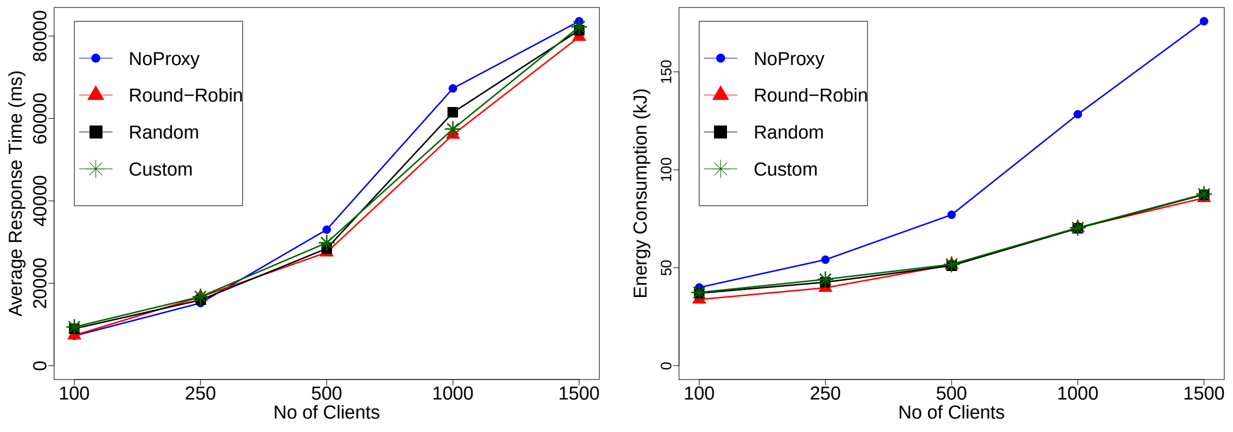
5.1 Results of Experiment 1

We measured the performance in terms of average response time, and energy consumption (using Power-API tool) in all databases throughout the whole simulation scenarios. All the numbers where averaged over 5 different runs to ensure the accuracy and avoid chances of large variance. Table 5.1 summarizes the results of Mann–Whitney U tests and Cliff's δ effect sizes for the energy consumption and the response time for individual and combined patterns. We marked significant results in bold.

Table 5.1 p-value of Mann–Whitney U test and Cliff's δ effect size for the first application

	Average Response Time		Average Energy Consumption	
	p-value	Effect Size	p-value	Effect Size
E0, E1	0.87	0.04	<**0.05**	0.44
E0, E2	0.77	0.06	<**0.05**	0.49
E0, E3	0.87	0.04	<**0.05**	0.44
E0, E4	<**0.05**	-0.68	0.36	-0.2
E0, E5	<**0.05**	-0.6	0.74	0.07
E0, E6	<**0.05**	-0.6	<**0.05**	0.42
E1, E1 + E12	0.38	0.19	0.53	-0.13
E2, E2 + E12	0.43	0.17	0.20	-0.28
E3, E3 + E12	0.51	0.14	0.53	-0.13
E4, E4 + E12	<**0.05**	0.68	0.2	-0.28
E5, E5 + E12	<**0.05**	0.52	0.2	-0.28
E6, E6 + E12	<**0.05**	0.49	<**0.05**	-0.6
E4, E3 + E4	<**0.05**	0.76	0.36	-0.2
E6, E2 + E6	<**0.05**	0.6	0.36	-0.2

Average amount of consumed energy: results presented in Table 5.1 show that there is a statistically significant difference between the average amount of energy consumed by an application implementing the Local Database Proxy pattern and an application not implementing this pattern. The effect size is large in almost all cases. Therefore, we reject H_x^1 for all Ex ($x \in \{1 \ldots 3\}$). Regarding the Local Sharding-Based Router pattern, except for the case where the pattern is implemented using the consistent hashing algorithm, the difference between the amount of energy consumed by an application implementing the pattern and another application that did not implemented the pattern is not significantly different. In other words, only consistent hashing tends to consume (to some extent) less energy than no Sharding strategy. However, the effect size is medium. Therefore, we cannot reject H_x^1 for all Ex ($x \in \{4 \ldots 6\}$).

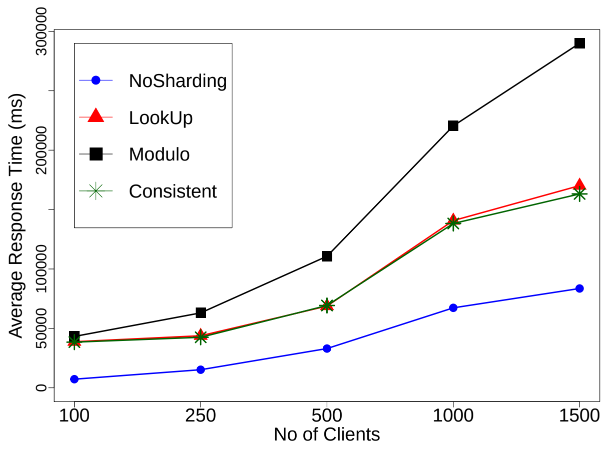


a) *Average Response Time* **b)** *Energy Consumption*

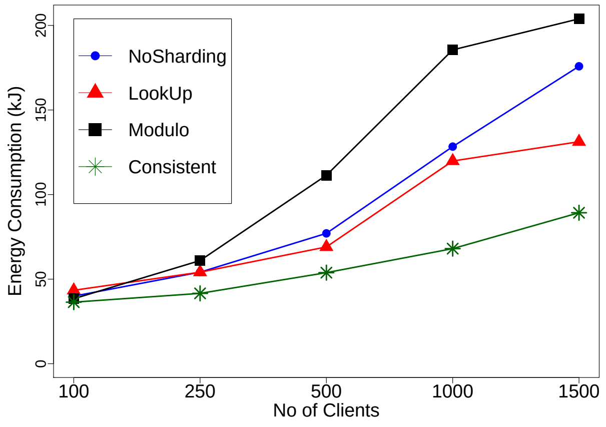
Figure 5.1 The Impact of Proxy Pattern

Our results show that any implementation of the Local Database Proxy pattern can significantly improve the energy efficiency of an application, while the Local Sharding-Based Router pattern has little effect on energy consumption. Figure 5.1 and Figure 5.2 summarize the results obtained for all the implementations of the two patterns.

Average response time: results from Table 5.1 show that there is a statistically significant difference between the average response time of an application implementing the Local Sharding-Based Router pattern and an application not implementing this pattern. Hence, we reject H_x^2 for all Ex ($x \in \{4 \ldots 6\}$). In fact, as shown on Figure 5.2, all the implemen-



a) *Average Response Time*



b) *Energy Consumption*

Figure 5.2 The Impact of Sharding Pattern

tations of the Local Sharding-Based Router pattern have a negative impact on the response time of the applications (*i.e.* the average response time is increased). Among the different implementations of the Local Sharding-Based Router pattern, the Modulo algorithm has the most negative impact on the response time. We explain this result by the randomness of this algorithm. However, more observations and tests are required to confirm our claim.

Regarding the Local Database Proxy pattern, there is no statistically significant difference between the response time of applications using any version of the pattern and applications not using the pattern. However, Figure 5.1, as well as effect size values, show that Local Database Proxy pattern has a (small) positive impact on the response time of the applications. Yet, we cannot reject H_x^2 for ($x \in \{1 \ldots 3\}$).

