Journal of the Brazilian Computer Society, 2023, 29:1, doi: 10.5753/jbcs.2023.3271

Non-Intrusive Continuous Monitoring of Smart City Platforms

João Victor Lopes [**Federal University of Rio Grande do Norte** | ***joao.victor_lopes27@outlook.com***]
Everton Cavalcante [**Federal University of Rio Grande do Norte** | ***everton.cavalcante@ufrn.br***]
Thais Batista [**Federal University of Rio Grande do Norte** | ***thaisbatista@gmail.com***]
André Solino [**Federal University of Rio Grande do Norte** | ***andresolino@imd.ufrn.br***]
Jorge Pereira [**Federal University of Rio Grande do Norte** | ***jorgepereirasb@gmail.com***]
Aluizio Rocha Neto [**Federal University of Rio Grande do Norte** | ***aluzio@imd.ufrn.br***]

Universidade Federal do Rio Grande do Norte, Campus Universitário Lagoa Nova, 59078-900, Natal, Rio Grande do Norte, Brasil.

Received: 23 March 2023 • **Accepted:** 31 October 2023 • **Published:** 08 December 2023

Abstract Smart city platforms provide several services to facilitate the development of applications. Such platforms typically manage several applications, deal with a large volume of data, and serve many devices and users that generate a high volume of requests. The large number of requests to handle and the complex operations to perform often cause overloads on the platform, degrading the quality of service provided to users and applications. In this context, monitoring the underlying computational infrastructure in which smart city platforms and applications are deployed and the platform operations is essential. The monitoring process can allow for examining fluctuations in the behavior of the platform's components to detect performance degradation and overloads (including unforeseen ones), contribute to avoiding interruptions in the platform's services, and increase its scalability to assimilate significant amounts of requests, devices, and users. This paper presents a strategy and architecture to enable the non-intrusive monitoring of operations on smart city platforms and their underlying infrastructure. The proposal covers monitoring at multiple levels and is based on the aspect-oriented programming (AOP) paradigm so that it is possible to monitor the platform's operations without intervening in the platform's implementation or generating coupling regarding monitoring. This paper presents the implementation of the monitoring architecture and its instantiation in the context of *Smart Geo Layers* (SGeoL), a platform that has been used in several real-world smart city applications. This paper also reports the results of computational experiments to evaluate the performance of the proposed monitoring architecture for response time to requests, CPU usage, and RAM utilization. The obtained results show an evident increase in response time with the number of simultaneous requests and a significant correlation between the response time and the CPU utilization in the deployment of the monitoring architecture.

Keywords: Smart cities, Platform, Monitoring, Aspect-oriented programming

1 Introduction

Smart cities can be understood as urban spaces that rely on modern Information and Communication Technologies and data to offer a better living environment for those in the city (citizens, companies, visitors) while providing better services and responding to several environmental, economic, and social challenges [ISO, 2019]. Technological development in recent years and the emergence of ubiquitous computing, cyber-physical systems, and the Internet of Things transformed smart cities into fully connected environments supported by many applications and software systems [Cavalcante *et al.*, 2017].

Smart city platforms arise as integrated environments supporting developers in designing, implementing, deploying, and managing smart city applications [Santana *et al.*, 2017], besides serving several users such as citizens, government agencies, and companies. These platforms are based on middleware offering reusable distributed services to users and applications, seamlessly integrating devices and systems, and handling various demands across the city.

The literature presents several platforms proposed to address challenges in developing and deploying applications and services for smart cities [Santana *et al.*, 2017]. Nonetheless, the scalability of these solutions has not been the focus of research in this context [Del Esposte *et al.*, 2019]. Scalability is an essential concern for smart city platforms as they must deal with a growing number of users, devices, services, and applications. They also store and process large volumes of data related to the city and need to support thousands of requests from users and applications using their services. Furthermore, the workload requested by the platform may be variable according to the characteristics of applications, users, and the city. Despite all these facets, smart city platforms need to maintain their operations at an acceptable quality of service.

Monitoring the operation of a smart city platform and its underlying computational infrastructure involves examining fluctuations in the behavior of the platform's components to detect performance degradation and anomalies. This can help detect overloads (including unforeseen ones), avoid an interruption of the platform's services, and increase its scalability to assimilate significant amounts of requests, devices, and users. Automated monitoring can also support the fast identification of bottlenecks and promptly trigger actions, e.g., (re)allocation of resources in the underlying infrastructure to better accommodate the current demand for the platform. However, the existing monitoring strategies for smart city platforms are intrusive. For open-source platforms, they need to have their source code modified to support monitoring, implying a non-negligible development effort.

Our previous work [Solino *et al*., 2022] introduced a strategy based on *aspect-oriented programming* (AOP) [Kiczales *et al*., 1997] to support monitoring smart city platforms. In our strategy, AOP enables continuous monitoring of the platform's operation and the use of resources from its underlying computational infrastructure without intervening in its implementation or generating coupling to the monitoring architecture. This paper goes a step further by proposing an architecture to continuously monitor the operations of a smart city platform and its infrastructure. It is worthwhile emphasizing that our purpose is to provide a comprehensive monitoring solution able to cover both the platform operations and the underlying infrastructure supporting it. This is an important contribution from our work, considering that most existing proposals focus on monitoring infrastructure elements such as virtual machines and containers, and monitoring application-level components requires significant changes to the platform's source code.

We implemented the architecture relying on the AOP-based strategy and validated it with *Smart Geo Layers* (SGeoL) [Pereira *et al*., 2022], a real-world georeferenced platform for smart cities that integrates heterogeneous data from different domains of a city. This paper also reports computational experiments to evaluate the monitoring architecture's performance, which we have not done so far. These experiments considered scenarios with different workloads and measured response time to requests, CPU usage, and RAM utilization. The obtained results showed an evident increase in response time with the number of simultaneous requests and a significant correlation between the response time and the CPU utilization in the deployment of the monitoring architecture.

The remainder of this paper is structured as follows. Section 2 provides the background to this work. Section 3 presents our AOP-based monitoring architecture and the considered metrics. Section 4 details the implementation of the proposed monitoring architecture. Section 5 presents and discusses the evaluation results. Section 6 discusses related work. Section 7 brings final remarks.

2 Background

This section provides some background about virtualizing and monitoring cloud-based environments (Section 2.1), monitoring approaches for smart city platforms (Section 2.2), and AOP (Section 2.3).

2.1 Virtualizing and monitoring cloud-based environments

Smart city platforms are typically deployed in virtualized cloud-based computational infrastructures. The traditional deployment of cloud applications through virtualization based on *virtual machines* (VMs) generates performance overhead and prevents this type of virtualization in high-performance computing environments. This limitation led to container-based virtualization, considered a lighter alternative to VMs [Bhardwaj and Krishna, 2021]. *Containers* are isolated environments where users can deploy applications and all the dependencies (libraries, binaries, and basic configurations) required for proper execution. Therefore, container-based virtualization is faster for handling computational resources and can meet the increasingly dynamic, variable workload [Taherizadeh and Stankovski, 2019]. Other benefits are reduced consumption of CPU and memory resources, speed of deployment, and the ability to scale containers to meet the growing demand.

Monitoring a Cloud Computing environment inherits traditional monitoring techniques like network traffic and computer resources. However, it tends to be more complex as multiple entities need to be monitored, including the use of resources of the VM infrastructure (processor, RAM, disk, and network), software layers and databases, and entities related to the user experience. It is necessary to monitor all these entities simultaneously to coordinate the use of resources in the Cloud Computing environment according to the demand of applications [Ma *et al*., 2013]. Monitoring primarily focuses on infrastructure metrics such as CPU load and RAM consumption. In addition to these metrics, application metrics should also be considered when deciding whether to scale any resource in cloud computing environments [Kampars and Pinka, 2017]. The infrastructure metrics must be preprocessed and combined with the application metrics before performing the required operations to scale the resources.

