Understanding the Impact of Cloud Patterns on Performance and Energy Consumption

Foutse Khomh[a], S. Amirhossein Abtahizadeh[a]

[a]*SWAT Lab., Polytechnique Montréal, Canada*

Abstract

Cloud Patterns are abstract solutions to recurrent design problems in the cloud. Previous work has shown that these patterns can improve the Quality of Service (QoS) of cloud applications but their impact on energy consumption is still unknown. In this work, we conduct an empirical study on two multi-processing and multi-threaded 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-based 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 implementation of a cloud-based application.

Keywords: Cloud Patterns, Energy Consumption, Performance Optimization, Energy Efficiency

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 both the hardware platform and the multiple software layers. Recently, researchers have started to investigate the role of software components and coding practices [1, 2] on the energy efficiency of software systems.

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 [3] has shown that these cloud patterns can improve the QoS of cloud based applications, their impact on energy consumption is still unknown. This paper investigates the impact on energy consumption of six cloud patterns: Local Database Proxy, Local Sharding-Based Router, Priority Queue, Competing Consumers, Gatekeeper, and Pipes and Filters. We aim to understand the trade-offs that may exist between energy consumption and performance when implementing cloud patterns in applications. The study is conducted 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 [4].

This paper extends our previous work [5] in three ways. First, we added a new cloud-based application to our study and deployed it in a commercial cloud environment (*i.e.,* Amazon EC2). Second, we have added three more cloud patterns to our study (*i.e.,* Competing Consumers, Gatekeeper, and Pipes and Filters). Third, we summarize our findings into architectural design guidelines that developers and software architects can follow to improve the performance and energy efficiency of cloud-based applications.

The remainder of this paper is organized as follows. Section 2 discusses the related literature on cloud patterns and green software engineering. Section 3 introduces the patterns that are investigated in this paper and Section 4 describes the design of our study. Section 5 discusses

Email addresses: foutse.khomh@polymtl.ca (Foutse Khomh), a.abtahizadeh@polymtl.ca (S. Amirhossein Abtahizadeh)

our results and Section 6 summarizes these results into guidelines for developers of cloud-based applications. Section 7 discusses threats to the validity of our study, while Section 8 concludes the paper and outlines future work.

2. Related Work

Energy consumption is the biggest challenge that cloud computing systems face today. Pinto *et al.* [6] who analyzed more than 300 questions and 550 answers on the Q&A web site Stack Overflow reported that the number of questions on energy consumption increased by 183% from 2012 to 2013. The majority of these questions were related to software design, showing that developers need guidance for designing green software systems. Similarly, Pang *et al.* [7] found that developers lack knowledge on how to develop energy-efficient software. Nowadays, cloud-based applications are becoming mainstream, thanks to their high availability and scalability. Many efforts have been focused on modeling and improving the energy efficiency of cloud infrastructures from the hardware point of view, 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 [3] and [8] have shown that patterns can improve the QoS of cloud based applications, their impact on energy consumption is still unknown.

2.1. Object-Oriented Design Patterns and Software Quality

Several works in the literature have assessed the impact of design patterns on software quality [9], software maintenance [10] and code complexity [11]. Overall, these studies have found that design patterns do not always improve the quality of applications. Khomh and Guéhéneuc [9] 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.* [12] 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 disadvantages 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.* [13] 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.* [14] and Burtica *et al.* [15] 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 [16] examined NoSQL and SQL data stores designed to scale by using replication and sharding. Similarly, they also did not perform energy consumption evaluations. Message oriented middlewares have been benchmarked by Sachs *et al.* [17], however, the energy consumption aspect was also ignored.

Regarding the relationship between cloud patterns and energy consumption, Beloglazov [18] 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 proposed 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 of OpenStack that implements the dynamic consolidation of Virtual Machines (VMs), using live migration.

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. Although developers can take advantage of software tools that monitor resources in order to observe the energy consumption of their applications, there is a lack of software development guidelines that developers can follow to minimize the energy consumed by their application, while preserving the QoS. Mahmoud *et al.* [20] 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 [21]. This book discusses the impact of software on environment

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

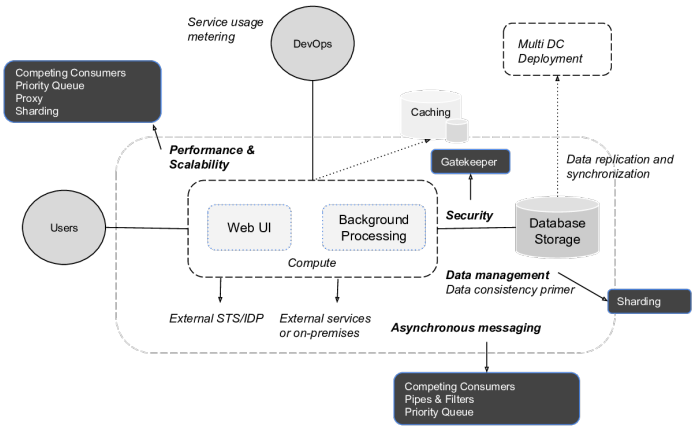


Figure 1: Illustration of the architecture of cloud applications (with pattern suggestions) [19]

and provides important guidelines for making software engineering "green", *i.e.*, reducing their environmental and energy footprints.

TPC-benchmark [22] enables benchmarking databases performance by simulating a series of users queries against the database. The TPC-Energy (which was introduced recently) allows examining the energy consumption of servers, disks, and other items that consume power. 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.

3. Designing Cloud Applications with Patterns

Designing applications for the cloud has some challenges. In the cloud, there is 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. Typically, cloud-based applications depend on third-party software on which we have no control. Therefore, it is important to implement fault-tolerance mechanisms to handle the potential failures of these third-party software. 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. In the following, we discuss several possible problems that could be tackled with cloud patterns.

Figure 1 illustrates different problems in cloud applications that can be solved using cloud patterns. As one can see in Figure 1, developers face many problems during the development of cloud based applications. In this paper, 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 1 maps four common problems faced by developers when designing and implementing cloud-based applications, to the appropriate cloud patterns. For example, the Gatekeeper pattern can be used whenever security is critical, and the Sharding-Based Router can be used to solve data management issues when the data is partitionable.

Table 1: Four common problems in cloud area and suggested patterns

Problem in cloud area	**Suggested pattern**
Messaging	Competing Consumers, Pipes & Filters, Priority Message Queue
Performance & Scalability	Competing Consumers, Priority Message Queue, Local Database Proxy, Local Sharding-Based Router
Security	Gatekeeper
Data Management	Local Sharding-Based Router

3.1. Cloud Patterns

We now describe in details the six cloud patterns studied in this paper.

3.1.1. Local Database Proxy

The Local Database Proxy pattern (also known as Proxy pattern) uses data replication between master/slave databases and a proxy to route requests [23]. 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.* [3]; 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

The Local Sharding-Based Router pattern (also known as Sharding pattern) is useful when an application needs scalability both for read and write operations [23]. 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) [3]. However, the impact of Sharding on energy consumption is still unclear.