Combination of Patterns: Table 5.1 and Figure 5.3 show that there is no statistically significant difference between the response time of applications implementing the Local Database Proxy pattern and applications implementing a combination of Local Database Proxy and Priority Message Queue patterns. Consequently, we cannot reject H_{x12}^2 for all Ex ($x \in \{1 \ldots 3\}$).

Regarding the combination of the Local Sharding-Based Router pattern and the Priority Message Queue pattern, results show that it can reduce an application's response time. Statistical tests show that there is a significant difference, regardless of the type of algorithm, between the response time of applications implementing the Sharding pattern and application implementing a combination of Sharding and Message queue patterns (see Figure 5.4).

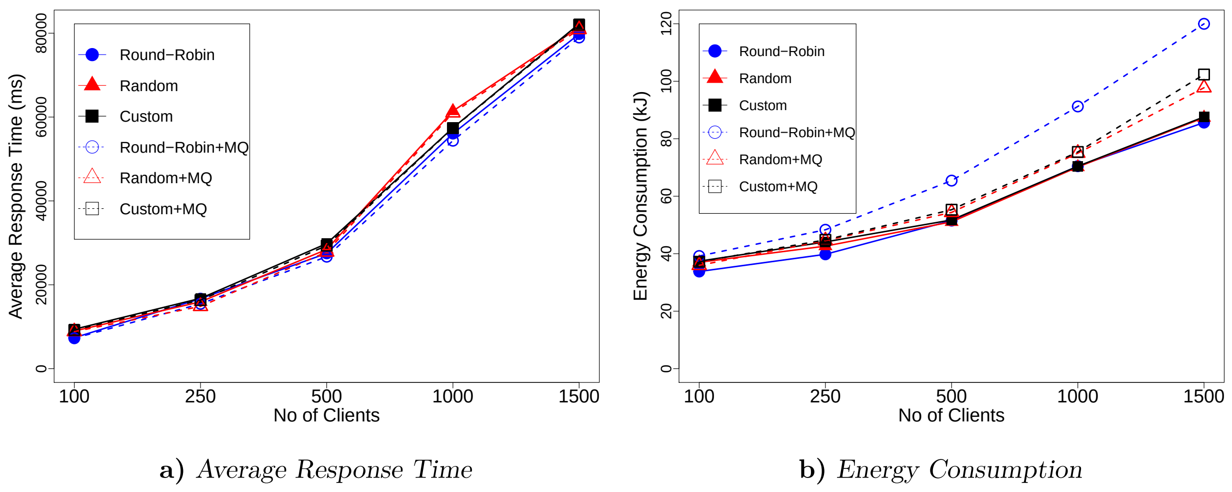


Figure 5.3 The Impact of Proxy Pattern Combined with Message Queue Pattern

The effect size is large in all three cases as shown on Table 5.1. Consequently, we reject H^2_{x7} for all Ex ($x \in \{4 \ldots 6\}$).

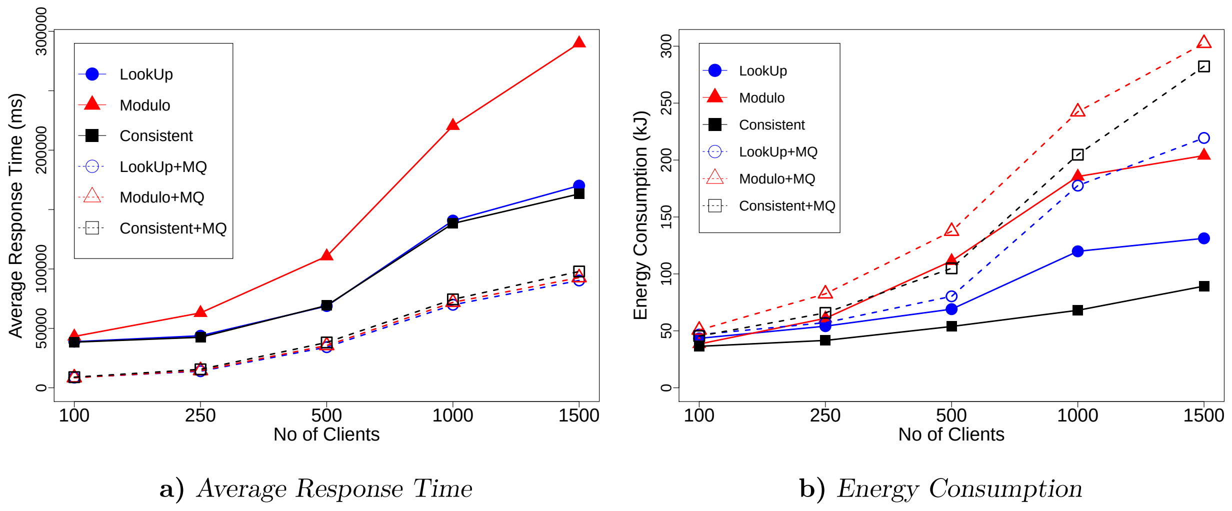
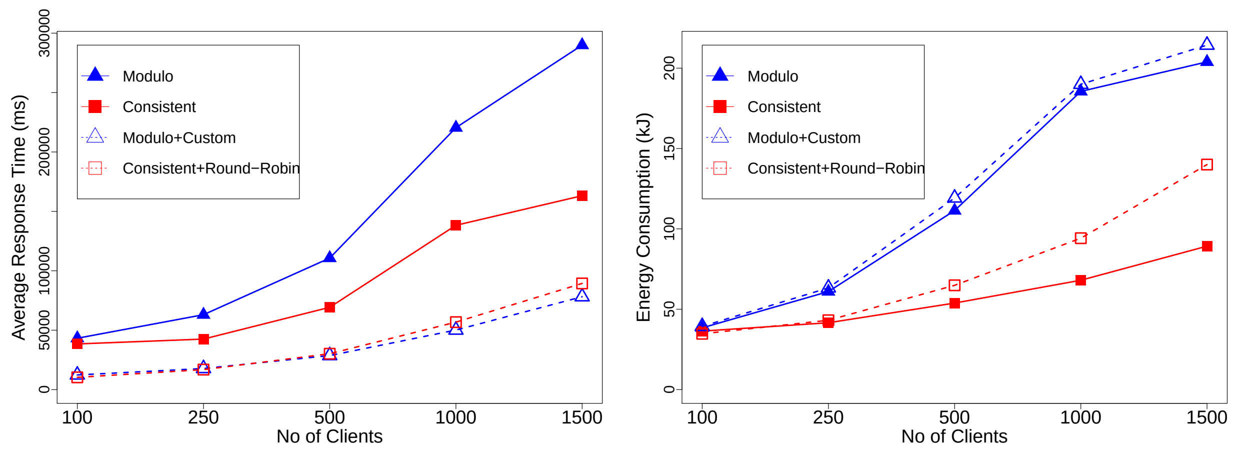


Figure 5.4 The Impact of Sharding Pattern Combined with Message Queue Pattern

Results from Table 5.1 show that Local Database Proxy combined with Priority Message Queue does not significantly increase the amount of energy consumed by an application (see Figure 5.3). Therefore, we cannot reject H^1_{x7} for all Ex ($x \in \{1 \ldots 3\}$). However, when the Local Sharding-Based Router pattern is combined with the Priority Message Queue, the consistent hashing algorithm can impact energy efficiency negatively. In fact, as shown on Figure 5.4, when the Priority Message Queue pattern is combined with the Local Sharding-Based Router pattern implemented using the consistent hashing algorithm, the resulting application consumes more energy. Hence, we reject H^1_{x7} for E_6, but not H^1_{x7} for E_4 and E_5.