According to Coutinho *et al*. [2015], the most used metrics in Cloud Computing environments move toward resources, e.g., CPU usage and RAM utilization, the former being the most common metric. It is also possible to notice that the transfer rate (represented in requests per second or megabytes per second) remains among the most used. Although energy use is not widely used, some works in the literature attempt to use it for energy-saving purposes in Cloud Computing environments.

2.2 Monitoring approaches for smart city platforms in Cloud Computing environments

From existing work in the literature (see Section 6), we propose a framework for classifying monitoring approaches for smart city platforms and their respective underlying infrastructure in Cloud Computing environments. Our framework comprises four groups, as illustrated in Figure 1. *Location* refers to where the monitoring occurs, whether in a single centralized node or distributed across multiple nodes. *Levels* refer to the number of objects considered in monitoring, whether VMs, containers, or applications. *Coordination* is related to whether the monitored metrics are combined or not to adjust resources supporting the platform operation. *Intrusiveness* defines whether the approach requires modifications in the application's code.

Distributed monitoring approaches run the monitoring components into the components to be monitored, and each

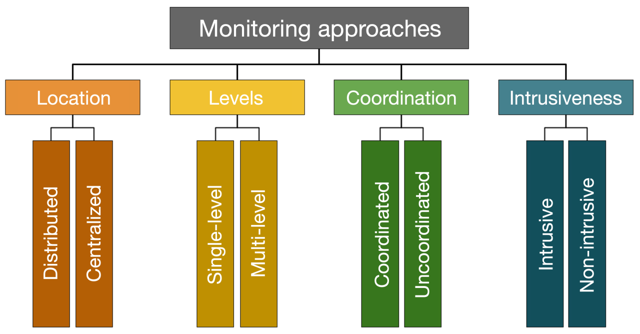

Figure 1. A classification framework for monitoring approaches.

component is responsible for monitoring itself. Therefore, the monitored components can independently identify faults and take action to fix them. On the other hand, *centralized monitoring* approaches have a central element responsible for the monitoring process. In this case, it is possible to use distributed monitoring agents to collect data from each component to be monitored, and such agents send the data to the central monitoring component.

Single-level monitoring uses only one-level metrics, e.g., CPU, memory, disk, and network usage metrics of containers. The level is defined according to the object to be monitored. If only containers are monitored, then the monitoring approach is at a single level, even if multiple metrics from that level are used. Single-level monitoring approaches are typically easier to implement and deploy as they require less effort from developers and operators. In turn, *multi-level monitoring* approaches use metrics focusing on more than one level, for example, container metrics (CPU usage and others) and application metrics (application response time and others). Containers and the application are monitored, i.e., two levels are monitored using metrics from both levels.

Coordinated monitoring approaches aim to use monitoring metrics in a coordinated way to carry out some adjustment action. In a coordinated monitoring approach that leverages CPU usage and disk usage, it is possible to scale a container through CPU usage and track disk usage to verify if new containers can scale. *Uncoordinated monitoring* approaches tend not to use monitoring metrics in a coordinated way, i.e., metrics are used individually to perform some adjustment action. In this case, containers can be scaled according to the CPU usage, but metrics are not combined to cover some particularities that can happen in a Cloud Computing environment. Some specific situations that cannot be overcome through uncoordinated monitoring approaches are (i) disk growth upon increasing the number of containers and (ii) allocating more CPU as application response time increases.

Intrusive monitoring approaches do not concern tangling the application code with the monitoring code. For this reason, they generally have a more significant coupling in the monitoring code with the monitored element code, which generates dependencies. These approaches are also more challenging to maintain as the monitored element code becomes more complex when mixed with the monitoring code. In *non-intrusive monitoring* approaches, the monitored element code is separate from the monitoring code. In this case, it is possible to decouple the application code and ease the maintenance of the monitoring code since it will not be mixed with the application code.

2.3 Aspect-oriented programming

Aspect-Oriented Programming (AOP) [Kiczales *et al*., 1997] proposes a solution for the modularization and composition of concerns that are transversal to several parts of a system, the so-called *cross-cutting concerns*. A cross-cutting concern refers to a particular part of a program that affects many other parts. Consequently, its implementation is dispersed in several components responsible for the basic functionalities of the system (base code), thereby violating basic principles of software design. Common examples of cross-cutting concerns are logging, authentication, exception handling, and persistence. AOP enables separating the basic functionalities of an application from the cross-cutting concerns, increasing its reusability and fostering its maintenance and readability.

The AOP abstractions for representing cross-cutting concerns are *aspects*, *join points*, *pointcuts*, and *advices*. Aspects implement and encapsulate cross-cutting concerns through instructions about where, when, and how they are invoked. An aspect promises greater modularity by avoiding tangling the code of the system's functionalities with the code of the cross-cutting concerns. Join points are well-defined places in the structure or execution flow of a program where behaviors defined by aspects can be inserted. Pointcuts combine a set of join points with objects and methods affected by one or more cross-cutting concerns [Kiczales and Mezini, 2005].

An advice is a set of operations performed when join points are reached. Each advice usually has a pointcut associated with it, determining the join points where this advice will be executed. Advice statements can be of three types: (i) *before*, to execute the advice when a join point is reached, e.g., before invoking a method; (ii) *after*, to execute the advice when control returns through the join point, i.e., after invoking a method; and (iii) *around*, to execute the advice when the pointcut is reached, with precise control over when the affected method should be executed.

While the application code base is implemented using a traditional programming language, aspects are implemented using an aspect-oriented language such as AspectJ [Kiczales *et al*., 2001], an extension of the Java programming language to AOP, or, more recently, the AOP-powered Spring Framework[1]. Listing 1 presents a simple code example to manage a list of registered employees for a company that must log when a new employee has been added. Following the AOP paradigm, the code responsible for the business logic (i.e., managing employees) should be separate from the code associated with logging. The *EmployeeService* class represents the base code implemented in Java to add an employee to the registry (line 4) and retrieve the list of registered employees (line 8). Logging (lines 13–24) is an aspect associated with the logging operations, specifically to execute some operations when the *addEmployee* method is called. A named pointcut called *logAddEmployee* (line 14) is declared within the aspect to define the method invocation that will be the target of the aspect, namely the *addEmployee* method in the *EmployeeService* class. Two advices are also declared within

[1]https://spring.io/projects/spring-framework

```
1  public class EmployeeService {
2    private List<String> employees = new
         ArrayList<String>();
3
4    public void addEmployee(String name) {
5       employees.add(name);
6    }
7
8    public List<String> getEmployees() {
9       return employees;
10   }
11 }
12
13 public aspect Logging {
14   pointcut logAddEmployee(String name) : call(
         void base.EmployeeService.addEmployee(
         String)) && args(name);
15
16   before(String name) : logAddEmployee() {
17     System.out.println("Adding employee " +
           name);
18   }
19
20   around(String name) : logAddEmployee() {
21     proceed();
22     System.out.println("Employee " + name + "
           added");
23   }
24 }
```

Listing 1. Example of base code to be intercepted by aspect code implemented in AspectJ

the aspect in association with the pointcut. The before advice (lines 16–18) executes before executing the captured join point, in this case, the body of the *addEmployee* method. The around advice (lines 20–23) surrounds the join point, but it forces executing the *addEmployee* method with the proceed call (line 21) before the execution of its action (line 22). The execution of this code will display when the employee is about to be added and when this operation has been completed.

AOP encompasses a *weaving process* that combines the code defined by the aspect with the base code, injecting the advice at the defined join points. Figure 2 illustrates the weaving process in AOP. An aspect is implemented separately from the base code of the application that will be affected by the pointcuts defined in the aspect. The pointcuts are defined to intercept a set of join points in the base code, being the interaction between it and the aspect defined through the advice. To carry out the composition process, a weaver identifies which join points will be intercepted by the defined pointcuts and produces a composite code resulting from the insertion of the code contained in the advice defined by the aspect in the base code of the application.