3.1.3. Priority Message Queue

The Priority Message Queue pattern (also known as Priority Queue pattern) implements a First In First Out (FIFO) queue. It 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. Low priority messages are pushed back to the end of the queue. Message Queues enable designing loosely coupled components and improve the scalability of applications [3].

3.1.4. Competing Consumers

Cloud-based application typically face heavy request loads from users. A common technique to ensure responses is to pass the requests 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 and coordinating multiple instances of the consumer service 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.

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.

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. The goal of this pattern is to decouple application instances from storage, ensuring that *trusted hosts* connect only to the gatekeeper(s) and not directly to clients. A secure communication channel (HTTPS, SSL, or TLS) must be employed between trusted hosts and the gatekeeper(s). 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.

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 the 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. 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.

3.2. 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 the tool (*i.e.,* Power-API) used in this paper to estimate the power consumption of applications.

Power-API is a system-level library that provides power information per PID for each system component (*e.g.,* CPU, network card, etc.) [4]. The energy estimation is performed using analytical models that characterize the consumption of the components (*i.e.*, CPU, memory, and disk). The accuracy of Power-API was evaluated using the bluetooth PowerMeter (PowerSpy) [24], and results revealed only minor variations between the energy estimations of Power-API, and the energy consumption measured by PowerSpy [4].

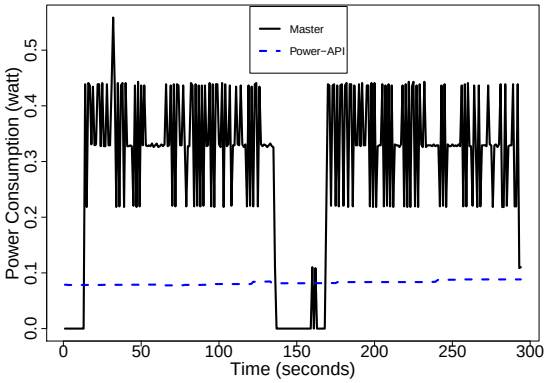


Figure 2: Power-API distortion test

Power-API was selected for this work because of its reported high accuracy [4], 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). Prior to this study, we performed a pilot procedure to examine the risk that Power-API introduces 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.

The result illustrated in Figure 2 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 the 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 were embedded in a bash script file in order to measure the energy consumption of the application during each execution of the application.

Other software-defined power meters available in the literature include Joulemeter [25], jRAPL [26], and Power Gadget[2].

4. Study Definition and Design

Previous work has shown that cloud patterns can improve the Quality of Service (QoS) of cloud applications [3], 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

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

trade-offs between performance and energy optimization, during the development of cloud-based applications.

The *objects* of this study are six cloud based patterns (*i.e.,* Local Database Proxy, Local Sharding-Based Router, Priority Message Queue, Competing Consumers, Gatekeeper and Pipes and Filters) implemented in two cloud-based applications. We select these cloud patterns because they concern critical aspects of the design of cloud-based applications (*i.e.,* messaging, data management, security, performance, and scalability) and they are recommended to developers as good design practices by [19, 23].

4.1. 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. Experimental Setup

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

4.2.1. Experiment 1

A multi-threaded distributed application, which communicates through REST calls was implemented by two master's students from Polytech Montpellier (with two years of programming experience) and one Master's student from Polytechnique Montreal (with more than five years of programming experience). The application was deployed on a GlassFish 4 application server. We chose MySQL as the database management system because at the time of our experimentations it was one of the most popular databases for cloud-based applications. Sakila database[3] provided by MySQL was selected because it contains a large number of records, making it interesting for experimentations. Sakila is consistent with existing databases. The test application was fully developed with the Java Development Kit 1.7 (by the same three master's students) 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 hardwares are 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 3.

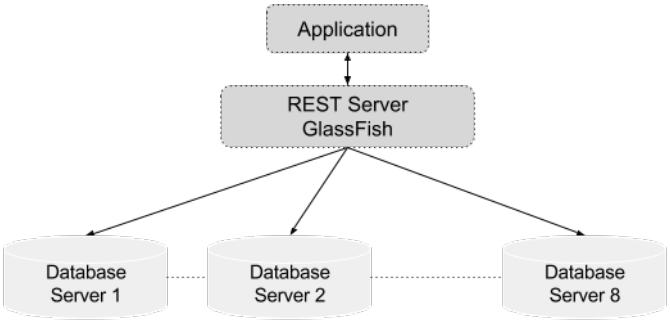


Figure 3: Architectural Design of the First Experiment

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 then performs a series of actions in a certain amount of time (see Figure 4). 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.

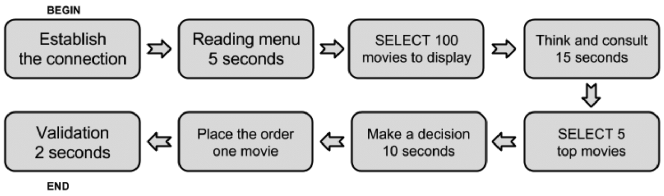


Figure 4: First Experiment Test Scenario

Table 2: List of microservices and their tasks

ID	Name	Task
1	Microservice 1	Login service
2	Microservice 2	Search a customer and display information
3	Microservice 3	List products
4	Microservice 4	View customer shipping information
5	Microservice 5	Add/Edit shipping information
6	Microservice 6	Preview order and calculate subtotal
7	Microservice 7	Submit an order

We repeated this scenario using different number of clients. The requests performed by the clients are: *select*

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

or *write* to *display* or *place the order* respectively. The application generated 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 was located on the database node and measured 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.

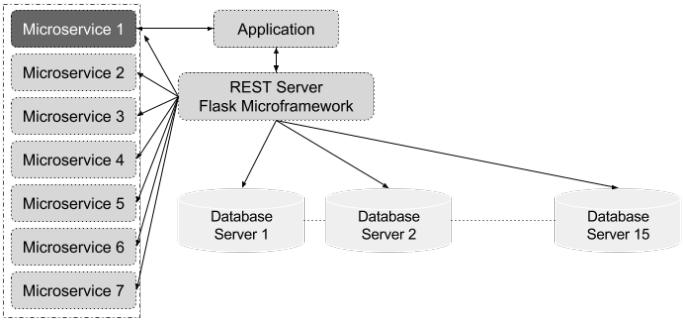


Figure 5: Architectural Design of the Second Experiment

4.2.2. 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 real world and, 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 2) 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 microservices 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 monolithic application to be rebuilt and deployed. In this work, we have implemented both a monolithic and a microservices versions for our application. The refactoring of the application into a microservices architecture was performed by the second author.

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 microservices 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 5.

Northwind sample database[4] was chosen for Experiment 2. 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 occurred 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[5] 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.

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 6. 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.

The number of clients used in our experiments are: 100, 250, 500, 1000, 1500, 2000, 2500 and 3000. Power-API is located on the database node and REST server nodes, and measures the amount of energy consumed by MySQL

[4]https://northwinddatabase.codeplex.com/
[5]http://github.com/dalers/mywind

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.