When the Local Sharding-Based Router pattern is combined with Local Database Proxy, the average response time of the application decreases significantly (see Figure 5.5 and Table 5.1). Conversely, statistical tests from Table 5.1 and trends on Figure 5.5 show that Local Sharding-Based Router pattern combined with Local Database Proxy pattern has no significant impact on the energy consumption of the application.

When the Local Database Proxy pattern is combined with a Message Queue, the average response time of the application decreases slightly, but not significantly, as shown on Figure 5.3. However, when the Local Sharding-Based Router pattern is combined with a Message Queue, the response time of the application improves significantly (see the Table 5.1). The effect sizes are greater than 0.6, for all three Sharding algorithms.



a) *Average Response Time* **b)** *Energy Consumption*

Figure 5.5 The Impact of Sharding Pattern Combined with Proxy Pattern

Summary: Combining the Priority Message Queue pattern with Local Database Proxy has no significant impact neither on application response time, nor on the average amount of energy consumed by the application. On the contrary, the combination of Priority Message Queue pattern and Sharding-Based Router pattern can improve the response time of an application experiencing heavy loads of read requests.

Besides, only the implementation of consistent hashing in Local Sharding-Based Router pattern can increase the energy consumption of the application. We conclude that although the Local Database Proxy pattern only has a small positive impact on the ability of applications to handle large number of requests of *read queries*, it can significantly improve the energy efficiency of a the application. Our first experiment shows that the Local Sharding-Based Router pattern when implemented using the consistent hashing strategy, can improve energy efficiency slightly in application for heavy *read requests.*

5.2 Results of Experiment 2

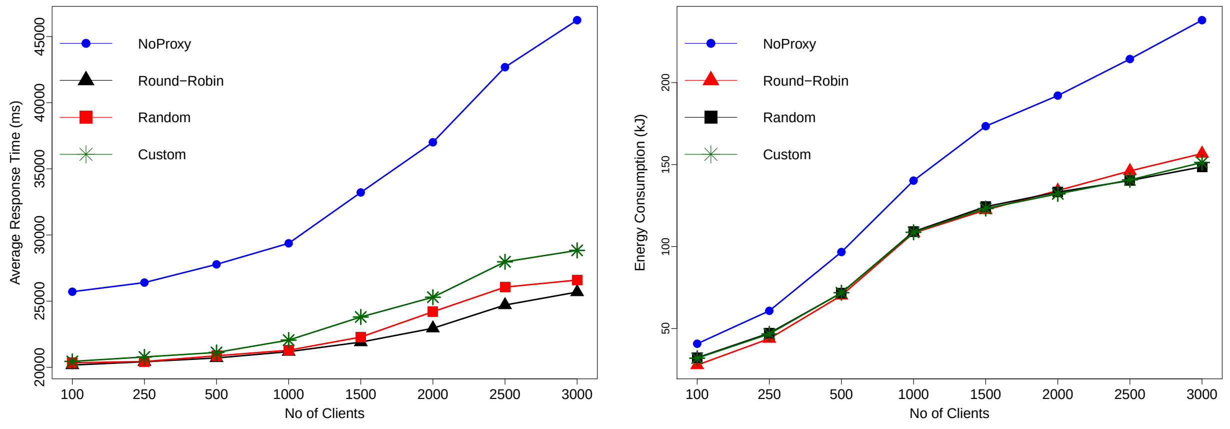
Second experiment was done by implementing six cloud patterns deployed in Amazon EC2 instances. The objective was, again, to measure the performance and energy consumption and compare the effectiveness of such patterns in a cloud-based application simulation. Table 5.2 summarizes the results of Mann–Whitney U tests and Cliff's δ effect sizes for the versions of our second application that implement a pattern. We marked significant results in bold.

Table 5.2 p-value of Mann–Whitney U test and Cliff's δ effect size for the second application, individual patterns

	Average Response Time		Average Energy Consumption	
	p-value	Effect Size	p-value	Effect Size
E0, E1	<**0.05**	0.95625	<**0.05**	0.3825
E0, E2	<**0.05**	0.94625	<**0.05**	0.3825
E0, E3	<**0.05**	0.84375	<**0.05**	0.3862
E0, E4	<**0.05**	0.45375	<**0.05**	0.3575
E0, E5	<**0.05**	0.40125	<**0.05**	0.315
E0, E6	<**0.05**	0.54	<**0.05**	0.39875
E0, E7	<**0.05**	0.40875	<**0.05**	0.4825
E0, E8	<**0.05**	-0.42875	0.1594	-0.18375
E0, E9	<**0.05**	0.5525	<**0.05**	0.455
E0, E10	<**0.05**	-0.55875	0.1094	-0.20875
E0, E11	<**0.05**	0.7125	<**0.05**	0.4725

Average response time is the average amount of time across all the clients to complete the scenario. The energy consumption as the previous experiment was measured in all databases using Power-API tool.

Average amount of consumed energy: Results presented in Table 5.2 show that there is a statistically significant difference between the average amount of energy consumed by an application implementing any of our six studied patterns and an application not implementing the patterns, except the monolithic version of gatekeeper and competing consumers patterns. The effect size is large in almost all cases, and it shows an improvement by reduction of the energy consumption. Therefore, we reject H_x^1 for all Ex ($x \in \{1 \ldots 7\}$), but not H_x^1 for E8 and E10. In other words, although the six patterns improve the energy efficiency of applications in general, when the Gatekeeper or the Competing consumers pattern are implemented in a monolithic application, the improvement of energy efficiency is negligible.

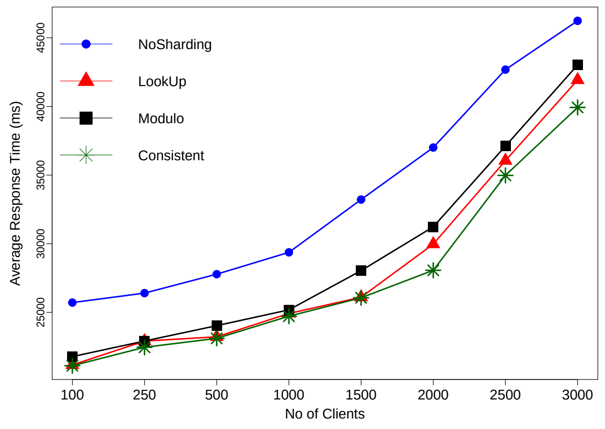


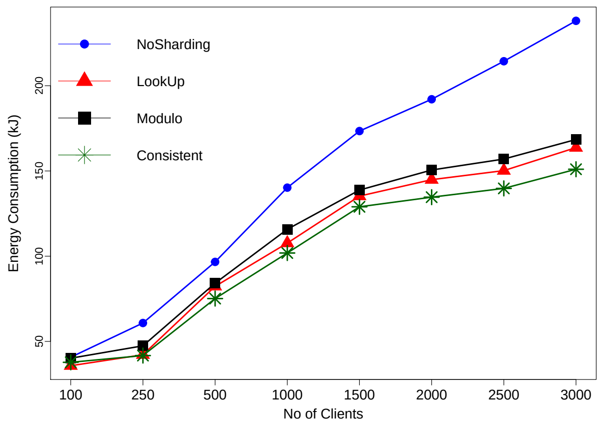
a) *Average Response Time* **b)** *Energy Consumption*

Figure 5.6 The Impact of Proxy Pattern

Average response time: Results presented in Table 5.2 show that, there is a statistically significant difference between the average response time of an application implementing any of our six studied patterns and an application not implementing the patterns. The effect size is large in almost all the cases. Therefore, we reject H_x^2 for all Ex ($x \in \{1 \ldots 11\}$). We recommend that developers interested in high performance, use these patterns during the development of their cloud-based application since they can significantly improve the response time of the application. Figures 5.6, 5.7, 5.8 and 5.9 show the improvements on response time, that can be achieved with the patterns.