Given that the base code and the aspect are defined separately and that the base code does not need to know the aspect code, AOP is quite suitable for situations in which a particular cross-cutting concern needs to be inserted into the application, but an intrusive approach should be avoided as it modifies the base code. These characteristics motivated the choice of AOP to implement the monitoring strategy for smart city platforms in this work.

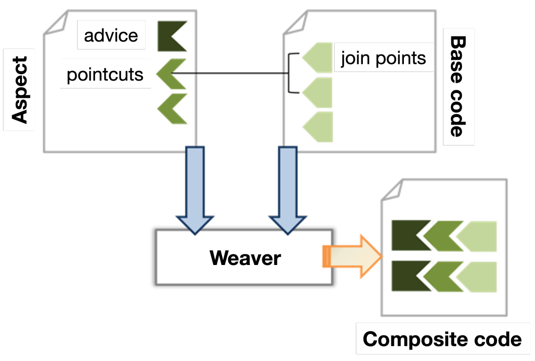

Figure 2. AOP abstractions and the weaving process.

3 Non-Intrusive Monitoring for Smart City Platforms

Our monitoring strategy and architecture consider monitoring metrics about the operation of containers and machines deployed in the computational infrastructure supporting a smart city platform, including measurements of CPU and RAM usage. Aiming at providing a more comprehensive monitoring solution able to cover both the platform and the underlying infrastructure supporting it, our proposal also allows monitoring of the platform's execution, e.g., monitoring the execution time of its internal operations. The non-intrusive continuous monitoring represents the central contribution of this work compared to existing proposals in the literature, which are intrusive in the sense they require modifications in the original implementation of the smart city platform [Araujo *et al.*, 2019; Casalicchio, 2019; Taherizadeh and Stankovski, 2019]. It is also worth mentioning that we do not consider coordinated monitoring since coordination is related to implementing an auto-scaling mechanism, which is out of the scope of this work.

This section describes our monitoring strategy and the proposal of an architecture to realize it. Section 3.1 describes the considered monitoring levels and the metrics for each level. Section 3.2 presents our monitoring architecture.

3.1 Monitoring levels and metrics

Continuous monitoring of a hardware/software infrastructure involves the levels that will be monitored and the measurements of the respective metrics that will be collected at each level. The number of levels may vary depending on the components to be monitored. The levels are associated with the object being monitored, the VMs, and the containers of Cloud Computing environments. Moreover, it is important to consider the application executing inside a container running on a physical machine or a VM.

The three primary levels for monitoring used by Casalicchio [2019] and Taherizadeh and Stankovski [2019] aiming at monitoring infrastructures of Cloud Computing environments are (i) the *host level*, being a physical machine or a VM, (ii) *the container level*, and (iii) *the application level*. The host level generates absolute usage metrics, which provide consumption information close to the actual consumption performed on the machine hosting the entire infrastructure. The container level is responsible for generating relative

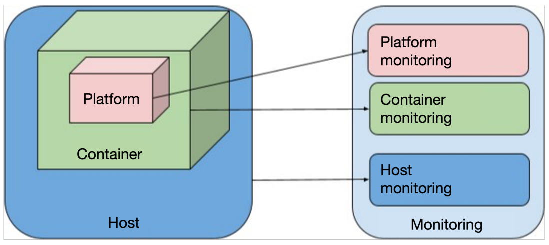

Figure 3. Monitored levels considered in this work.

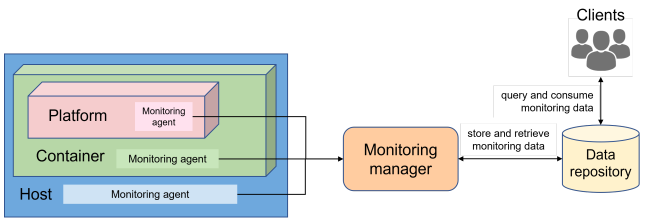

Figure 4. Overview of the monitoring architecture.

usage measures, which do not provide information about the actual consumption of the machine but only show the relative consumption of each container. The application level is responsible for generating usage measures for a specific application, for example, the number of requests made to the application or its response time.

Figure 3 illustrates the levels considered in our monitoring strategy. Regarding the underlying infrastructure supporting the platform, metrics are monitored at two levels, the host and the containers, if used. The host and container levels collect CPU usage, RAM utilization, disk usage, and network bandwidth metrics. In turn, platform-level monitoring can collect metrics such as the time spent executing a particular internal platform operation, e.g., queries to access data managed by the platform.

There are several approaches and tools (including open-source ones) available to accomplish both host and container monitoring levels, such as Elasticsearch[2], Prometheus[3], and NetData[4]. Monitoring at such levels is usually performed by deploying monitoring agents responsible for collecting the desired metrics. These agents can send the monitored metrics to a database server, a messaging service, or another monitoring service. The monitoring solutions available in some studies in the literature consider metrics such as CPU, RAM, network, and HTTP requests, among others [Abranches and Solis, 2016; Matsumoto *et al.*, 2019; Narayana *et al.*, 2020]. These works monitor the host and container levels, but none considers metrics related to monitoring the internal operation of the platform, thereby a differential of this work compared to the state of the art.

3.2 Monitoring architecture

Figure 4 presents an architecture for implementing our monitoring strategy. A *monitoring agent* collects the metrics for each monitored element, either the host, container, or platform. The host and container levels respectively concern absolute and relative usage measurements regarding CPU, RAM, and disk. The platform level concerns usage metrics for the smart city platform, specifically the time of database access operations, time spent for HTTP requests, and transfer rate. The monitoring agents represent tools that can be used to collect usage measurements at specific levels. The *monitoring manager* receives the collected measurements from the monitoring agent and stores them in a *data repository*. Clients can query the data repository to consume stored measurements for further analysis.

[2]https://www.elastic.co/elasticsearch/
[3]https://prometheus.io/
[4]https://www.netdata.cloud/agent/

Host and container levels can be monitored using existing monitoring tools, which serve as monitoring agents. Such monitoring agents connect to the components that must be monitored and collect measurements at those levels. Monitoring the host and the container levels does not require much implementation effort since existing open-source monitoring tools that can be used, such as Node Exporter[5], cAdvisor[6]. To monitor such levels, it is only necessary to learn the different monitoring tools to integrate them via connection URIs to group the metrics to be monitored and sent to the monitoring manager. The monitoring manager is responsible for storing the measurements in a database and retrieving them for activities after the monitoring.

Using a monitoring agent is not enough to monitor the internal structure of the platform since this requires knowledge of such structure. The monitoring agent has the external view of the platform as an opaque component, thus not having access to its internal details. For example, the time spent executing a given platform operation might be the sum of the execution times of other operations called by the first one. Monitoring the internal structure makes it possible to know the time spent in each of these operations individually and identify which is causing eventual performance degradation, but this usually requires modifying the platform's source code. Modifying the platform's source to support monitoring is an intrusive approach since it is necessary to have a more profound knowledge of the platform structure and add the monitoring code at specific points. Our monitoring uses AOP to collect data about the platform without changing it to include monitoring directives [Solino *et al.*, 2022].

Even though the implementation of an intrusive approach could be potentially more straightforward, it is likely to generate technology lock-in, i.e., becoming confined to a specific set of technologies, languages, etc. The use of a non-intrusive monitoring architecture can foster maintainability as it does not generate coupling to the targeted smart city platform. Our strategy of relying on AOP is especially beneficial to avoid the monitoring code spreading throughout the smart city platform's code, causing code tangling and scattering, damaging software modularization, and hampering understandability and maintainability. By avoiding mixing the platform's code with the monitoring code, AOP allows for the separation of crosscutting concerns, resulting in systems that are easier to understand and simpler to handle. Using this approach, no modification to the monitored platform is required, and different monitoring strategies can be (un)plugged in a non-intrusive way. Therefore, it provides the flexibility of easily changing the monitoring strat-

[5]https://github.com/prometheus/node_exporter
[6]https://github.com/google/cadvisor

egy when necessary. The non-intrusive approach is hence useful when the platform source code is inaccessible.