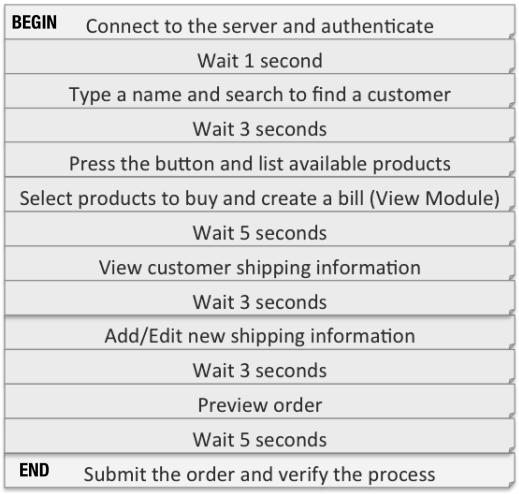


Figure 6: Second Experiment Test Scenario

To better understand the impact of combinations of cloud patterns, we designed two approaches (lightweight and heavyweight approaches) mimicking the behavior of ordinary customers and more demanding customers. These two types of customers represent companies that are typical buyers. They are characterized by the complexity of their queries.

The lightweight approach simulates ordinary customers (for example, customers who do not require a specific security protection). When a sales representative faces this type of customer, the application automatically selects the lightweight approach to provide services. In the lightweight approach, the application formulates simple queries (for example, the number of security verifications can be reduced), which improves the response time of the application. A set of routine services will use the combination of Priority Message Queue, Proxy and Sharding patterns in order to ensure scalability and data management. Since the queries are not too complex (there is no join between tables), these combinations are enough to keep the application running with a high QoS.

The heavyweight approach simulates customers that require some extra care. In this configuration, the amount of the queries that are sent to databases exceeds the typical load for an ordinary customer and there are several joins between the tables in order to collect and represent data. Therefore, this second approach utilizes a combination of Sharding, Competing Consumers and Pipes and Filters patterns, to ensure data management, and the highest possible scalability. Developers can also add the Gatekeeper pattern to increase the security of the application. In summary, we experiment with the following two configurations:

- Lightweight approach (Simple queries):

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

- Heavyweight approach (Complex queries):

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

4.3. 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 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 are examined: version E7 which has a microservices architecture and E8 which is monolith and performs all the tasks in a series of orders. We also examine two implementations of the Competing Consumers pattern: versions E9 and E10 which correspond respectively to a microservices and a monolithic architectural style. We implemented the Pipes and Filters pattern following the microservices architectural style and refer to it as E11. 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 3 summarizes the different versions of the applications that were produced.

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. Outlier measurement values were recorded during these two 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 application. In the following, we explain in details the specific algorithms that were implemented in each pattern.

Table 3: 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

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, *No-Proxy* 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. The query is executed consequently whenever the result is null on the master node in order to make sure that the replication did not fail. 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[6]. 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 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 it 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 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 explain-

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

Table 4: 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

ing 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 summarizes the SQL injection rules applied to this pattern in our study.

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 using 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 lightweight 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 as summarized in Table 3. 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.

4.4. Hypotheses

To answer our research questions, we formulate the following null hypotheses, where E0, Ex ($x \in \{1 \dots 11\}$), and E12 are the different versions of the applications, as described in Table 3.

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

For each implementation of the patterns Local Database proxy, Local Sharding Based Router, Competing Consumers, Gatekeeper and Pipes and Filters, we formulate the following null hypothesis to compare the energy consumed by the version of the application implementing the pattern (i.e., Ex ($x \in \{1 \dots 11\}$)), and the version of the application in which no pattern is implemented, i.e., E0.

H^1_x *: There is no difference between the average amount of energy consumed by design Ex and design E0.*

We do not have a null hypothesis for the pattern Priority Message Queue because in practice this pattern is used in combination with other patterns such as the Competing Consumers pattern, and not in isolation.

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

To answer this research question we examine common combined implementations of our six studied patterns as described in [19]. Specifically, we examine combinations of cloud patterns that are recommended to address the challenges of our lightweight and heavyweight approaches described in Section 4.2.2.

- Lightweight approach (Simple queries)

- MQ + Proxy and MQ + Sharding. We formulate the following null hypothesis; H^1_{x12} *: The average amount of energy consumed by the combination of designs Ex ($x \in \{1 \dots 6\}$) and E12 is not different from the average amount of energy consumed by each design taken separately.*

- Sharding + Pipes & Filters. We formulate the following null hypothesis; H^1_{x11} *: The average amount of energy consumed by the combination of designs Ex ($x \in \{4 \dots 6\}$) and E11 is not different from the average amount of energy consumed by each design taken separately.*

- Heavyweight approach (Complex queries):

- Sharding + Competing Consumers + Pipes & Filters. We formulate the following null hypothesis; H^1_{x9} : *The average amount of energy consumed by the combination of designs Ex ($x \in \{4 \dots 6\}$) and E11 and E9 is not different from the average amount of energy consumed by each design taken separately.*

- Sharding + Competing Consumers + Gatekeeper + Pipes & Filters. We formulate the following null hypothesis; H^1_{x7} : *The average amount of energy consumed by the combination of designs Ex ($x \in \{4 \dots 6\}$) 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 related to the performance of the different implementations of the applications (measured in terms of response time):

- 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 ($x \in \{1 \dots 6\}$) 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.*

4.5. Analysis Method

Since the observations of each scenario were independent of those of the other scenarios, we perform the Mann-Whitney U test [27] to test H^1_x, H^2_x, H^1_{x7}, H^2_{x7}. Moreover, we computed the Cliff's δ effect size [28] 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 [29]. We perform all our tests using a 95% confidence level (*i.e.*, p-value < 0.05). Since we conduct multiple null hypothesis tests, to counteract the problem of multiple comparisons, we apply the Bonferroni correction [30] that consists in dividing the threshold p-value by the number of tests. 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 assumption 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 [28]. Cliff's δ effect size values range 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 [31]. 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 $>= 0.474$.

4.6. Independent Variables

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 3 are the independent variables of our study.

4.7. 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: $E_{kJ} = (P_{watt} \text{ x } t_s)/\ 1000$.

5. The Impact of Cloud Patterns on Performance and Energy Consumption

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

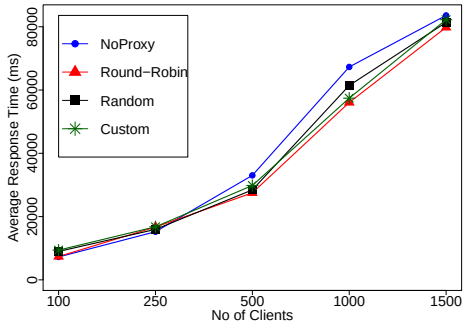
5.1. Results of Experiment 1

Table 5 summarizes the results of Mann–Whitney U tests and Cliff's δ effect sizes for the energy consumption and the response time. We marked significant results in bold.

