Journal of Grid Computing (2023) 21:44
https://doi.org/10.1007/s10723-023-09675-w

RESEARCH

Enhancement of Cloud-native applications with Autonomic Features

Joanna Kosińska · Krzysztof Zieliński

Received: 22 February 2023 / Accepted: 31 May 2023 / Published online: 15 July 2023

Abstract The Autonomic Computing paradigm reduces complexity in installing, configuring, optimizing, and maintaining heterogeneous systems. Despite first discussing it a long ago, it is still a top research challenge, especially in the context of other technologies. It is necessary to provide autonomic features to the Cloud-native execution environment to meet the rapidly changing demands without human support and continuous improvement of their capabilities. The present work attempts to answer how to explore autonomic features in Cloud-native environments. As a solution, we propose using the AMoCNA framework. It is rooted in Autonomic Computing. The success factors for the AMoCNA implementation are its execution controllers. They drive the management actions proceeding in a Cloud-native execution environment. A similar concept already exists in Kubernetes, so we compare both execution mechanisms. This research presents guidelines for including autonomic features in Cloud-native environments. The integration of Cloud-native Applications with AMoCNA leads to facilitating autonomic management. To show the potential of our concept, we evaluated it. The developed executor performs cluster autoscaling and ensures autonomic management in the infrastructure layer. The experiment also proved the importance of observations. The knowledge gained in this process is a good authority of information about past and current state of Cloud-native Applications. Combining this knowledge with defined executors provides an effective means of achieving the autonomic nature of Cloud-native applications.

J. Kosińska (✉)
e-mail: kosinska@agh.edu.pl
AGH University of Science and Technology, Faculty of Computer Science, Electronics and Telecommunications, Institute of Computer Science, al. A. Mickiewicza 30, 30-059 Krakow, Poland

1 Introduction

Autonomic Computing (AC) paradigm [21,30,48] introduced by IBM in 2001, is a response to the need to reduce complexity in the installation, configuration, optimization, and maintenance of heterogeneous systems. Similarly, as the human brain is relieved of vital functions, the computing system unconsciously deals with management tasks. AC vision focuses on self-managed systems.

Autonomic computing is not a new paradigm, but it is not fully exploited in complex computing systems [2] yet. The reason is the broad spectrum of possible solutions that are difficult to evaluate and highly dependent on specific use cases. These problems concern several theoretical and technical aspects. Theoretical aspects include the choice of analysis algorithms and their implementation. The technical difficulties are mainly related to the structure of the closed-loop feedback control [20] that is the core of every AC system. It is often challenging to integrate the loop into the base system. However, a large body of research addressing these issues [6,28,33,45] exists. The most relevant

in the context of Cloud-native applications (CNApps) and their execution infrastructure relates to the Kubernetes (k8s) ecosystem [11,24]. Its concept is based on a declarative infrastructure driven by a reconciliation loop [27]. The concept of a reconciliation loop ensures that the current system's state matches the desired. The idea is similar to a MAPE-K loop [6] present in AC. The described problems motivated our work.

The main contribution of this paper is the set of guidelines for including autonomic features in Cloud-native environments. We present the concept of the AC paradigm in the context of Cloud-native. To fulfill the contribution, we introduce the notion of an Execution Controller that triggers the management actions. We verified the proposed process using the AMoCNA prototype. The name AMoCNA comes from the first letters of **A**utonomic **M**anagement of **C**loud-**n**ative **A**pplications. It is rooted in AC. We developed this system to prove the correctness of the proposed concepts.

At first glance, AMoCNA reduces the complexity of managing Cloud-native applications. AMoCNA addresses the notion of autonomic management. Autonomic management is defined as performing administrative tasks such as installation, configuration, optimization, and maintenance in heterogeneous computing systems with or without human intervention.

This paper also compares our proposition with solutions already used in Kubernetes, such as controllers and operators. They constitute the indispensable elements of a reconciliation loop. We contrast them with the AMoCNA controllers elaborated on in our work. AMoCNA controllers combined with k8s supply AC capabilities. It is noticeable that both solutions have complementary functionalities and that Cloud-native environments can benefit from their co-existence.

In summary, this research basis on our previous work [34,35]. In paper [34], we specify an abstract view of a Cloud-native application execution environment. Then we refer to AC and characterize the realization of a Cloud-native autonomic element. In that paper, we introduce the novel concept of a MRE-K loop (It comes from the first letters of the loop, which are Monitor, Rule engine, Execute, and Knowledge). The MRE-K loop extends the MAPE-K loop [6] with appropriate adjustments to the Cloud-native context. We also introduce the AMoCNA model with its microservices architecture. On the other hand, in the paper [35], we evaluate the concepts developed. The main additional contributions, compared to our previous papers, are as follows:

1. Comparison of the controller characteristics offered by the proposed AMoCNA framework and the current orchestrators' implementations (using the example of Kubernetes [39]).
2. The set of guidelines for including autonomic features in Cloud-native environments.
3. The concept of an execution controller. This concept results from the above guidelines.