4 Implementation and Validation

We implemented and instantiated our AOP-based monitoring strategy for the SGeoL platform Pereira *et al.* [2022], a scalable platform for developing applications for smart cities. SGeoL offers several functionalities that facilitate the development of applications, such as geographic data management, context information, and data time series, in addition to integrating heterogeneous data and visualization of geographic and analytical information. All these functionalities are provided by APIs and distributed components deployed in VMs and containers. It is worth mentioning that we have considered only SGeoL due to the unavailability of many open-source platforms. As our AOP-based strategy monitors platform operations, having the source code is imperative for instrumentation purposes.

The SGeoL platform deals with a large volume of information, mainly geographic data. It performs complex and costly tasks, such as geographic queries and importing heterogeneous data available through files, spreadsheets, or APIs from other systems. SGeoL has already been overloaded due to several users, devices, and applications requesting simultaneous operations to handle significant amounts of data. Therefore, it is crucial to monitor platform actions and user requests to identify possible performance issues. Monitoring would also make it possible to trigger auto-scaling processes to avoid overloading the platform's infrastructure, thus offering a good experience using its services and avoiding under- or over-provisioning of computational resources in the infrastructure.

This section presents how we instantiated our monitoring architecture to the SGeoL platform to validate the AOP-based monitoring strategy. Section 4.1 describes the platform components and the components responsible for the multi-level monitoring. Section 4.2 details how we used AOP for monitoring at the platform level while not being intrusive to the source code of SGeoL. Section 4.3 presents an API developed to foster the reuse of our monitoring architecture. The current implementation of our AOP-based monitoring is publicly available at https://projetos.imd.ufrn.br/solino/aop-monitoring.

4.1 Instantiating the monitoring architecture with AOP

Figure 5 shows the monitored components of SGeoL and the monitoring components. One of the monitored components is *SGeoL Core*, which aggregates the central platform functionalities and has a significant workload as it provides information from geographic layers to other applications. The other monitored components are the relational and non-relational databases responsible for storing data the platform handles.

We have used three open-source tools to implement the architecture for monitoring the underlying infrastructure on which SGeoL is deployed and runs (host and container). Metricbeat[7] is the monitoring agent responsible for collecting host and container-level metrics, while Filebeat[8] collects platform-level metrics. Collected data are sent to the same database managed by Elasticsearch[9], thus enabling the integration of monitoring data from various levels and providing a holistic view of the platform monitoring. We have chosen these tools because they offer the necessary functionalities to implement the monitoring strategy and can provide the metrics of interest. Furthermore, they are relatively easy to configure and integrate with each other [Bagnasco *et al.*, 2015]. However, our architecture can be implemented using other tools.

4.2 AOP support for monitoring

For monitoring platform-level metrics relying on our AOP-based strategy, it is only necessary to know the interfaces of the operations provided by the platform. Therefore, the aspect associated with monitoring can define the join points for each operation and the respective advices for before and after calling the monitored operation. Through the weaving process, the platform's source code is used with the definitions of the monitoring-related aspects. In our monitoring strategy, the critical operations of the platform are monitored through join points, one per operation every time it is invoked. For each join point, there is an advice responsible for collecting the desired monitoring information, such as the time spent executing each operation individually and, consequently, the total execution time of an operation that involves calling other operations [Solino *et al.*, 2022].

We used AOP to instrument the SGeoL source code written in Java to monitor platform operations and retrieve the necessary information. For example, Listing 2 shows the implementation of the *LayerResourceAspect* aspect in AspectJ. In line 2, the *allMethods* pointcut is created for all methods of the *LayerResource* class. In lines 4–10, an around-type advice is designed for the *allMethods* pointcut to compute the execution time of any method of the *LayerResource* class, corresponding to a platform operation. Every time a method is executed, the execution time is computed and displayed in the standard output to be captured by Filebeat (the monitoring agent) and sent to Elasticsearch. It was unnecessary to change anything in implementing the original *LayerResource* class in the source code of SGeoL, thus demonstrating the benefit of adopting the AOP paradigm to enable monitoring.

The implementation made it possible to validate that (i) our strategy can monitor in a non-intrusive way and (ii) it was possible to obtain metrics from the platform's internal operations that are usually not accessible to monitoring agents. Monitoring has not changed the source code of SGeoL due to adopting the AOP paradigm. Inserting the aspect shown in the example in Listing 2 allowed capturing the monitoring metrics of any method of the *LayerResource* class. Therefore, using AOP was relevant for implementing the monitoring architecture, considering the complexity of this task for smart city platforms, as is the case of SGeoL.

[7]https://www.elastic.co/beats/metricbeat

[8]https://www.elastic.co/beats/filebeat

[9]https://www.elastic.co/elasticsearch/

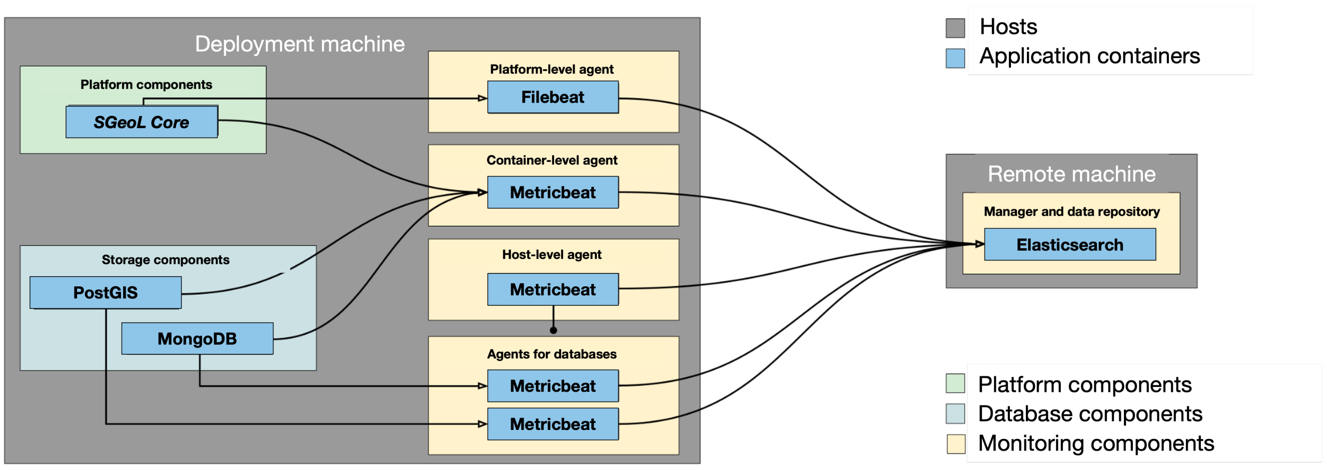

Figure 5. Overview of the monitoring architecture instantiated to monitor components of the SGeoL platform.

```java
public aspect LayerResourceAspect {
  pointcut allMethods() : execution(public *
      LayerResource.*(..))

  around(): allMethods() {
    Long beginTime = System.currentTimeMillis
        ();
    proceed();
    Long endTime  =  System.currentTimeMillis
        ();
    System.out.println("[monitoring] " +
        thisJoinPoint.getSignature() + " " + (
        endTime - beginTime));
    return object;
  }
}
```

Listing 2. AspectJ implementation of the *LayerResource* aspect for monitoring operations of the SGeoL platform

Our monitoring strategy also allowed us to identify the costliest operations regarding response time on the SGeoL platform. SGeoL makes dozens of operations available to its users through its RESTful API, which were monitored using the implementation of the aspect for accounting for the time spent by the platform to process them when accessed by users. The monitoring enabled us to notice that operations dealing directly with geographic processing and data import via vector files are the most demanding ones in terms of time. We also observed that most of the time spent on geographic processing operations is consumed by calls to the database that stores geographic data. This information allowed us to resize the infrastructure supporting this database to obtain better response times specifically for these operations.