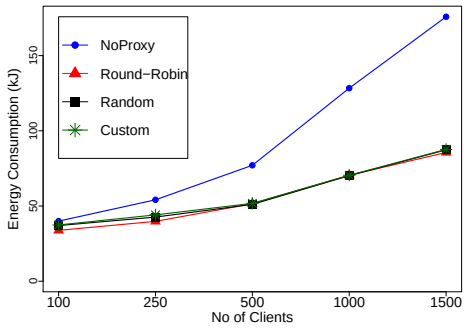
Average amount of consumed energy: results presented in Table 5 show that, there is a statistically significant difference between the average amount of energy consumed by an application implementing the Local Database Proxy pattern and an application not implementing this pattern. The effect size is large in almost all cases. Therefore, we reject H^1_x for all Ex ($x \in \{1 \dots 3\}$). Regarding the Local Sharding-Based Router pattern, except for the

Table 5: p-value of Mann–Whitney U test and Cliff's δ effect size for the first application

Version	Avg. Response Time		Avg. 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, E2	0.59	0.12	0.71	0.08
E1, E3	0.59	-0.12	0.48	-0.15
E1, E4	**<0.05**	-0.76	**<0.05**	-0.52
E1, E5	**<0.05**	-0.6	**<0.05**	-0.44
E1, E6	**<0.05**	-0.6	0.87	0.04
E2, E3	0.46	-0.16	0.56	-0.12
E2, E4	**<0.05**	-0.76	**<0.05**	-0.52
E2, E5	**<0.05**	-0.6	**<0.05**	-0.44
E2, E6	**<0.05**	-0.6	0.59	-0.12
E3, E4	**<0.05**	-0.76	**<0.05**	-0.51
E3, E5	**<0.05**	-0.6	0.08	-0.36
E3, E6	**<0.05**	-0.6	0.87	0.04
E4, E5	0.18	0.28	0.36	0.2
E4, E6	0.09	0.36	**<0.05**	0.52
E5, E6	0.59	0.12	**<0.05**	0.46
E1, E1 + E7	0.38	0.19	0.53	-0.13
E2, E2 + E7	0.43	0.17	0.20	-0.28
E3, E3 + E7	0.51	0.14	0.53	-0.13
E4, E4 + E7	**<0.05**	0.68	0.2	-0.28
E5, E5 + E7	**<0.05**	0.52	0.2	-0.28
E6, E6 + E7	**<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



(a) Average Response Time



(b) Energy Consumption

Figure 7: The Impact of the Local Database Proxy Pattern

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 implement 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 low. Therefore, we cannot reject H_x^1 for all Ex ($x \in \{4 \dots 6\}$).

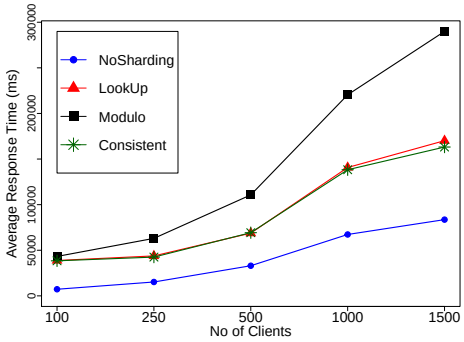
Our results show that any implementation of the Local Database Proxy pattern can significantly improve the energy efficiency of an application, while the Local Sharding-Based Router pattern has little effect on energy consumption. Figure 7 and Figure 8 summarize the results obtained for all the implementations of the two patterns.

Average response time: results from Table 5 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 ($x \in \{4 \dots 6\}$). In fact, as shown on Figure 8, all the implementations 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.

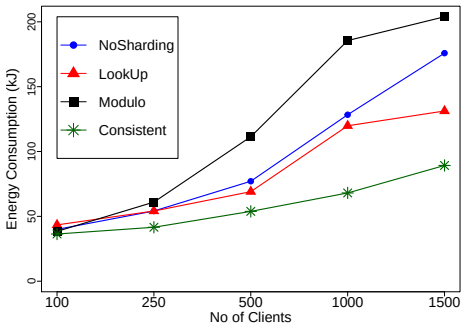
Regarding the Local Database Proxy pattern, there is no statistically significant difference between the response time of applications using any version of the pattern and applications not using the pattern. However, Figure 7, as well as effect size values show that Local Database Proxy pattern has a (small) positive impact on the response time of the applications. Yet, we cannot reject H_x^2 for ($x \in \{1 \dots 3\}$).

Combination of Patterns: Table 5 and Figure 9 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_{x7}^2 for all Ex ($x \in \{1 \dots 3\}$).

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 Local Sharding-Based Router pattern and application imple-

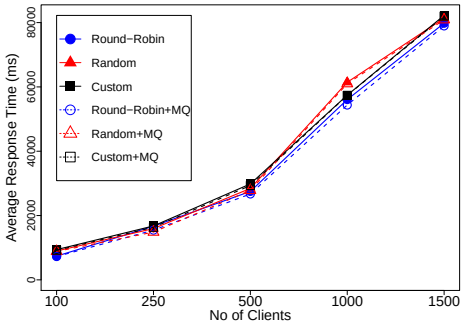


(a) Average Response Time

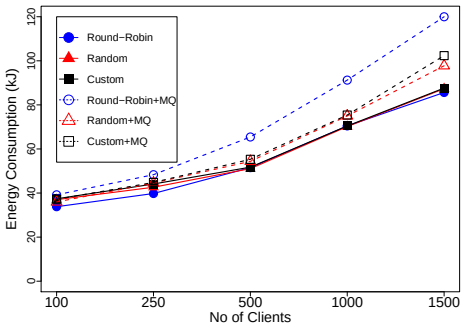


(b) Energy Consumption

Figure 8: The Impact of the Local Sharding-Based Router pattern



(a) Average Response Time



(b) Energy Consumption

Figure 9: The Impact of Local Database Proxy combined with Priority Message Queue

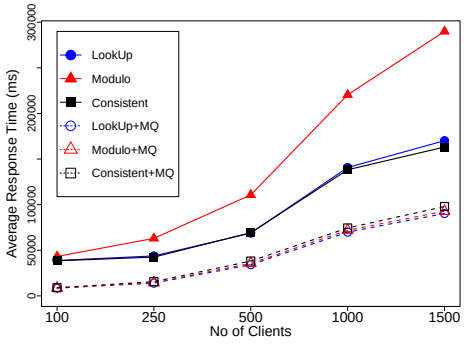
menting a combination of Local Sharding-Based Router and Priority Message queue patterns (see Figure 10). The effect size is large in all three cases as shown on Table 5. Consequently, we reject H^2_{x7} for all Ex ($x \in \{4 \dots 6\}$).

Results from Table 5 show that Local Database Proxy combined with Priority Message Queue does not significantly increase the amount of energy consumed by an application (see Figure 9). Therefore, we cannot reject H^1_{x7} for all Ex ($x \in \{1 \dots 3\}$). 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 10, 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.