In Figure 5.6, we can see that the Proxy pattern has a large impact on the average response time of the application. Moreover, the amount of energy consumed by the application is not significantly higher in comparison to other versions of the application, hence we conclude

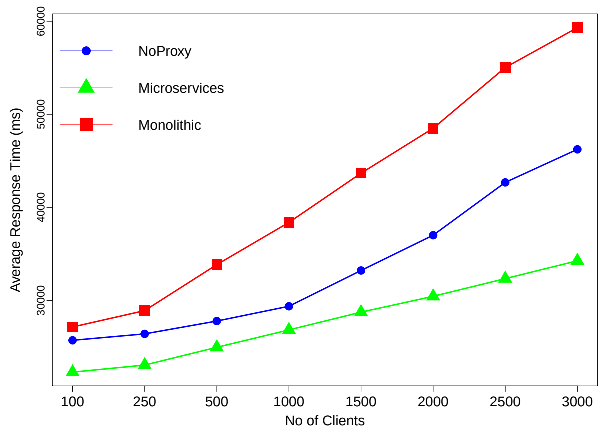


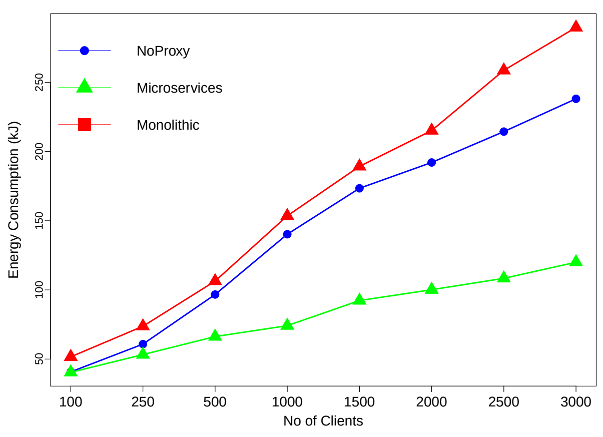


a) *Average Response Time* **b)** *Energy Consumption*

Figure 5.7 The Impact of Sharding Pattern

that the Proxy pattern can improve the performance of an application without sacrificing its energy efficiency. The Sharding pattern also improves both the performance (see Table 5.2) and the energy efficiency of applications (see Figure 5.7). However, the effect size is medium, i.e., 0.45375, 0.40125 and 0.54 respectively.

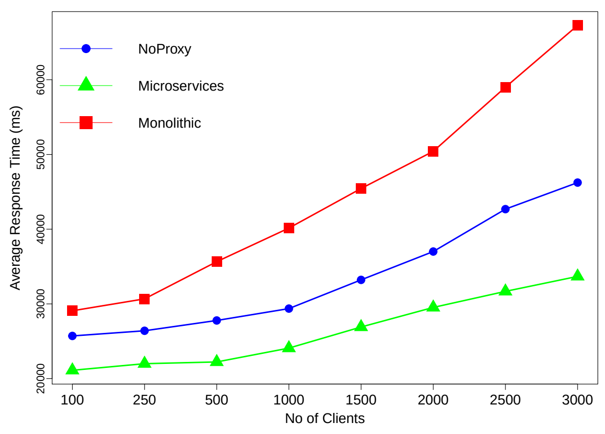




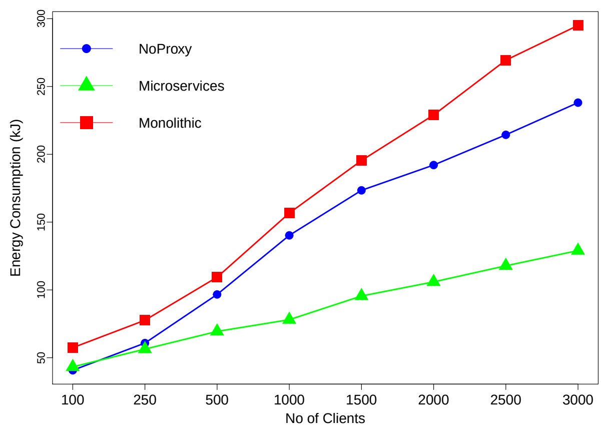
a) *Average Response Time* **b)** *Energy Consumption*

Figure 5.8 The Impact of Gatekeeper Pattern

The Gatekeeper pattern can improve both performance and energy consumption only when it is implemented in a microservices application (see Figure 5.8). This pattern is used to ensure security in cloud-based applications. When the pattern is implemented in a monolithic application, the amount of consumed energy increases significantly. Similar to Gatekeeper, the Competing Consumers pattern improves performance and energy efficiency only when the application follows a microservices architectural style (see Figure 5.9).

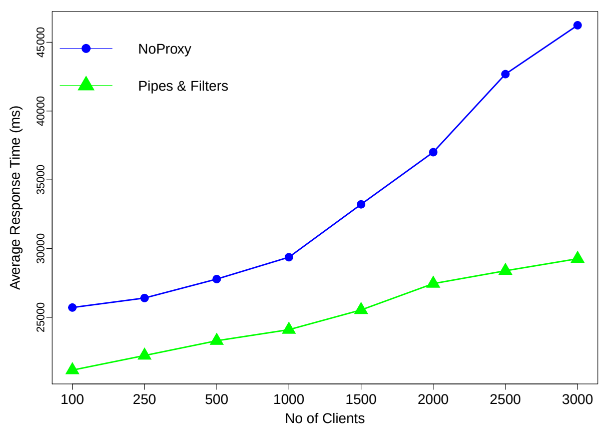


a) *Average Response Time*

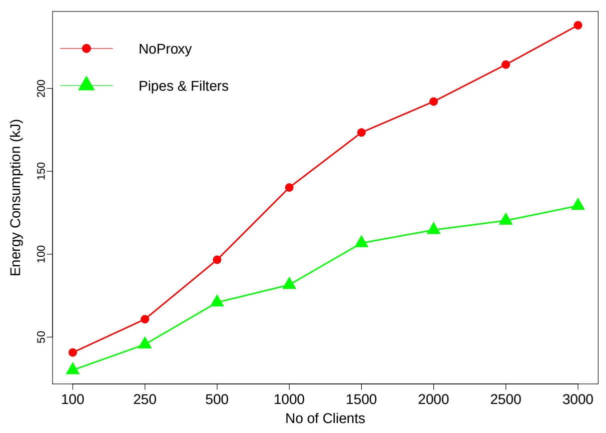


b) *Energy Consumption*

Figure 5.9 The Impact of Competing Consumers Pattern



a) *Average Response Time*



b) *Energy Consumption*

Figure 5.10 The Impact of Pipes & Filters Pattern

Pipes & Filters pattern can have a positive impact on both performance and energy consumption. Figure 5.10 shows the results obtained in our experiment. The average response time of the application decreases (the trend levels off when the number of clients increased) when the pattern is implemented. The energy consumed by the application is also lower in comparison with the version of the application with no pattern (p-value << 0.05, with an effect size of 0.7, see Table 5.2). We highly recommend that developers use this pattern since it significantly improves both response time and energy efficiency. Table 5.3 summarizes the results of Mann–Whitney U test and Cliff's δ effect sizes for the versions of our second application that implement a combination of patterns. We marked significant results in bold.