4.3 Monitoring API

The monitoring API was developed in Java using the Spring Framework[10], which has libraries for integrating with Elasticsearch and Prometheus. This facility enabled the monitoring agents of both solutions to collect the usage measurements of the underlying infrastructure composed of VMs and containers and the smart city platform operations.

[10]https://spring.io

The monitoring API methods consider a set of parameters. The *level* parameter must be informed to know the level at which usage measurements should be retrieved. The possible values for level are *host*, *container*, or *platform*. The *metric* parameter must be informed to know the metric that must be retrieved. Possible values are *cpu*, *disk*, *memory*, *network*, and *time* so that it is possible to retrieve measurements about the CPU usage, RAM utilization, disk, and network usage of a host or a container, as well as the time spent by a platform operation. The parameter *findby[host|container|method]* must be informed according to the level informed, i.e., if the level is for a container, then the parameter should be *findbycontainer*.

Monitoring API methods may also have request parameters. The parameter *[host|container|method]name* refers to the name of the monitored element according to the level (host, container, platform). *Quantity* refers to the number of measurements to be retrieved by the query to the API. The *Order* parameter specifies the order in which measures should be returned, either ascending or descending. The *start* parameter must be informed to specify the timestamp from which measurements must be retrieved. In conjunction with the *start* parameter, the *end* parameter forms a time interval for which measures must be retrieved. The *agent* parameter provides the monitoring agent for the collected metrics, either *metricbeat* for host and container-level metrics or *filebeat* for platform-level metrics.

The monitoring API allows the client to make relevant queries to know what is happening in the platform's underlying infrastructure (VMs and containers) and observe the platform's behavior regarding the time spent executing its methods to handle requests. For instance, a request with

```
/platform/time/findbymethod?
methodname=findlayer&quantity=50
&order=desc&agent=filebeat
```

retrieves the last 50 measurements of the time spent by the *findlayer* method. In turn, a request with

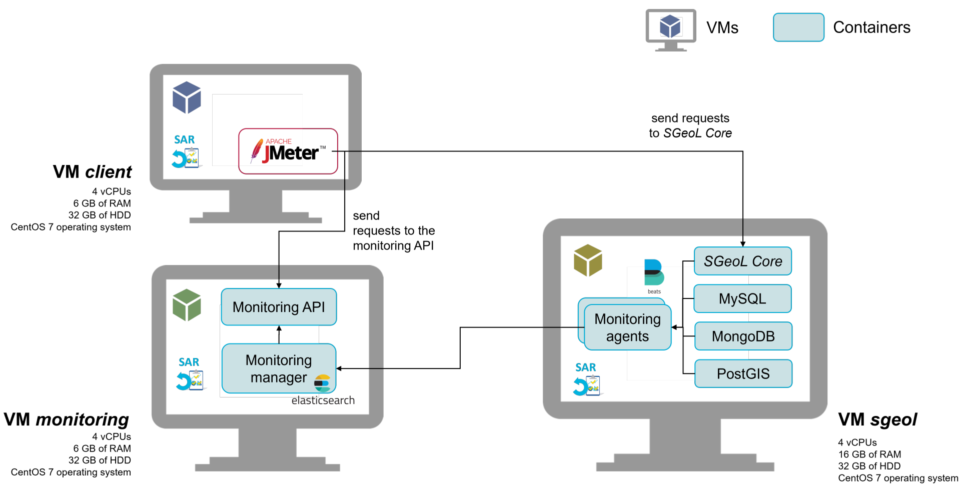

Figure 6. Overview of the monitoring architecture instantiated to monitor components of the SGeoL platform.

```
/host/memory/findwithstartandend?
start=2023-03-10T00:00:00
&end=2023-03-10T23:59:59&hostname=brazil
&quantity=500&order=asc&agent=metricbeat
```

retrieves the first 500 measurements of RAM consumption of a machine named *brazil* for a 24-hour interval on March 10, 2023.

5 Evaluation

This section reports the evaluation of the continuous monitoring of the SGeoL and its underlying infrastructure using our AOP-based strategy and architecture. Following the Goal-Question-Metric (GQM) approach [Basili *et al.*, 1994], our experiments aimed to evaluate the performance of the continuous monitoring implemented on the SGeoL platform and its underlying infrastructure. We aimed to investigate the following question: *What is the performance of continuous monitoring when different workloads are applied to the monitoring API?* To answer this question, we have defined three metrics to be explored in the experiments:

M1: The *response time* for querying the monitoring API
M2: The *CPU usage* of the machines in which the platform components are deployed
M3: The *RAM utilization* of the machines in which the components are deployed

The experiments considered increasing simultaneous requests to overload the monitoring API. This enabled us to assess the performance of the continuous monitoring upon increasing the number of requests regarding resource usage and the time required to process those requests.

Section 5.1 describes the computational infrastructure used in the experiments. Section 5.2 describes a preliminary investigation to determine the optimal interval for periodic monitoring. Section 5.3 presents and discusses the results obtained from the main experiment to assess the performance of the monitoring API.

5.1 Experimental setup

The computational environment to support the experiments comprised three VMs, as depicted in Figure 6. The VM identified as *sgeol* contained the deployment of the *SGeoL Core* and its two databases, as well as the monitoring agents (Filebeat and Metricbeat). The VM identified as *monitoring* contained the monitoring API, the monitoring manager, and the Elasticsearch data repository for storing collected measurements. Each of these elements is deployed in an individual container. The VM identified as the client deployed the Apache JMeter tool[11] that simulates simultaneous client requests to the monitoring API. We also deployed the System Activity Report (*sar*) tool in all the VMs to collect CPU usage and RAM utilization.

5.2 Preliminary experiment: Setting an optimal periodic monitoring

The preliminary experiment aimed to determine the optimal interval for periodic monitoring. We chose the optimal interval as the shortest response time (in milliseconds) for queries made to the monitoring API. The purpose of determining such an interval is to use it as a fixed parameter in the main experiment.

In the preliminary experiment, we varied the interval duration between 1 to 30 seconds (with a one-second step) to make one request to the monitoring API. Following the patterns for querying the monitoring API (see Section 4.3), the request

```
/container/cpu/findbycontainer?
containername=sgeol-dm&quantity=1
&order=desc&agent=metricbeat
```

retrieves a CPU measurement from the container in which *SGeoL Core* is deployed, with Metricbeat as the monitoring agent. While making the request, the SGeoL platform performed a given operation, specifically a geographic query, to retrieve information about city neighborhoods. This operation was performed 50 times to intentionally overload SGeoL

[11] https://jmeter.apache.org

and allow for capturing monitoring information.

We conducted 20 runs for each tested interval and assessed the request's response time to the monitoring API. We also adopted a significance level at $\alpha = 0.05$ in the statistical analysis of the experimental results.

Figure 7 presents the average response time of the monitoring API (in milliseconds) for the intervals tested in the preliminary experiment, namely 1 to 30 seconds. We noticed that most of the observed average response times remained between 20 ms and 30 ms (23.5 ms on average, 95% CI [22.65, 24.35]). The interval with the shortest response time was 11 seconds when the average response time was 19.49 ms, 95% CI [14.22, 24.75].

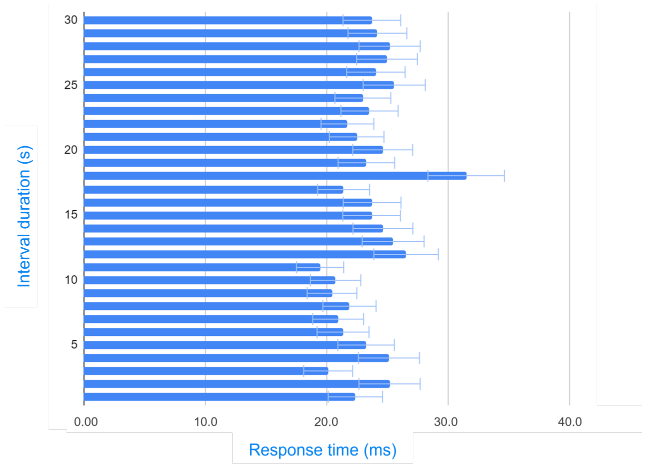

Figure 7. Average response time of the monitoring API for the tested intervals.