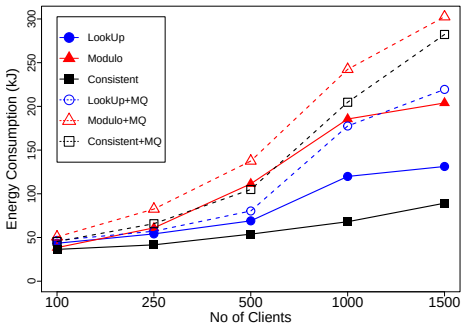
When the Local Sharding-Based Router pattern is combined with Local Database Proxy, the average response time of the application decreases significantly (see Figure 11 and Table 5). Conversely, statistical tests from Table 5 and trends on Figure 11 show that Local Sharding-Based Router pattern combined 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 Priority Message Queue, the average response time of the application decreases slightly, but not significantly, as shown on Figure 9. However, when the Local Sharding-Based Router pattern is combined with a Priority Message Queue, the response time of the application improves significantly (see the response time values of E4 E6 in Table 5). The effect sizes are greater than 0.6, for all three sharding algorithms.

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 an 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*.

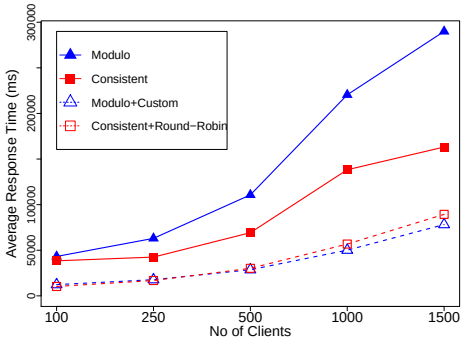


(a) Average Response Time

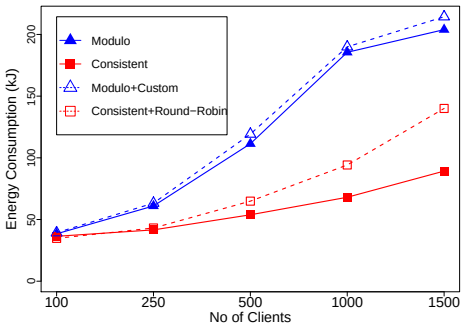


(b) Energy Consumption

Figure 10: The Impact of Local Sharding-Based Router combined with Priority Message Queue



(a) Average Response Time



(b) Energy Consumption

Figure 11: The Impact of Local Sharding-Based Router combined with Local Database Proxy

5.2. Results of Experiment 2

Table 6 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.

Average amount of consumed energy: Results presented in Table 6 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 of the energy efficiency (*i.e.,* a reduction of the energy consumption). Therefore, we reject H_x^1 for all Ex ($x \in \{1 \dots 7\}$), 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.

Average response time: Results presented in Table 6 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 ($x \in \{1 \dots 11\}$). 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 12, 13, 14 and 15 show the improvements on response time, that can be achieved with the patterns.

In Figure 12, we can see that the Local Database 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 that the Local Database Proxy pattern can improve the performance of an application without sacrificing its energy efficiency. The Local Sharding-Based Router pattern also improves both the performance (see Table 6) and the energy efficiency of applications (see Figure 13). However, the effect size is medium, *i.e.*, 0.45375, 0.40125 and 0.54 respectively.

The Gatekeeper pattern can improve both performance and energy consumption only when it is implemented in a microservices application (see Figure 14). 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

Table 6: p-value of Mann–Whitney U test and Cliff's δ effect size for the second application, individual patterns

Version	Response Time		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
E1, E2	0.6983	-0.05125	0.92	0.01375
E1, E3	0.2323	-0.15625	1.0	0.00005
E1, E4	**<0.05**	-0.65875	0.4755	-0.09375
E1, E5	**<0.05**	-0.665	0.116	-0.205
E1, E6	**<0.05**	-0.5125	0.9962	-0.00125
E1, E7	**<0.05**	-0.75125	**<0.05**	0.325
E1, E8	**<0.05**	-0.9975	**<0.05**	-0.47
E1, E9	**<0.05**	-0.46875	0.05708	0.2475
E1, E10	**<0.05**	-1.0	**<0.05**	-0.4975
E1, E11	**<0.05**	-0.59	0.0971	0.21625
E2, E3	0.2894	-0.13875	0.8445	0.02625
E2, E4	**<0.05**	-0.6125	0.3354	-0.12625
E2, E5	**<0.05**	-0.60625	0.1011	-0.21375
E2, E6	**<0.05**	-0.46	0.9048	-0.01625
E2, E7	**<0.05**	-0.72125	**<0.05**	0.325
E2, E8	**<0.05**	-0.98625	**<0.05**	-0.48
E2, E9	**<0.05**	-0.45	**<0.05**	0.26
E2, E10	**<0.05**	-1.0	**<0.05**	-0.52
E2, E11	**<0.05**	-0.51875	**<0.05**	0.26
E3, E4	**<0.05**	-0.4475	0.3601	-0.12
E3, E5	**<0.05**	-0.475	-0.23	
E3, E6	**<0.05**	-0.3175	0.9504	-0.00875
E3, E7	**<0.05**	-0.57625	**<0.05**	0.336
E3, E8	**<0.05**	-0.9325	**<0.05**	-0.48
E3, E9	**<0.05**	-0.33875	**<0.05**	0.257
E3, E10	**<0.05**	-0.9775	**<0.05**	-0.51
E3, E11	**<0.05**	-0.34	**<0.05**	0.25875
E4, E5	0.8595	-0.02375	0.3306	-0.1275
E4, E6	0.2598	0.1475	0.268	0.145
E4, E7	0.6702	-0.05625	**<0.05**	0.3475
E4, E8	**<0.05**	-0.705	**<0.05**	-0.4437
E4, E9	0.1031	0.2125	**<0.05**	0.3025
E4, E10	**<0.05**	-0.7675	**<0.05**	-0.47
E4, E11	0.1116	0.2075	**<0.05**	0.30875
E5, E6	0.3073	0.13375	0.07575	0.23125
E5, E7	0.8295	-0.02875	**<0.05**	0.3975
E5, E8	**<0.05**	-0.665	**<0.05**	-0.4012
E5, E9	0.1682	0.18	**<0.05**	0.3525
E5, E10	**<0.05**	-0.72625	**<0.05**	-0.4
E5, E11	0.05336	0.25125	**<0.05**	0.37875
E6, E7	0.1901	-0.17125	**<0.05**	0.3275
E6, E8	**<0.05**	-0.75875	**<0.05**	0.49
E6, E9	0.4464	0.1	**<0.05**	0.2875
E6, E10	**<0.05**	-0.82125	**<0.05**	-0.5
E6, E11	0.4755	0.09375	**<0.05**	0.261
E7, E8	**<0.05**	-0.75875	**<0.05**	-0.55
E7, E9	0.0859	0.22375	0.4238	-0.105
E7, E10	**<0.05**	-0.835	**<0.05**	-0.63
E7, E11	**<0.05**	0.3425	**<0.05**	-0.171
E8, E9	**<0.05**	0.81	**<0.05**	0.54875
E8, E10	0.2894	0.13875	0.4875	-0.09125
E8, E11	**<0.05**	0.86	**<0.05**	0.52125
E9, E10	**<0.05**	-0.84	**<0.05**	-0.5887
E9, E11	0.6494	0.06	0.6356	-0.0625
E10, E11	**<0.05**	0.95125	**<0.05**	0.56

Table 7: p-value of Mann–Whitney U test and Cliff's δ effect size for the second application implementing a combination of patterns

Version	Avg. Response Time		Avg. 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

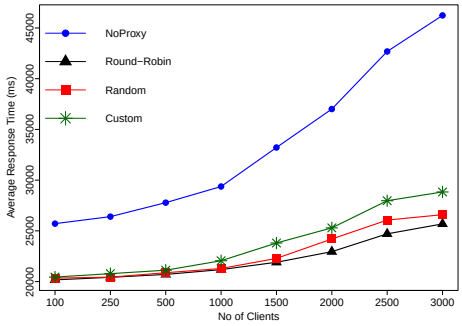
application follows a microservices architectural style (see Figure 15).