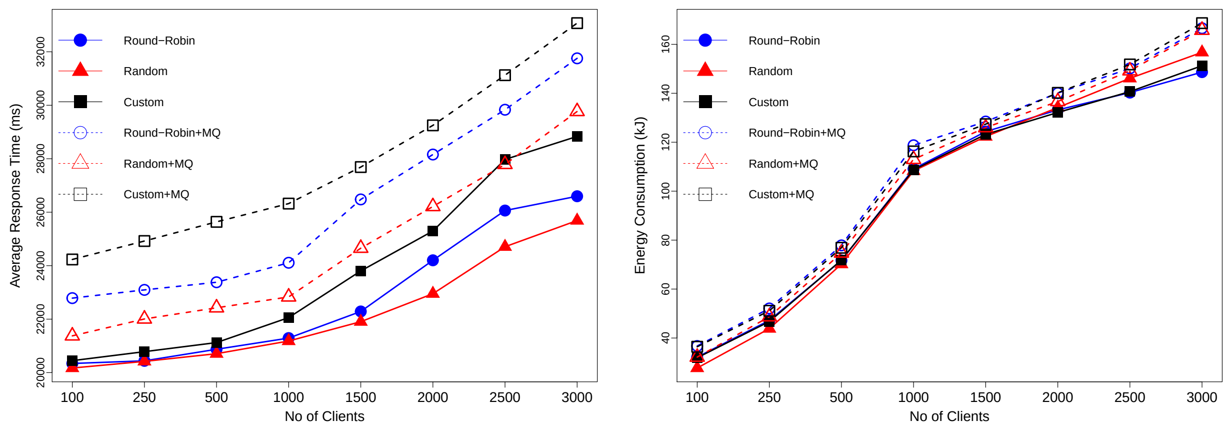
Combination of Patterns: When combining Proxy pattern with Message Queue pattern, no remarkable difference was observed on the energy utilization of the application (in comparison to the version with only the Proxy pattern). We see only a slight uptick on the energy consumption curve from Figure 5.11. The Mann–Whitney U tests from Table 5.3 show that the observed differences are not statistically significant (p-values are high: 0.6019, 0.3354 and 0.4875).

Table 5.3 p-value of Mann–Whitney U test and Cliff's δ effect size for the second application implementing a combination of patterns

	Average Response Time		Average Energy Consumption	
	p-value	Effect Size	p-value	Effect Size
E1, E1 + E12	<**0.05**	-0.49	0.6019	-0.06875
E2, E2 + E12	<**0.05**	-0.59375	0.3354	-0.12625
E3, E3 + E12	<**0.05**	-0.62375	0.4875	-0.09125
E4, E4 + E12	0.1869	0.43375	<**0.05**	-0.2575
E5, E5 + E12	<**0.05**	0.2425	<**0.05**	0.2675
E6, E6 + E12	0.3027	0.3125	<**0.05**	-0.3075
E4, E4 + E11	<**0.05**	0.52875	<**0.05**	0.2575
E5, E5 + E11	<**0.05**	0.28125	<**0.05**	-0.265
E6, E6 + E11	<**0.05**	0.57	<**0.05**	-0.2825
E4, E4 + E9 + E11	<**0.05**	-0.30125	<**0.05**	-0.26125
E5, E5 + E9 + E11	<**0.05**	-0.31125	<**0.05**	-0.29125
E6, E6 + E9 + E11	<**0.05**	-0.3575	<**0.05**	-0.3025
E4, E4 + E9 + E7 + E11	<**0.05**	-0.4925	<**0.05**	-0.37375
E5, E5 + E9 + E7 + E11	<**0.05**	-0.49375	<**0.05**	-0.3875
E6, E6 + E9 + E7 + E11	<**0.05**	-0.5175	<**0.05**	-0.4

On the contrary, joining the Proxy patten with Message Queue has a negative impact on the average response time (see the curves on Figure 5.11).

The Mann–Whitney U tests from Table 5.3 confirm that these observed differences are statistically significant. Hence, we conclude that the combination of Proxy and Message Queue patterns can have a negative impact on the performance of an application. The effect sizes vary from medium to large.

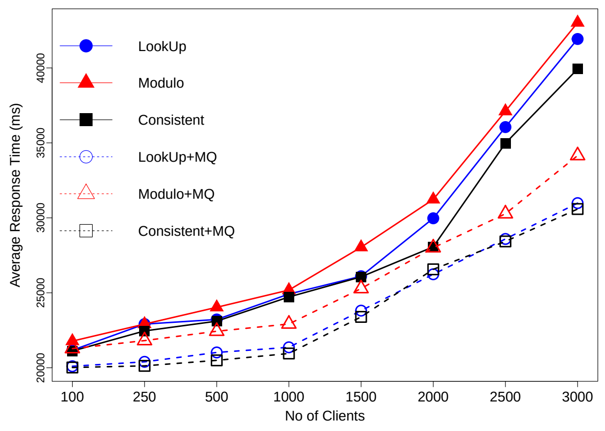


a) *Average Response Time* **b)** *Energy Consumption*

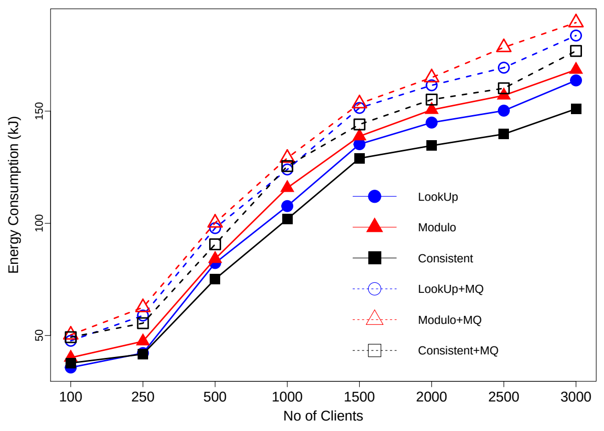
Figure 5.11 The Impact of Proxy Pattern Combined with Message Queue Pattern

Therefore, although the Proxy pattern can help improve the scalability of an application, a combination with the Message Queue pattern can result in a performance degradation. Regarding the Sharding pattern, a combination with Message Queue have a negative impact on energy efficiency (see Figure 5.12), but the effect size is not large in all cases. In fact, an slight difference observed on the energy consumption curves, nevertheless, the response time is improved when the Message Queue pattern is added to the Sharding pattern.