The structure of this paper is as follows. First, we present the aim and contribution of the research. Section 2 outlines standards and technologies related to Cloud-native application management. In the next section, we describe the Kubernetes reconciliation loop and compare its controllers with the AMoCNA controllers. In this section, we present our first contribution. Based on the insights of Sections 2 and 3, the following section proposes the AC part of the framework for the autonomic management of CNApps. Section 4 presents data and invocation flows between the distinguished components. Then, in Section 5, we identify the steps necessary to include autonomic features in Cloud-native environments. The steps form the guidelines that are our main contribution. Based on the AMoCNA architecture, Section 6 describes the controllers of the autonomic element microservice with a deep analysis of an execution controller. This controller is our third contribution to Cloud-native Computing. Its usefulness is evaluated in Section 7. Finally, we summarize the paper. We also suggest directions for further development and research.

2 Related work

For decades, system and software components have evolved to address the increasing complexity of system control, resource sharing, and operational management [20]. Development of elements responsible for self-management adds the following autonomous properties to the system [48]: self-configuration, self-healing, self-optimization, and self-protection. Such system architectures solve the overall complexity of resource management. [22] presents a comprehensive study with a taxonomy of this domain. The International Workshop on Intelligent Techniques and Architectures for Autonomic Clouds discusses topics fun-

damental to concepts, architectures, and applications. ITAAC 2011 [5] selected four articles for the Special Issue, one of which [31] seems particularly interesting in the context of resource management. The authors propose an architecture for integrated intelligence in urban management and indirectly suggest that integrated environmental monitoring is a must [18,53,55]. To this end, researchers have developed an IEM service that allows data collection, storage, processing, visualization, and dissemination. Only systems that ensure these features can be autonomic.

Adaptive and autonomic management of computing resources is sometimes incorrectly used as synonyms. Adaptive systems involve processes that change their behavior based on the execution context. Autonomic systems cover systems with a broad knowledge of their execution environment and operate beyond their boundaries. The knowledge collected enables such systems to make involuntary decisions and proceed without human intervention. These systems are often called 3A (Automatic, Adaptive, and Aware) systems [47,51].

[36] address adaptability by focusing on multi-cloud provisioning. The presented research has developed the CAMEL language, extending it to support the adaptive provisioning of multi-cloud business processes (BPs). The proposed environment includes, among others, cross-level monitoring and adaptation of BPs. As a digression, something worth noting is that AMoCNA has no additional layer associated with the representation of knowledge, as in the cited work. The knowledge representation would, among others, make Cloud-native applications understandable for third-party systems and improve interoperability between microservices. However, most of the research in Semantic Computing [25] focuses only on Cloud Computing (CC). The Cloud-native has risen on the top of the CC. It is working at the microservices level and among Cloud-native apps. Most of the concepts introduced are not yet present in well-known modeling languages (CloudML [23] or CAMEL [3], etc.). Let us return to the main topic after that brief digression. [44], in turn, presents the adaptation at the cloud environment level, strictly managing the rational use of computing resources. Management tasks are often the responsibility of a Resource Management System (RMS). RMS acts as a middleware between resources and application requirements. Requirements are negotiated in a contract (SLA) and are encapsulated in QoS metrics. The autonomic behavior of the system requires the specification of the goal resulting from the adaptation or realization of autonomic processes. The validity of objectives implies meeting the goals in adapting or evolving systems [44]. In the research [44], authors indicate that achieving cloud architecture continuity requires systems to change their architecture and maintain the validity of the goals that determine the architecture. The authors propose several models for adaptation and evolution in research and industry consulting projects.

[10] adopted the definition of awareness (the third capability of a 3A system) as a property of the system that demonstrates cognition and learning. It proposes to find inspiration in the Cognitive Immune Network [15]. The mentioned RMS seeks to maximize resource metrics and, at the same time, achieve the negotiated SLA. The RMS uses monitoring capabilities to ensure this feature. In modern systems, monitoring is mandatory [55]. The knowledge gained provides insight into the environment and allows them to predict their future behavior.

Context-aware systems store and process a lot of data. Knowledge helps systems become aware of situations, recognize states, and react to changes. [54] present a high-level model of structured knowledge and a formal model of awareness in autonomic service-component ensembles (Autonomic Service-Component ENSembles (ASCENS)). In such systems, initial knowledge that addresses self-awareness and context-awareness is crucial. The authors propose an algorithm that initializes the knowledge of the system. The algorithm uses the ASCENS ontology. Similar research in the Cloud-native area would significantly improve awareness.