The Pipes & Filters pattern can have a positive impact on both performance and energy consumption. Figure 16 shows the results obtained in our experiment. The average response time of the application decreases (the trend levels off when the number of clients increases) 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 6). We highly recommend that developers use this pattern since it significantly improves both response time and energy efficiency. Table 7 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 the Local Database Proxy pattern with the Priority Message Queue pattern, no remarkable difference was observed on the energy utilization of the application (in comparison to the version with only the Local Database Proxy pattern). We see only a slight uptick on the energy consumption curve from Figure 17. The Mann–Whitney U tests from Table 7 show that the observed differences are not statistically significant (p-values are high: 0.6019, 0.3354 and 0.4875).

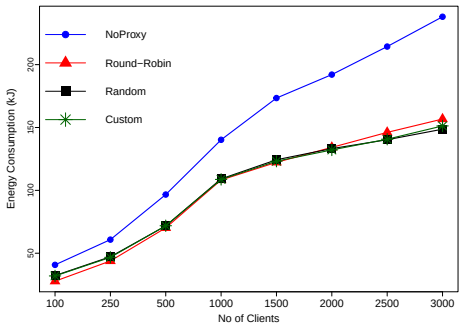
On the contrary, joining the Local Database Proxy patten with Priority Message Queue has a negative impact on the average response time (see the curves on Figure 17).

The Mann–Whitney U tests from Table 7 confirm that these observed differences are statistically significant. Hence, we conclude that the combination of Local Database Proxy and Priority Message Queue patterns can have a negative impact on the performance of an application. The effect sizes vary from medium to large.

Therefore, although the Local Database Proxy pattern can help improve the scalability of an application, a combination with the Priority Message Queue pattern can result in a performance degradation. Regarding the Local Sharding-Based Router pattern, a combination with Priority Message Queue does not have a negative impact on response time or energy efficiency (see Figure 18). The slight difference observed on the energy consumption curves is not statistically significant. The response time is improved

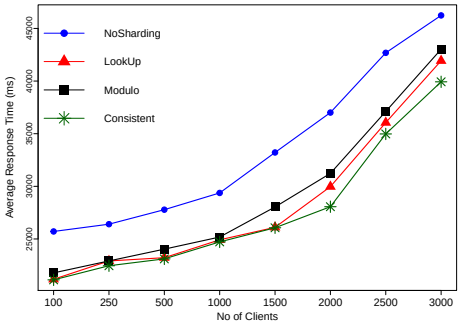


(a) Average Response Time

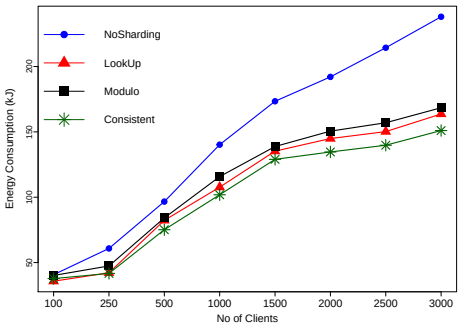


(b) Energy Consumption

Figure 12: The Impact of the Local Database Proxy Pattern



(a) Average Response Time



(b) Energy Consumption

Figure 13: The Impact of Local Sharding-Based Router Pattern

when the Priority Message Queue pattern is added to the Local Sharding-Based Router pattern.

To simulate customers that require extra care (*i.e.,* the heavyweight approach), we have implemented in our application a combination of Local Sharding-Based Router 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 Local Sharding-Based Router and Pipes and Filter patterns can significantly improve the performance of an application (see Table 7), however, this will be detrimental of the energy efficiency, as shown on Figure 19.

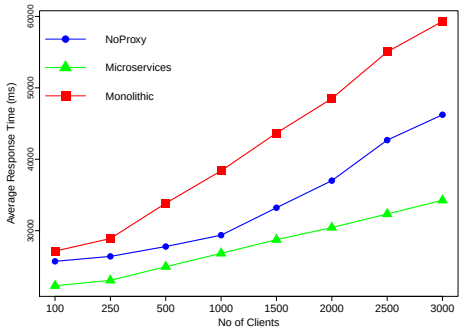
One advantage of the Competing Consumers pattern is the fact that it enables scaling up the application, adding more resources to the computation and data handling services, as needed. In order to test the efficiency of this pattern, we designed a mixed combination of the Local Sharding-Based Router, 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.

We observed that the combination of these three patterns did not significantly increased the amount of energy consumed by the application (see Figure 20). However, the response time was impacted (we obtained significant p-values, and effect sizes of around 0.3, see Table 7). 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.

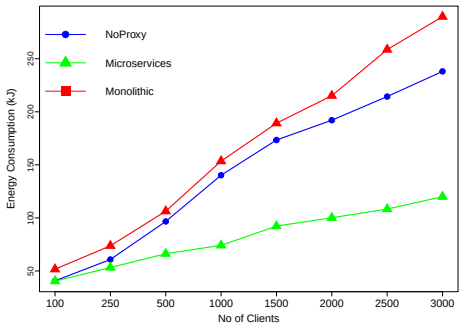
Since in a typical cloud-based application, developers use a Gatekeeper to ensure that only authorized requests are processed, we experimented with a combination of Gatekeeper, Sharding-Based Router, Pipes and Filters, and Competing Consumers patterns. Results presented in Figure 21 shows that the resulting 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 7).