In order to observe more dynamics of combined patterns, we have chosen a heavy-weighted scenario of the application in which a VIP customer pushes an order to the system using a combination of Sharding pattern (to guarantee the locality of the data and speed of retrieval) and Pipes and Filters pattern (to ensure high efficiency by enabling multitasking and increasing the computing power). Our results show that the combination of Sharding and Pipes and Filter patterns can significantly improve the performance of an application (see Table 5.3), however, this will be detrimental of the energy efficiency, as shown on Figure 5.13.

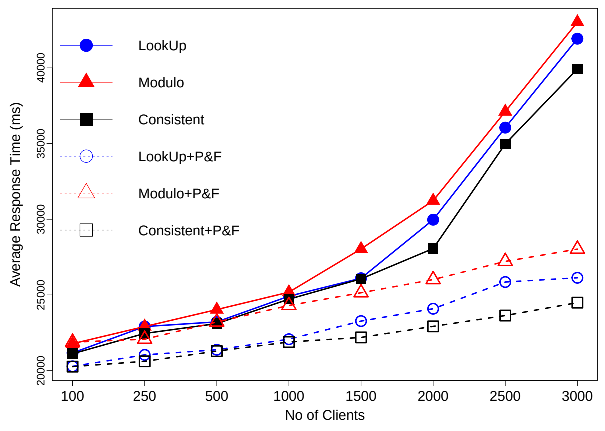


a) *Average Response Time*

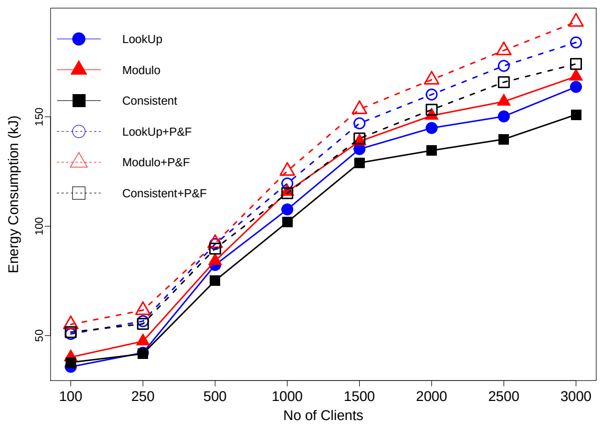


b) *Energy Consumption*

Figure 5.12 The Impact of Sharding Pattern Combined with Message Queue Pattern



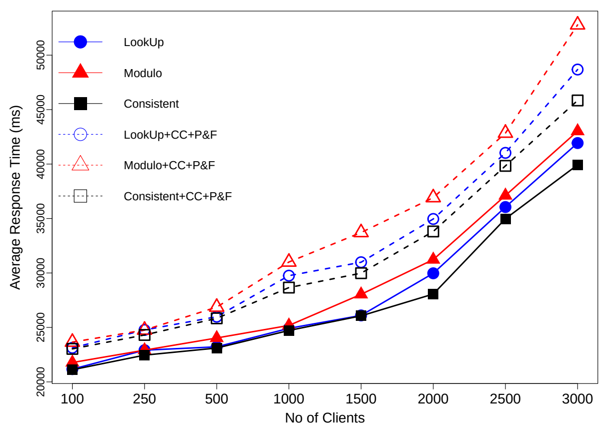
a) *Average Response Time*



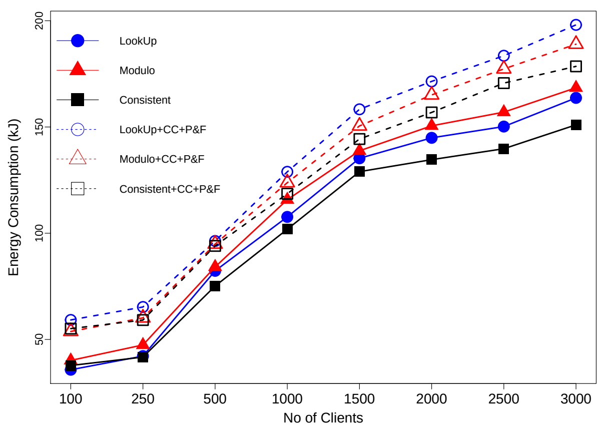
b) *Energy Consumption*

Figure 5.13 The Impact of Sharding Pattern Combined with Pipes & Filters Pattern

One advantage of the Competing Consumers pattern is the fact that it enables scaling up the application, adding more resources to the computation and data handling services, as needed. In order to test the efficiency of this pattern, we designed a mixed combination of the Sharding, Pipes and Filters, and Competing Consumers patterns. A request from a client would be sharded accordingly, processed via the Pipes and Filters and delivered through the Competing Consumers message queue channel.



a) *Average Response Time*

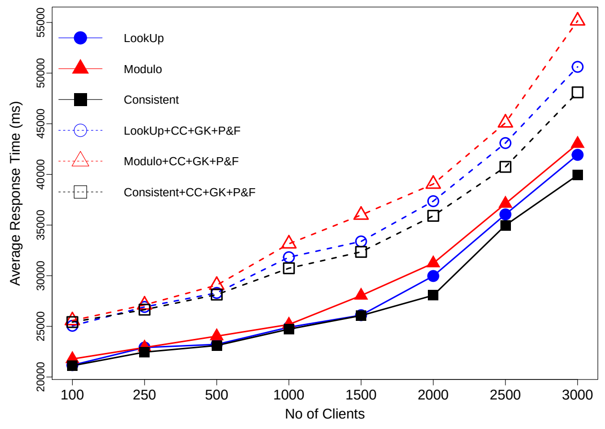


b) *Energy Consumption*

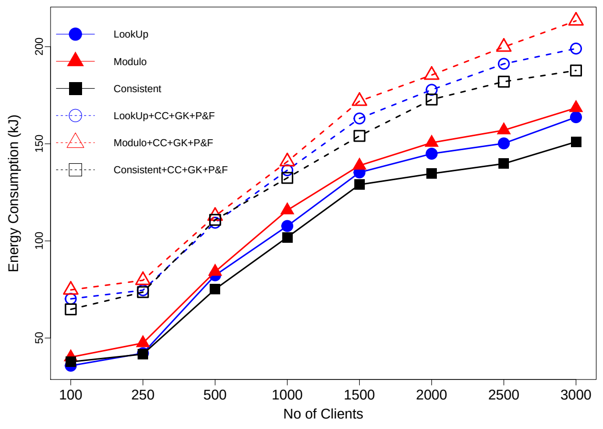
Figure 5.14 The Impact of Sharding Pattern Combined with Competing Consumers and Pipes & Filters Patterns

We observed that the combination of these three patterns did not significantly increase the amount of energy consumed by the application (see Figure 5.14). However, the response time was impacted (we obtained p-values less than 0.05 in Table 5.3, with effect size of around 0.3). Hence, developers should be aware that, although a combination of these three patterns can improve the scalability of their applications with little impact on energy efficiency, the performance can be penalized.

Since in a typical cloud-based application, developers use a Gatekeeper to ensure that only authorized requests are processed, we experimented with a combination of Gatekeeper, Sharding, Pipes and Filters, and Competing Consumers patterns. Results presented in Figure 5.15 shows that the resulting application is slower and consumes more energy. The impact of a combination of these four patterns, on both response time and energy consumption is statistically significant (see Table 5.3).



a) *Average Response Time*



b) *Energy Consumption*

Figure 5.15 The Impact of Sharding Pattern Combined with Competing Consumers, Gatekeeper and Pipes & Filters Patterns

The Gatekeeper seems to increase the security at the expense of both response time and energy efficiency. This outcome was expected because of the additional processing occurring at the level of the Gatekeeper, to filter out malicious queries.