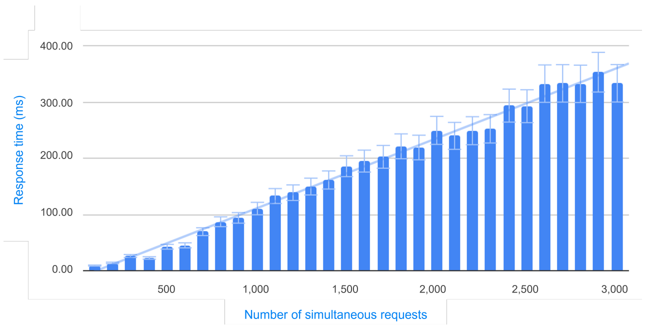

Figure 8. Average response time of the monitoring API for the tested workloads.

5.3 Main experiment: Assessing the performance of the monitoring API

In the main experiment, we collected measurements of CPU usage and RAM utilization of each VM during a varied workload (i.e., number of simultaneous requests) for *SGeoL Core* and the monitoring API. We varied the number of simultaneous requests to the monitoring API from 100 to 3,000, with a step of 100. We also fixed the periodic monitoring interval to 11 seconds, the optimal interval determined in the preliminary experiment (see Section 5.2). The request to the monitoring API and the operation performed by the SGeoL platform were the same as in the preliminary experiment.

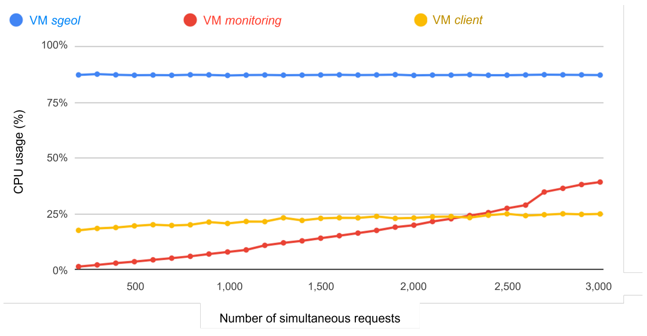

Figure 9. Average CPU usage of the VMs used in the experiment for the tested workloads.

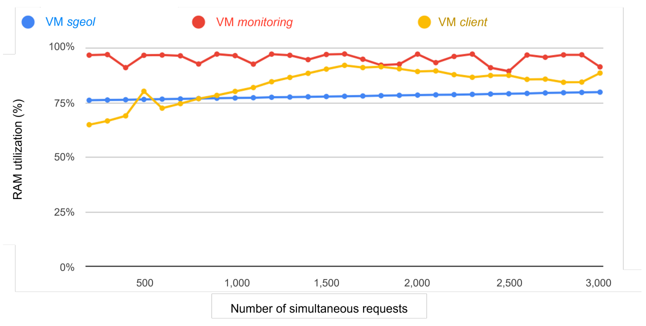

Figure 10. Average RAM utilization of the VMs used in the experiment for the tested workloads.

The *sar* tool ran while *SGeoL Core* and the monitoring API were overloaded by requests, enabling us to observe the VM's resource usage for each applied load. The *sar* tool collected 180 measurements every ten seconds. The collected measurements were used to calculate the average CPU usage and RAM utilization of each VM at the end of the experiment. For statistical significance, we performed 20 runs for each tested workload. We also adopted a significance level at $\alpha = 0.05$ in the statistical analysis of the experimental results.

Figure 8 presents the average response time of the monitoring API (in milliseconds) for the workloads tested in the experiment, namely 100 to 3,000 simultaneous requests. We noticed that the response time tends to increase with the number of requests, which is expected behavior. We also could identify that the increase in the response time is nearly linear upon increasing the number of requests.

Figure 9 shows the average CPU usage (in percentage) of the VMs for the varying workloads of the monitoring API. The average CPU usage of the VM *sgeol* was 87.35% on average (95% CI [87.31, 87.39]) and remained almost constant since the load over the SGeoL platform did not change. The VM *monitoring* demonstrated an increasing CPU usage because the workload of the monitoring API gradually increased from 100 to 3,000 simultaneous requests. The increase in CPU usage was 1% on average for every 100 requests. The VM *client* also demonstrated an increasing CPU usage, likely to run the threads responsible for the simultaneous requests to the monitoring API and SGeoL. However, the increase in CPU usage was smaller than the one observed for VM monitoring.

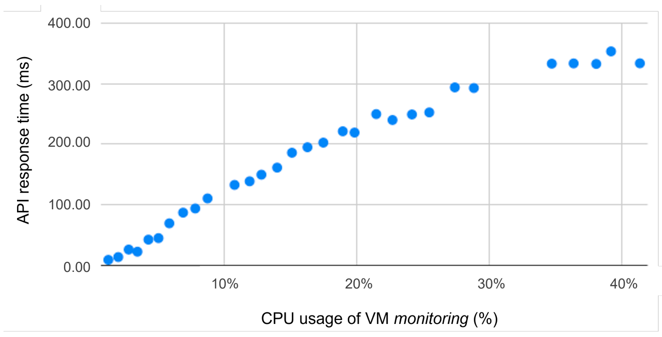

Figure 11. Correlation between the monitoring API response time and CPU usage.

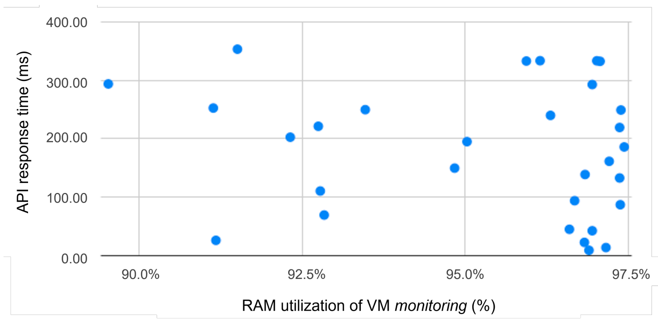

Figure 12. Correlation between the monitoring API response time and RAM utilization.

Figure 10 shows the average RAM utilization (in percentage) of the VMs for the varying workloads of the monitoring API. The average RAM utilization of the VM *sgeol* was 78.10% (95% CI [77.68%, 78.52%]), increasing by 0.13%. For the VM *monitoring*, the average RAM utilization was 95.3% (95% CI [94.44%, 96.16%]), leading us to conclude that the machine needs to be provided with enough resources to handle the workload. The VM *client* had a particular behavior regarding RAM utilization. The RAM utilization increased by 1.75% on average for workloads until 1,600 requests, then it decreased and increased again in the scenarios near 3,000 requests. The observed RAM utilization for the VM *client* presented a larger variation with a mean of 83.21%, 95% CI [80.48%, 85.94%].

The monitoring API response time and the CPU usage of the VM in which it was deployed are the metrics that increased the most during the evaluation. We calculated the Pearson correlation coefficient to quantitatively measure the relationship between the response time and CPU usage. We obtained $r = 0.9784$, indicating a strong positive correlation (almost linear) between the monitoring API response time and CPU usage. This behavior can also be observed in the graph shown in Figure 11.

We also calculated the Pearson correlation coefficient to quantitatively measure the relationship between the monitoring API response time and the CPU utilization of the VM in which it was deployed. We obtained $r = -0.171$, indicating a weak negative correlation between these two variables. This behavior can also be observed in the graph shown in Figure 12.

6 Related Work

Despite several platforms targeting smart cities, a few works in the literature present an architecture with components explicitly responsible for monitoring operations. This section discusses these works and considers some research contributions related to monitoring in Cloud Computing environments, typically used in the context of smart cities.