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 Local Database Proxy and Priority Message Queue patterns can have a negative impact on the performance of an application, but not on energy efficiency. However, a combination of Priority Message Queue and Sharding-Based Router patterns does not have a negative impact neither on response time or energy efficiency. When the Sharding-Based Router pattern is combined with Pipes and Filters, the performance of

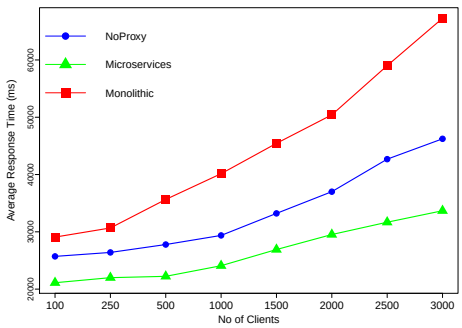


(a) Average Response Time

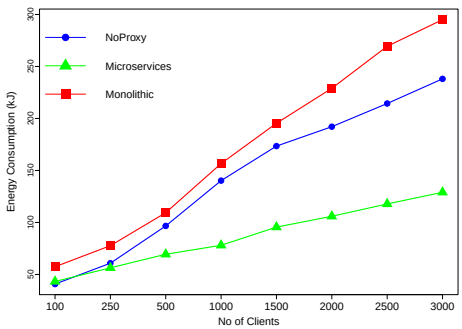


(b) Energy Consumption

Figure 14: The Impact of Gatekeeper Pattern

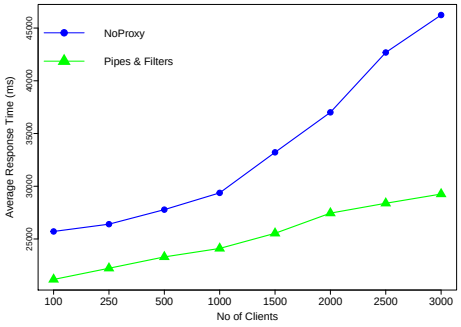


(a) Average Response Time

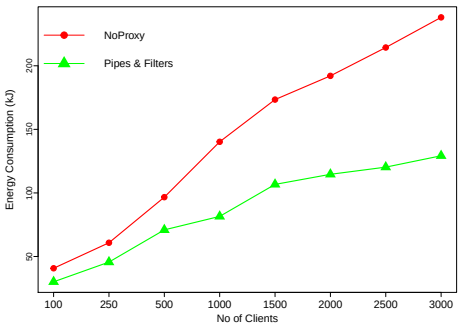


(b) Energy Consumption

Figure 15: The Impact of Competing Consumers Pattern

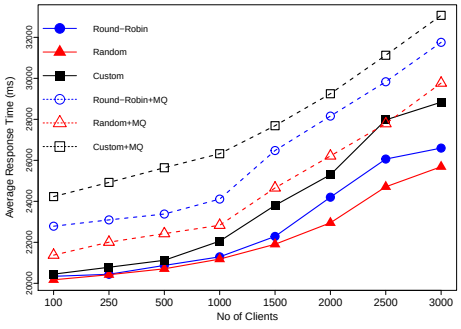


(a) Average Response Time

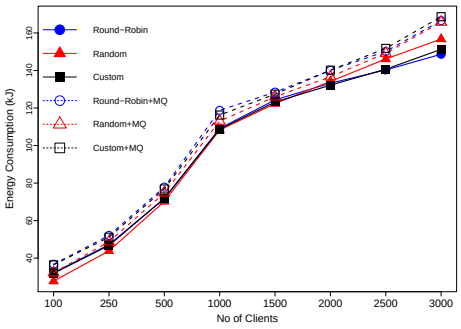


(b) Energy Consumption

Figure 16: The Impact of the Pipes & Filters Pattern

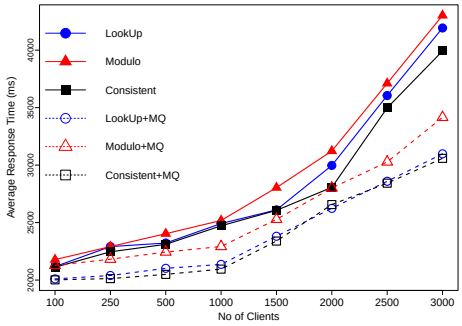


(a) Average Response Time

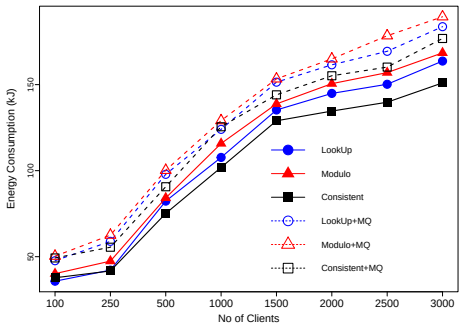


(b) Energy Consumption

Figure 17: The Impact of Local Database Proxy combined with Priority Message Queue

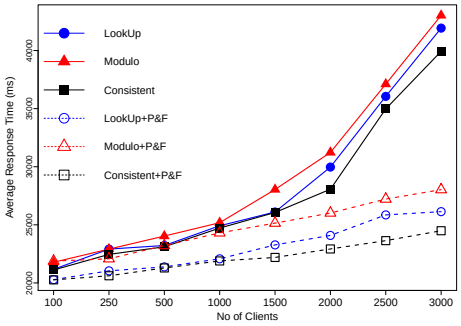


(a) Average Response Time

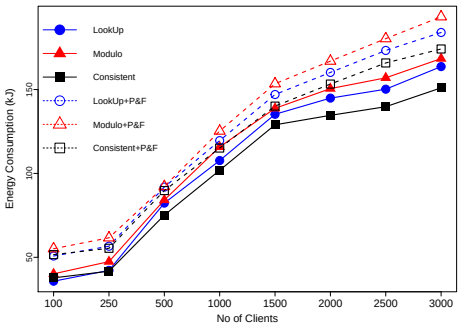


(b) Energy Consumption

Figure 18: The Impact of Local Sharding-Based Router combined with Priority Message Queue



(a) Average Response Time



(b) Energy Consumption

Figure 19: The Impact of Local Sharding-Based Router combined with Pipes & Filters

the application can improve significantly. However, this improvement can be at the expense of energy efficiency. A combination of Sharding-Based Router, 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.

6. Discussion

In general, there appear 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 8 summarizes the impact on response time and energy efficiency of the six studied patterns. The Message Queue pattern is not mentioned in Table 8 because this pattern is generally used in combination with other patterns.

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. Local Database 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 Local Database Proxy and Competing Consumers patterns can improve response time and energy consumption (see Table 8).

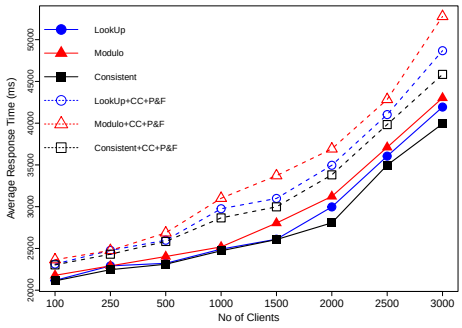
Data Management: Cloud infrastructures offer the possibility to store and access large amounts of data quickly. Sharding algorithms can have a positive impact on increasing productivity, specially in No-SQL databases which are designed to bring scalable data storage to cloud. We observed that the two 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

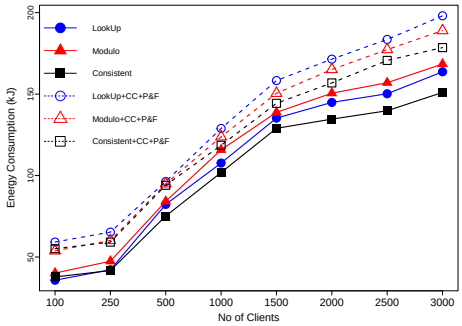
Table 8: Impact of Patterns on Energy Efficiency and Response Time*

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

*The Message Queue pattern is not mentioned because this pattern is generally used in combination with other patterns, and we did not analyze the impact of an isolated implementation of this pattern.