Summary : a combination of Proxy and Message Queue patterns can have a negative impact on the performance of an application, but not on energy efficiency. However, a combination of Message Queue and Sharding patterns does not have a negative impact neither on response time or energy efficiency. When the Sharding pattern is combined with Pipes and Filters, the performance of the application can improve significantly. However, this improvement can be at the expense of energy efficiency. A combination of Sharding, Pipes and Filters, and Competing Consumers patterns can improve the scalability of an application, with little impact on energy efficiency. However, the application may experience a performance degradation. When the Gatekeeper is added to an application, both response time and energy efficiency can be affected.

5.3 Threats to Validity

Any empirical study is subject to threats to validity. This section discusses threats to the validity of our work following the guidelines provided by Wohlin *et al.* (Wohlin et al., 2012):

Construct validity threats concern the relation between theory and observations. In this study, they are measurement errors. We repeated each experimentation five times and computed average values, in order to mitigate the potential biases by the highest and lowest that could be induced by perturbations on the network or the hardware, and our tracing. the Power-API tool was carefully compiled and calibrated before each run of the application. The first and last values were eliminated to obtain lower variance between the average and maximum. We believe that these operations increased the quality of our measurements.

Internal validity threats concern our selection of subject systems and analysis methods. Although we have used a well-known benchmark, and well-known patterns and algorithms, some of our findings may still be specific to our studied applications which were designed specifically for the experiments. Applications from two different languages (Java and Python) have been used, and we obtained consistent results for energy consumption and minor variance in response time. However, more studies should be performed using different cloud-based applications and different energy measurement tools, to make our findings more generic.

External validity threats concern the possibility to generalize our findings. Further validation with different cloud patterns can extend our understanding of the impact of cloud patterns on the energy consumption of applications, providing developers with guideline about the usage of cloud patterns when developing cloud-based applications.

Reliability validity threats concern the possibility of replicating this study. We attempt to provide all the necessary details to replicate our study. All the data used in this thesis are available online: `https://goo.gl/Cczot4` and `https://goo.gl/MB1Yvk`.

Finally, *conclusion validity* threats refer to the relation between the treatment and the outcome. We paid attention not to violate the assumptions of the performed statistical tests. We mainly used non-parametric tests that do not require making assumptions about the distribution of the metrics.

CHAPTER 6 CONCLUSION

6.1 Summary of the Results

In this study, we examine the impact on energy consumption of six cloud patterns (*i.e.,* Local Database Proxy, the Local Sharding-Based Router, the Priority Queue, Competing Consumers, The Gatekeeper and Pipes and Filters patterns), with the aim to provide some guidance to developers about the usage of cloud patterns. More specifically, we conducted two experiments with different versions of two cloud-based applications, implementing the patterns. Our results show that any implementation of the Proxy pattern can significantly improve the energy efficiency of a cloud-based application, while the Local Sharding-Based Router pattern only has a small effect on energy consumption. In fact, only the consistent hashing algorithm seems to have a positive effect on the energy efficiency of applications using the Local Sharding-Based Router pattern. Overall, the Local Database proxy appears to be more adapted for applications experiencing heavy loads of *read requests*, while the Local Sharding-Based Router is not suitable for such applications, but seems more appropriate for applications handling huge *write requests* loads.

In addition, our results show that combining the Priority Message Queue pattern with the Local Database Proxy pattern has no significant impact neither on the application's response time, nor on the average amount of energy consumed by the application. Local Sharding Based Router when combined with Local Database Proxy improves response time. Interestingly, the implementation of the custom proxy algorithm in a Local Database Proxy pattern combined with the modulo algorithm in a Local Sharding-Based Router pattern can improve the response time of an application, without penalizing its energy efficiency.

We also observed that migrating an application to a microservices architecture can improve the performance of the application, while significantly reducing its energy consumption. A combination of Sharding, Pipes and Filters, and Competing Consumers patterns can improve the scalability of an application, with little impact on energy efficiency. However, the application may experience a performance degradation. Although the Gatekeeper can improve the security of a cloud-based application, this is done at the expense of both response time and energy efficiency.

In general, there appears to be a trade-off between the response time and the energy consumption. Developers should be careful when selecting a cloud pattern for their application. Table 6.1 summarizes the impact on response time and energy efficiency of the six studied patterns.

Table 6.1 Impact of Patterns on Energy Efficiency and Response Time

Context Problem	**Pattern**	**Algorithm**	**Energy**	**Response Time**
Performance & Scalability	Proxy	PRX-RND	Decreasing ↓	Decreasing ↓
Performance & Scalability	Proxy	PRX-RR	Decreasing ↓	Decreasing ↓
Performance & Scalability	Proxy	PRX-CU	Decreasing ↓	Decreasing ↓
Data Management	Sharding	SHRD-LU	Decreasing ↓	Decreasing ↓
Data Management	Sharding	SHRD-MD	Increasing ↑	Increasing ↑
Data Management	Sharding	SHRD-CU	Decreasing ↓	Decreasing ↓
Performance & Scalability	Competing Consumers	CCP-Mono	Increasing ↑	Increasing ↑
Performance & Scalability	Competing Consumers	CCP-Micro	Decreasing ↓	Decreasing ↓
Security	The Gatekeeper	GK-Mono	Increasing ↑	Increasing ↑
Security	The Gatekeeper	GK-Micro	Decreasing ↓	Decreasing ↓
Messaging & Performance	Pipes & Filters	P&F	Decreasing ↓	Decreasing ↓

Performance & Scalability: Most of the cloud-based applications require high performance and scalability. Cloud computing, in fact, tries to provide the infrastructure with which an application can scale to thousand, even millions of concurrent users. Proxy and Competing Consumers patterns aim to improve the performance and the scalability of applications. The results of this study suggests that all the three algorithms used to implement the Proxy pattern and the Competing Consumers pattern can improve response time and energy consumption (see Table 6.1).

Data Management: Cloud infrastructures offer the possibility to store and access large amounts of data quickly. Sharding algorithms can have a positive impact on increasing productivity, specially in No-SQL databases which are designed to bring scalable data storage to cloud. We observed that the two algorithms LookUp and Consistent Hashing are able to lower energy consumption and increase the performance of the applications. On the contrary, the Modulo algorithm did not improve response time or energy efficiency.

Security: Applications, deployed in the cloud, face many of the same threats as traditional corporate networks. Because of the large amount of data stored on cloud servers, most cloud providers are prime targets for attackers. The magnitude and sensitivity of the data stored in cloud servers make security breaches more severe. The Gatekeeper pattern aims to secure the access to resources hosted in the cloud. However, when used in a monolithic application, this pattern can increase the response time and the energy consumption of the application. We recommend that developers make use of this pattern primarily in micro-services applications, since it can significantly improve the performance and the energy efficiency of micro-services applications.

Messaging: The Pipes and Filters pattern uses message queues to provide an asynchronous communications protocol. By doing so, the sender and receiver of the message (application and service pool instances) do not need to interact with the message queue at the same time. This results in scalable and highly performant architectures, which significantly increases the reliability and availability of the cloud-based applications. Our study have shown that Pipes and Filters patterns also increase the response time and the energy efficiency of applications.