FIWARE[12] is an open-source platform developed in the European Community with a set of generic, extensible functional components used in several successful cases related to smart city applications. Araujo *et al.* [2019] performed a performance evaluation of FIWARE on a monitoring infrastructure based on Prometheus, an open-source monitoring and alerting solution for cloud-based environments. The evaluation focused on specific FIWARE components: Internet of Things agents, context management, and a non-relational database. The authors considered the agents' CPU and RAM utilization and the database's transfer rate as primary metrics obtained through source code monitoring and instrumentation tools. The measurements are manually analyzed to identify the platform's limits and design a future implementation of an auto-scaling mechanism. In their work, instrumenting the source code for monitoring represents an intrusive approach since the client libraries provided by Prometheus require implementation within the application code. Nevertheless, the performed evaluation could assess performance information and visualize bottlenecks, concluding that the main limitation of the platform was in the database responsible for storing data from different Internet of Things devices.

Del Esposte *et al.* [2019] present an evaluation of the InterSCity platform[13], an open-source microservice-based middleware to support the development of smart city applications. Monitoring in InterSCity targets the microservice containers implementing the platform's functionalities, specifically those related to data collection and management, resource discovery, and request coordination via a gateway. The authors evaluated InterSCity through large-scale simulations using the Kubernetes[14] container orchestration platform. The Kubernetes monitoring engine considers the CPU usage of individual containers to analyze platform behavior and adjust the number of container instances for each service. However, monitoring based only on this metric does not comprehensively demonstrate the behavior of the platform, so other metrics should be used to obtain a broader view of the operation of the components of a smart city platform. The evaluation also showed that InterSCity has an architecture capable of scaling horizontally, but the Kubernetes autoscaling mechanism delays allocating new containers, which can impact the number of failed requests and the response time.

Casalicchio [2019] studied performance measures to monitor application containers (specifically Kubernetes pods), considering their CPU usage and the CPU usage on the machine responsible for their execution. The study was performed on a monitoring infrastructure composed of cAdvisor, Prometheus, and Grafana. Using container and VM mon-

[12]https://www.fiware.org
[13]https://interscity.org/software/interscity-platform/
[14]https://kubernetes.io

Table 1. Comparison of monitoring approaches from related work.

Related work	Location	Levels	Coordination	Intrusiveness
Araujo *et al.* [2019]	Centralized	Multi-level	Uncoordinated	Intrusive
Del Esposte *et al.* [2019]	Centralized	Single-level	Uncoordinated	Intrusive
Casalicchio [2019]	Centralized	Multi-level	Coordinated	Intrusive
Taherizadeh and Stankovski [2019]	Centralized	Multi-level	Coordinated	Intrusive
This work	**Centralized**	**Multi-level**	**Uncoordinated**	**Non-intrusive**

itoring metrics allows for visualizing the platform's behavior at these two levels. However, his work did not consider metrics at the level of the applications that run inside the containers, and it is not possible to visualize the internal behavior of the application's operations.

Taherizadeh and Stankovski [2019] used a multi-level approach for monitoring containers and applications. The container-level monitoring considers CPU, RAM, network, and disk usage as metrics, and the application-level monitoring considers the response time to requests and throughput. In their approach, monitoring agents are deployed into containers together with the application to allow for collecting monitoring metrics. Nonetheless, the approach is intrusive as it requires implementing the monitoring components and deploying them in the application to enable sending measurements to the monitoring agent responsible for data collection.

Table 1 classifies the monitoring approaches used by the works discussed in this section according to the framework presented in Section 2.2. All works used a centralized monitoring approach since the components do not monitor themselves, and they used tools such as Prometheus, Kubernetes, and JCatascopia [Trihinas *et al.*, 2014] to monitor the desired components. The advantage of a centralized monitoring approach is that it allows for easy management of monitoring data. The centralized monitoring approach also fosters historical analysis of a more extensive set of monitoring data, which is more difficult in a distributed monitoring approach because the components monitor themselves and typically do not have enough resources to store a large amount of data.

Most works use the multi-level monitoring approach since they consider more than one level. For example, the assessment of the FIWARE platform carried out by Araujo *et al.* [2019] monitored infrastructure resources (CPU usage and RAM utilization) and the transfer rate of the non-relational database. Casalicchio [2019] monitored the machine and containers through absolute and relative CPU usage, while Taherizadeh and Stankovski [2019] monitored the load balancer to capture the response time of requests made to the application and containers. The InterSCity assessment by Del Esposte *et al.* [2019] monitored only service containers, so only one level is monitored. The advantage of a multi-level monitoring approach is that it enables the user to analyze various aspects of the environment where the smart city platform runs, i.e., it is a more informed, richer monitoring. However, more work is required to monitor more levels since it involves more monitoring tools and, in some cases, significant implementation effort.

The analysis of the FIWARE and InterSCity platforms used an uncoordinated monitoring approach, i.e., the metrics are not associated with obtaining more advanced monitoring. On the other hand, Casalicchio [2019] and Taherizadeh and Stankovski [2019] used a coordinated monitoring approach. The former used relative and absolute usage metrics to get a more realistic view of CPU usage across workloads, whereas the latter used application metrics in conjunction with container metrics for more advanced monitoring. The coordinated monitoring approach is more advantageous in some cases as it would allow scaling the containers through the CPU and checking if the disk still allows scaling more containers. However, the code to perform coordinated monitoring tends to be more complex than that for uncoordinated monitoring.

Finally, all works in this section rely on an intrusive monitoring approach since none mentions decoupling, which is an interesting requirement in developing smart city platforms. Taherizadeh and Stankovski [2019] used the *StatsD* protocol[15] to send the measurements collected at the application level, so it is necessary to import the protocol libraries into the application code to implement such an approach. The other works did not monitor the application level, i.e., they used monitoring tools only to monitor the machine and container levels. The disadvantage of the intrusive monitoring approach is the tight coupling between the application code and the monitoring code and its complexity. In addition, it is advantageous to use the non-intrusive monitoring approach to separate the concerns of application developers from those of developers responsible only for monitoring.

7 Final Remarks

This work presented a non-intrusive continuous monitoring strategy and architecture for smart city platforms. The proposal covers monitoring at multiple levels: host, containers, and platform. The host and container levels are monitored using monitoring agents, which can use consolidated, widely used tools, while the platform level is related to monitoring platform operations. Monitoring at these three levels constitutes one of the main contributions of this work, considering that the proposals found in the literature often consider only the host and container levels and not the internal operation of the platform.

Monitoring platform operations is performed through an AOP-based approach, initially introduced in our previous work [Solino *et al.*, 2022]. Our strategy makes it possible to collect data necessary for monitoring the platform in a non-intrusive way, thus relieving developers from a complex, error-prone task. This is another significant contribution of this work since AOP makes monitoring modular and decou-

[15]https://github.com/b/statsd_spec

pled, and it does not require any modifications in the platform's source code. In our strategy, each critical operation of the platform is monitored through join points. For each join point, there is an advice responsible for collecting the desired monitoring information, such as the time spent executing each operation individually.

The proposed monitoring architecture was implemented and validated in the context of the SGeoL smart city platform, which has been used in several real-world applications [Pereira *et al.*, 2022]. Monitoring made it possible to identify which operations require more processing time to be attended to, allowing the platform developers and administrators to intervene directly in these operations and on the platform's underlying infrastructure to improve its performance. It is worth emphasizing that the strategy was validated in the context of SGeoL, but it could also be implemented for other smart city platforms, taking advantage of the facilities of using AOP to monitor the computational infrastructure and platform operations.

This paper also reported the results of computational experiments to evaluate the performance of the continuous monitoring API in scenarios with different workloads. The experiments considered response time to requests, CPU usage, and RAM utilization as metrics. The results showed an evident increase in response time with the number of simultaneous requests. Furthermore, we observed a significant correlation between the response time of the monitoring API and the CPU usage of the machine in which it is deployed.