All these systems have the characteristics of an autonomic system. They all acknowledge that AC can be applied in any domain. However, this research recommends using AMoCNA in Cloud-native environments to achieve autonomic characteristics. The AMoCNA's policy approach uses DSL and brings a significant benefit over the mentioned research. Its potential lies in the possibility of declaring many management policies at once. They can be related to different aspects and can be composed at runtime. With AMoCNA, it becomes possible to design any management action, even actions that enable self-configuration, self-healing, self-optimization, and self-protection of the CNApps.

In addition to AC, this research topic operates within the boundaries of Cloud-native. The concept of loosely coupled microservices [42] is one of the building blocks of Cloud-native. These microservices are thousands. Therefore, manual management of such tasks is not possible. An orchestrator is a tool that addresses the requirement for automatic management of services. It is a process of automating, coordinating, and managing specific IT tasks. The three most popular open-source tools for container orchestration are: Docker Swarm [16], Kubernetes [39], and Apache Mesos [41]. Out of the box, they manage Docker [16] containers. However, there are also some successful adoptions to orchestrate other container implementations. For example, in HPC, Singularity [49] has become the basic container runtime. [56] introduces a Torque-Operator that serves as a bridge between the HPC workload manager (TORQUE) and a container orchestrator (Kubernetes).

For the record [34], the architecture of AMoCNA is based on five layers (Fig. 1) that cooperate in a loop. We propose to achieve autonomy features through the support of the rule engine. The layers with flow depicted as blue arrows control the CNApp. The lowest layer is the Instrumentation Layer. This layer exposes the data of a Cloud-native application to the Observation Layer, which collects information related to observability. After processing the data, the Observation Layer passes them on to the Inference Layer. TheInference Layer, among others, correlates data from different sources, transforms the observation data into a format accepted by the rule engine, and interprets, analyzes, and infers from the data. This layer helps to reconfigure CNApp following the externally declared policies (depicted as a red arrow). High-level directives, called policies, usually include the parameters of an SLA contract [35]. AMoCNA uses a policy management approach in Cloud-native execution environments to significantly reduce the burden of defining enforced actions. The Inference Layer produces management actions and passes them to the Management Layer, and through the Instrumentation Layer, invokes proper actions in the Cloud-native Execution Environment. The procedure is closed, and another loop execution collects feedback from a CNApp. The Instrumentation Layer gathers feedback expressed as observability-related information. We give an example to illustrate the AMoCNA control loop. In the example presented, the end user declares a management policy (red flow in Fig. 1) that influences CNApp (Policy 1). The policy begins its flow in the Control Layer, which checks whether it obeys the SLA contract. The next layer is the Inference Layer. It is based on the observation parameters and changes the policy into executable management actions. These actions are directed to the proper components by the Management Layer. The Instrumentation Layer, through effectors, invokes the CNApp's

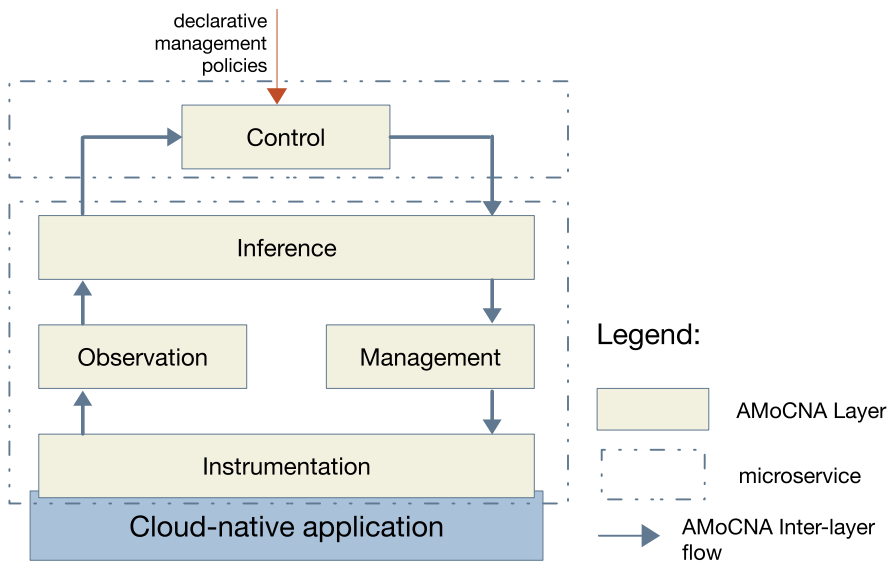

Fig. 1 Simplified microservice architecture of AMoCNA

Policy 1 Check whether it is necessary to increase CNApp's CPU request

Require: RT – [milliseconds] (response time) the duration of execution of a CNApp.
SLA_RT – [milliseconds] (acceptable response time) duration of execution of a CNApp agreed on SLA.
Ensure: Increase of CNApp's CPU request.
if $RT > SLA\_RT$ **then**

```
increase CPU Request
```
end if

methods that reflect the management actions. Succeeding flow in the loop collects updated observations originating from the execution of the declared policy. Our paper [34] details the AMoCNA architecture and its features.