6.2 Future Work

The study presented in this thesis can be extended to different relational databases and NoSQL databases, where multiple fine grained optimizations are performed to improve service availability. NoSQL databases are not using the relational model, and they are increasingly used in big data applications, which are consuming more and more energy these days, and hence would significantly benefit from energy-aware designs. Patterns can be also applied to scientific computing and High Performance Computing (HPC) applications, in order to achieve performance and energy optimizations.

REFERENCES

A. Ampatzoglou, G. Frantzeskou, and I. Stamelos, "A methodology to assess the impact of design patterns on software quality," *Information and Software Technology*, vol. 54, no. 4, pp. 331–346, 2012.

K. Aras, T. Cickovski, and J. A. Izaguirre, "Empirical evaluation of design patterns in scientific application," 2005.

C. A. Ardagna, E. Damiani, F. Frati, D. Rebeccani, and M. Ughetti, "Scalability patterns for platform-as-a-service," in *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on.* IEEE, 2012, pp. 718–725.

A. Author, *Information technology - Cloud computing, Overview and vocabulary*, ISO ISO/IEC 17 788:2014, 2014.

A. Beloglazov, "Energy-efficient management of virtual machines in data centers for cloud computing," 2013.

R. Burtica, E. M. Mocanu, M. I. Andreica, and N. Ţăpuş, "Practical application and evaluation of no-sql databases in cloud computing," in *Systems Conference (SysCon), 2012 IEEE International.* IEEE, 2012, pp. 1–6.

C. Calero and M. Piattini, *Green in Software Engineering.* Springer, 2015.

R. Cattell, "Scalable sql and nosql data stores," *ACM SIGMOD Record*, vol. 39, no. 4, pp. 12–27, 2011.

N. Cliff, "Dominance statistics: Ordinal analyses to answer ordinal questions." *Psychological Bulletin*, vol. 114, no. 3, p. 494, 1993.

J. Cohen, *Statistical power analysis for the behavioral sciences (rev.* Lawrence Erlbaum Associates, Inc, 1977.

I. El Korbi and S. Zeadally, "Energy-aware sensor node relocation in mobile sensor networks," *Ad Hoc Networks*, vol. 16, pp. 247–265, 2014.

B. Furht and A. Escalante, *Handbook of cloud computing.* Springer, 2010, vol. 3.

M. Goraczko, A. Kansal, J. Liu, and F. Zhao, "Joulemeter: Computational energy measurement and optimization," 2011.

G. Hecht, B. Jose-Scheidt, C. De Figueiredo, N. Moha, and F. Khomh, "An empirical study of the impact of cloud patterns on quality of service (qos)," in *6th International Conference on Cloud Computing Technology and Science.* IEEE, 2014, pp. 278–283.

A. Homer, J. Sharp, L. Brader, M. Narumoto, and T. Swanson, *Cloud Design Patterns: Prescriptive Architecture Guidance for Cloud Applications.* Microsoft patterns & practices, 2014.

F. Khomh and Y.-G. Guéhéneuc, "Do design patterns impact software quality positively?" in *Software Maintenance and Reengineering, 2008. CSMR 2008. 12th European Conference on.* IEEE, 2008, pp. 274–278.

K. Liu, G. Pinto, and Y. D. Liu, "Data-oriented characterization of application-level energy optimization," in *International Conference on Fundamental Approaches to Software Engineering.* Springer, 2015, pp. 316–331.

S. S. Mahmoud and I. Ahmad, "A green model for sustainable software engineering," *International Journal of Software Engineering and Its Applications*, vol. 7, no. 4, pp. 55–74, 2013.

R. Nambiar, M. Poess, A. Dey, P. Cao, T. Magdon-Ismail, A. Bond *et al.*, "Introducing tpcx-hs: the first industry standard for benchmarking big data systems," in *Technology Conference on Performance Evaluation and Benchmarking.* Springer, 2014, pp. 1–12.

A. Noureddine, A. Bourdon, R. Rouvoy, and L. Seinturier, "A preliminary study of the impact of software engineering on greenit," in *Green and Sustainable Software (GREENS), 2012 First International Workshop on.* IEEE, 2012, pp. 21–27.

C. Pang, A. Hindle, B. Adams, and A. E. Hassan, "What do programmers know about software energy consumption?" *IEEE Software*, vol. 33, no. 3, pp. 83–89, 2016.

G. Pinto, F. Castor, and Y. D. Liu, "Mining questions about software energy consumption," in *MSR*, 2014, pp. 22–31.

L. Prechelt, B. Unger, W. F. Tichy, P. Brossler, and L. G. Votta, "A controlled experiment in maintenance: comparing design patterns to simpler solutions," *Software Engineering, IEEE Transactions on*, vol. 27, no. 12, pp. 1134–1144, 2001.

F. Prosperi, M. Bambagini, G. Buttazzo, M. Marinoni, and G. Franchino, "Energy-aware algorithms for tasks and bandwidth co-allocation under real-time and redundancy constraints," in *Emerging Technologies & Factory Automation (ETFA), 2012 IEEE 17th Conference on.* IEEE, 2012, pp. 1–8.

J. Romano, J. D. Kromrey, J. Coraggio, and J. Skowronek, "Appropriate statistics for ordinal level data: Should we really be using t-test and cohen'sd for evaluating group differences on the nsse and other surveys," in *annual meeting of the Florida Association of Institutional Research*, 2006, pp. 1–33.

K. Sachs, S. Kounev, J. Bacon, and A. Buchmann, "Performance evaluation of message-oriented middleware using the specjms2007 benchmark," *Performance Evaluation*, vol. 66, no. 8, pp. 410–434, 2009.

D. J. Sheskin, *Handbook of parametric and nonparametric statistical procedures.* crc Press, 2003.

S. Strauch, V. Andrikopoulos, U. Breitenbuecher, O. Kopp, and F. Leyrnann, "Non-functional data layer patterns for cloud applications," in *Cloud Computing Technology and Science (CloudCom), 2012 IEEE 4th International Conference on.* IEEE, 2012, pp. 601–605.

B. G. Tudorica and C. Bucur, "A comparison between several nosql databases with comments and notes," in *Roedunet International Conference (RoEduNet), 2011 10th.* IEEE, 2011, pp. 1–5.

W. Vereecken, W. Van Heddeghem, D. Colle, M. Pickavet, and P. Demeester, "Overall ict footprint and green communication technologies," in *4th International Symposium on Communications, Control and Signal Processing (ISCCSP 2010).* IEEE, 2010.

M. Vokáč, W. Tichy, D. I. Sjøberg, E. Arisholm, and M. Aldrin, "A controlled experiment comparing the maintainability of programs designed with and without design patterns—a replication in a real programming environment," *Empirical Software Engineering*, vol. 9, no. 3, pp. 149–195, 2004.

C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in software engineering.* Springer Science & Business Media, 2012.

APPENDIX A CO-AUTHORSHIP

The results of this study have been published as follows:

- S. A. Abtahizadeh, F. Khomh, YG. Guéhéneuc, “How green are cloud patterns?”, *Proceedings of the 2015 IEEE 34th International Performance Computing and Communications Conference (IPCCC)*, December 14 - 16, 2015, Nanjing, China.

  My contribution: Methodology, analysis, and paper writing.