The monitoring architecture proposed in this work can be used as a basis for an auto-scaling mechanism of the infrastructure resources used by the platform and the monitored components. Auto-scaling refers to reconfiguring the allocation of hardware and software resources at runtime to cope with time-varying environmental conditions and performance requirements [Chen *et al.*, 2018]. An auto-scaling mechanism automatically and timely adds or removes cloud resources on an on-demand basis in response to dynamic workload variations, allowing to overcome the scalability limitations due to the increase in the number of users, devices, services, and data.

Regarding ongoing and future work, we are integrating our monitoring strategy into an approach based on the MAPE-K control loop [Kephart and Chess, 2003] to autonomously provide elasticity to smart city platforms. MAPE-K is a reference model for the development of autonomic systems in different contexts, which includes stages of monitoring, analysis, planning, and execution of actions, which allow a system to manage itself at runtime. With this, it will be possible to dynamically adjust the underlying computational infrastructure that supports the deployment and execution of the platform according to the workload to maintain the desired quality of service levels.

Declarations

Funding

This work was partially supported by the Smart Metropolis Lab (https://smlab.imd.ufrn.br/).

Authors' Contributions

J. V. Lopes and A. Solino contributed with Software, Formal analysis, Investigation, and Writing - Original Draft. E. Cavalcante contributed with Conceptualization, Methodology, Writing - Original Draft, and Supervision. T. Batista contributed with Conceptualization, Writing - Review & Editing, and Project administration. J. Pereira and A. Rocha Neto contributed with Writing - Review & Editing. All authors read and approved the final manuscript.

Competing interests

The authors declare that they have no competing interests.

Availability of data and materials

Data can be made available upon request.

References

Abranches, M. C. and Solis, P. (2016). A mechanism of auto elasticity based on response times for cloud computer viroments and autossimilar workload. In *Proceedings of the XLII Latin American Computing Conference*, USA. IEEE. DOI: 10.1109/CLEI.2016.7833403.

Araujo, V., Mitra, K., Saguna, S., and Åohlund, C. (2019). Performance evaluation of FIWARE: A cloud-based IoT platform for smart cities. *Journal of Parallel and Distributed Computing*, 132:250–261. DOI: 10.1016/j.jpdc.2018.12.010.

Bagnasco, S., Berzano, D., Guarise, A., Lusso, S., Masera, M., and Vallero, S. (2015). Monitoring of IaaS and scientific applications on the cloud using the Elasticsearch ecosystem. *Journal of Physics: Conference Series*, 608. DOI: 10.1088/1742-6596/608/1/012016.

Basili, V. R., Caldiera, G., and Rombach, H. D. (1994). The Goal Question Metric Approach. In Marciniak, J. J., editor, *Encyclopedia of Software Engineering*, volume 2. John Wiley & Sons. DOI: 10.1002/0471028959.sof142.

Bhardwaj, A. and Krishna, C. R. (2021). Virtualization in Cloud Computing: Moving from hypervisor to containerization – a survey. *Arabian Journal for Science and Engineering*, 46:8585–8601. DOI: 10.1007/s13369-021-05553-3.

Casalicchio, E. (2019). A study on performance measures for auto-scaling CPU-intensive containerized applications. *Cluster Computing*, 22(3):995–1006. DOI: 10.1007/s10586-018-02890-1.

Cavalcante, E., Cacho, N., Lopes, F., and Batista, T. (2017). Challenges to the development of smart city systems:a system-of-systems view. In *Proceedings of the XXXI Brazilian Symposium on Software Engineering*, pages 244–249, USA. ACM. DOI: 10.1145/3131151.3131189.

Chen, T., Bahsoon, R., and Yao, X. (2018). A survey and taxonomy of self-aware and self-adaptive cloud autoscaling systems. *ACM Computing Surveys*, 51(3). DOI: 10.1145/3190507.

Coutinho, E. F., de Carvalho Sousa, F. R., Rego, P. A. L., Gomes, D. G., and de Souza, J. N. (2015). Elasticity in

Cloud Computing: A survey. *Annals of Telecommunications*, 70:289–309. DOI: 10.1007/s12243-014-0450-7.

Del Esposte, A. M., Santana, E. F. Z., Kanashiro, L., Costa, F. M., Braghetto, K. R., Lago, N., and Kon, F. (2019). Design and evaluation of a scalable smart city software platform with large-scale simulations. *Future Generation Computer Systems*, 93:427–441. DOI: 10.1016/j.future.2018.10.026.

ISO (2019). Iso 37122:2019 - sustainable cities and communities – indicators for smart cities. Available at: https://www.iso.org/standard/69050.html.

Kampars, J. and Pinka, K. (2017). Auto-scaling and adjustment platform for cloud-based systems. *Environment. Technologies. Resources.*, 2:52–57. DOI: 10.17770/etr2017vol2.2591.

Kephart, J. O. and Chess, D. M. (2003). The vision of autonomic computing. *Computer*, 36(1):41–50. DOI: 10.1109/mc.2003.1160055.

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G. (2001). An overview of AspectJ. In Knudsen, J. L., editor, *ECOOP 2001 – Object-Oriented Programming*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–354. Springer-Verlag Berlin Heidelberg, Germany. DOI: 10.1007/3-540-45337-$7_1$8.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. (1997). Aspect-oriented programming. In Akşit, M. and Matsuoka, S., editors, *ECOOP'97 – Object Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer Berlin Heidelberg, Germany. DOI: 10.1007/bfb0053381.

Kiczales, G. and Mezini, M. (2005). Aspect-oriented programming and modular reasoning. In *Proceedings of the 27th International Conference on Software Engineering*, pages 49–58, USA. ACM. DOI: 10.1145/1062455.1062482.

Ma, K., Sun, R., and Abraham, A. (2013). Toward a module-centralized and aspect-oriented monitoring framework in clouds. *Journal of Universal Computer Science*, 19(15):2241–2265. DOI: 10.3217/jucs-019-15-2241.

Matsumoto, R., Kondo, U., and Kuribayashi, K. (2019). Fast-Container: A homeostatic system architecture high-speed adapting execution environment changes. In *Proceedings of the IEEE 43rd Annual Computer Software and Applications Conference*, pages 270–275, USA. IEEE. DOI: 10.1109/COMPSAC.2019.00047.

Narayana, S., Mainak, S., and Paul, A. S. (2020). Application deployment using containers with auto-scaling for microservices in cloud environment. *Journal of Network and Computer Applications*, 160. DOI: 10.1016/j.jnca.2020.102629.

Pereira, J., Batista, T., Cavalcante, E., Souza, A., Lopes, F., and Cacho, N. (2022). A platform for integrating heterogeneous data and developing smart city applications. *Future Generation Computer Systems*, 128:552–566. DOI: 10.1016/j.future.2021.10.030.

Santana, E. F. Z., Chaves, A. P., Gerosa, M. A., Kon, F., and Milojičić, D. S. (2017). Software platforms for smart cities: Concepts, requirements, challenges, and a unified reference architecture. *ACM Computing Surveys*, 50(6). DOI: 10.1145/3124391.

Solino, A., Lopes, J. V., Batista, T., Cavalcante, E., Pereira, J., and Rocha Neto, A. (2022). Uma estratégia orientada a aspectos para monitoramento de plataformas para cidades inteligentes. In *Anais do XLIX Seminário Integrado de Software e Hardware*, pages 164–175, Brazil. SBC. DOI: 10.5753/semish.2022.223200.

Taherizadeh, S. and Stankovski, V. (2019). Dynamic multi-level auto-scaling rules for containerized applications. *The Computer Journal*, 62(2):174–197. DOI: 10.1093/comjnl/bxy043.

Trihinas, D., Pallis, G., and Dikaiakos, M. D. (2014). JCatascopia: Monitoring elastically adaptive applications in the cloud. In *Proceedings of the 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 226–235, USA. IEEE. DOI: 10.1109/ccgrid.2014.41.