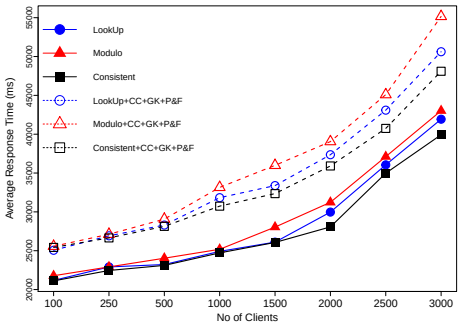


(a) Average Response Time

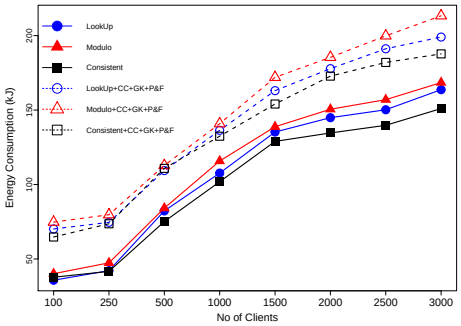


(b) Energy Consumption

Figure 20: The Impact of Local Sharding-Based Router combined with Competing Consumers and Pipes & Filters Patterns



(a) Average Response Time



(b) Energy Consumption

Figure 21: The Impact of Local Sharding-Based Router combined with Competing Consumers, Gatekeeper and Pipes & Filters Patterns

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: 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 high performing 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.

Developers and software architects can consult the guidelines from Table 9 to select patterns among our six studied cloud patterns during the development of their applications.

7. 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.* [32]:

Table 9: Guidelines for selecting the six patterns

Application's most important non-functional requirement	**Local Database Proxy**	**Local Sharding-Based Router**	**Priority Message Queue**	**Competing Consumers**	**The Gatekeeper**	**Pipes and Filters**
Security	–	–	–	–	☺	–
Energy efficiency	☺	☹	☺	☹	☹	☺
Performance	☺	☹	☺	☹	☹	☺
Scalability	☺	☺	☺	☺	–	☺

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 that could be induced by perturbations on the network or the hardware, and our tracing. The PowerAPI 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, the tool used to measure energy consumption (*i.e.,* PowerAPI), and the configuration of our cloud environment.

External validity threats concern the possibility to generalize our findings. These results have to be interpreted carefully as they may depend on the specific set up of our experiments. However, since we obtained consistent results from the two experiments that involved applications implemented by two different teams in two different programming languages (Java and Python), we believe the findings to be robust. Nevertheless, further validations with different applications and possibly more patterns, are desirable to improve our understanding of the impact of cloud patterns on the energy consumption of 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 paper 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.

8. Conclusion

In this study, we examine the impact on energy consumption of six cloud patterns (*i.e.,* Local Database Proxy, Local Sharding-Based Router, the Priority Message Queue, Competing Consumers, 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 Local Database 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-Based Router, 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.

The study presented in this paper 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.

References

[1] F. Prosperi, M. Bambagini, G. Buttazzo, M. Marinoni, 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.

[2] I. El Korbi, S. Zeadally, Energy-aware sensor node relocation in mobile sensor networks, Ad Hoc Networks 16 (2014) 247–265.

[3] G. Hecht, B. Jose-Scheidt, C. De Figueiredo, N. Moha, 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.

[4] A. Bourdon, A. Noureddine, R. Rouvoy, L. Seinturier, PowerAPI: A Software Library to Monitor the Energy Consumed at the Process-Level, ERCIM News 92 (2013) 43–44.
URL `https://hal.inria.fr/hal-00772454`

[5] S. A. Abtahizadeh, F. Khomh, Y.-G. Guéhéneuc, How green are cloud patterns?, in: Computing and Communications Conference (IPCCC), 2015 IEEE 34th International Performance, IEEE, 2015, pp. 1–8.

[6] G. Pinto, F. Castor, Y. D. Liu, Mining questions about software energy consumption, in: MSR, 2014, pp. 22–31.

[7] C. Pang, A. Hindle, B. Adams, A. E. Hassan, What do programmers know about software energy consumption?, IEEE Software 33 (3) (2015) 83–89.

[8] A. Ampatzoglou, G. Frantzeskou, I. Stamelos, A methodology to assess the impact of design patterns on software quality, Information and Software Technology 54 (4) (2012) 331–346.

[9] F. Khomh, 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.

[10] M. Vokáč, W. Tichy, D. I. Sjøberg, E. Arisholm, 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 9 (3) (2004) 149–195.

[11] L. Prechelt, B. Unger, W. F. Tichy, P. Brossler, L. G. Votta, A controlled experiment in maintenance: comparing design patterns to simpler solutions, Software Engineering, IEEE Transactions on 27 (12) (2001) 1134–1144.

[12] K. Aras, T. Cickovski, J. A. Izaguirre, Empirical evaluation of design patterns in scientific application, Technical Report TR-2005-08, Department of Computer Science and Engineering, University of Notre Dame.

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

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

[15] R. Burtica, E. M. Mocanu, M. I. Andreica, 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.

[16] R. Cattell, Scalable sql and nosql data stores, ACM SIGMOD Record 39 (4) (2011) 12–27.

[17] K. Sachs, S. Kounev, J. Bacon, A. Buchmann, Performance evaluation of message-oriented middleware using the specjms2007 benchmark, Performance Evaluation 66 (8) (2009) 410–434.

[18] A. Beloglazov, Energy-efficient management of virtual machines in data centers for cloud computing, Ph.D. Thesis, Department of Computing and Information Systems, The University of Melbourne.

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

[20] S. S. Mahmoud, I. Ahmad, A green model for sustainable software engineering, International Journal of Software Engineering and Its Applications 7 (4) (2013) 55–74.

[21] C. Calero, M. Piattini, Green in Software Engineering, Springer, 2015.

[22] 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.

[23] S. Strauch, V. Andrikopoulos, U. Breitenbuecher, O. Kopp, 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.

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

[25] M. Goraczko, A. Kansal, J. Liu, F. Zhao, Joulemeter: Computational energy measurement and optimization (2011).

[26] K. Liu, G. Pinto, 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.

[27] D. J. Sheskin, Handbook of parametric and nonparametric statistical procedures, crc Press, 2003.

[28] J. Romano, J. D. Kromrey, J. Coraggio, 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.

[29] J. Cohen, Statistical power analysis for the behavioral sciences, Lawrence Erlbaum Associates, Inc, 1977.

[30] A. Dmitrienko, G. Molenberghs, C. Chuang-Stein, W. Offen, Analysis of Clinical Trials Using SAS: A Practical Guide, SAS Institute, 2005.
URL `http://www.google.ca/books?id=G5ElnZDDm8gC`

[31] N. Cliff, Dominance statistics: Ordinal analyses to answer ordinal questions, Psychological Bulletin 114 (3) (1993) 494.

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