The AMoCNA framework is based on an architectural model of self-management and clearly distinguishes the constructs of autonomic elements.In paper [35], we evaluate the usefulness of AMoCNA and the improvement of Cloud-native environments. Their results assess this framework positively.

3 Reconciliation loop basics

Cloud-native leverages open-source software stack and deploys new applications as containers. Then these containers are dynamically orchestrated to optimize resource utilization [13]. CNCF provided a trail map that is an overview of moving toward Cloud-native architecture [14]. The main identified and obligatory steps in the Cloud-native context are containerization [8], CI/CD philosophy [32] and orchestration [46]. Although orchestration is the last identified step, it is not the least one. An orchestrator is a workflow management solution that automates resource creation, monitoring, and deployment in the execution environment. Most cloud platforms use Kubernetes to orchestrate resources [52]. The Kube Controller Manager [39] manages the embodied core control loops [9]. Kubernetes control loops observe the shared state of the cluster. On the basis of the observations, they adjust it according to the desired state. Examples of Kubernetes controllers are the replication, endpoints, namespace, serviceaccounts controller, etc. AMoCNA framework also distinguishes controllers. Section 6 describes them. Their underlying objective is similar. The controllers in both solutions try to keep the current state in sync with the desired state. However, there are some subtle differences. Some are listed in Table 1 and are further described in more detail.

The distinguished AMoCNA controllers do not follow the Kubernetes directives. They serve AMoCNA's goals but use the same premises to enforce a closed feedback control loop [29]. The fundamental difference between AMoCNA and Kubernetes controllers is their objective. AMoCNA operates according to declared high-level policies. Policies reflect the SLA contract and enforce actions that help to obey established rules. Kubernetes controllers proceed considering the state of the system. They aim to synchronize the current and desired state.

A detailed description of the AMoCNA framework and MRE-K loop concepts is given in our previous work [34]. However, for the necessary background, we present some basic information in this section. At the heart of the AMoCNA framework is a Cloud-native MRE-K loop. The framework is based on the Autonomic Elements' [26] constructs adopted from AC and its MAPE-K loop (see Fig. 2). Using Autonomic Elements in the context of Cloud-native requires that the elements follow the Cloud-native philosophy presented in the CNCF trail map. Cloud-native autonomic elements expose the features and capabilities of managed components. These realize sensors and effectors. Fig. 2 depicts the mapping between an autonomic element and a Cloud-native autonomic element, particularly our concept of using a rule engine as a decision module. The figure conceptualizes the internal structure of the element. The Analyze and Plan parts of the loop are combined in a rule engine, causing the substitution of the letters AP from MAPE-K into R. The loop name changes from MAPE-K to MRE-K. The letter K stands for Knowledge.

Similar features of an autonomic system have Kubernetes based on the concept of a reconciliation loop. The loop encompasses diverse components that are loosely coupled and work in a separation, driving the cluster to the desired state.

Following [4], we define the policy as a set of restrictions imposed on all possible forms of system behavior so that the result is a subset of acceptable system behaviors. In AMoCNA, policies are rules (if-then constructs) processed by the MRE-K loop via a rule engine. The rule engine enables the declaration of many policies at once, and they can regard various aspects. Policies can be composed on the fly at runtime. With

Table 1 Comparison of the characteristics of the controllers offered by the proposed AMoCNA framework and the Kubernetes [39] controllers

Feature	AMoCNA Controllers	Kubernetes Controllers
Objective	Agreement between the current state and declared policy	Synchronize current state with the desired state
Control style	Cloud-native MRE-K loop concept	Plenty components work together to implement reconciliation behavior
Declarative versus Imperative Programming	Declarative	Declarative
Area of activities	All layers of CNApp stack	Containerization Layer
Driven by	Declarative management policies	API call commands
Architecture	Not specified	Well-known
Extending the default architecture	Not applicable	Through operator pattern
Linkage	Yes	No

AMoCNA, it is possible to declare any management action.

To achieve their functionality, all controllers use declarative programming. In AMoCNA, Cloud-native Autonomic Elements (depicted in Fig. 2) accomplish the declarative features, strictly speaking, the Cloud-native MRE-K loop structures and declaration capabilities of the contained rule engine. The core of Kubernetes makes up the API server. Communication with Kubernetes components and the invocation of nearly all operations accomplish the exposed HTTP API, also called a declarative API. All methods are accessible through REST calls or CLI tools for administrators.

We should stress that because of characteristics of Cloud-native, AMoCNA operates in the context of the orchestrated environment. The coexistence of AMoCNA and an orchestrator benefits the CNApps execution environment. In addition to orchestrator

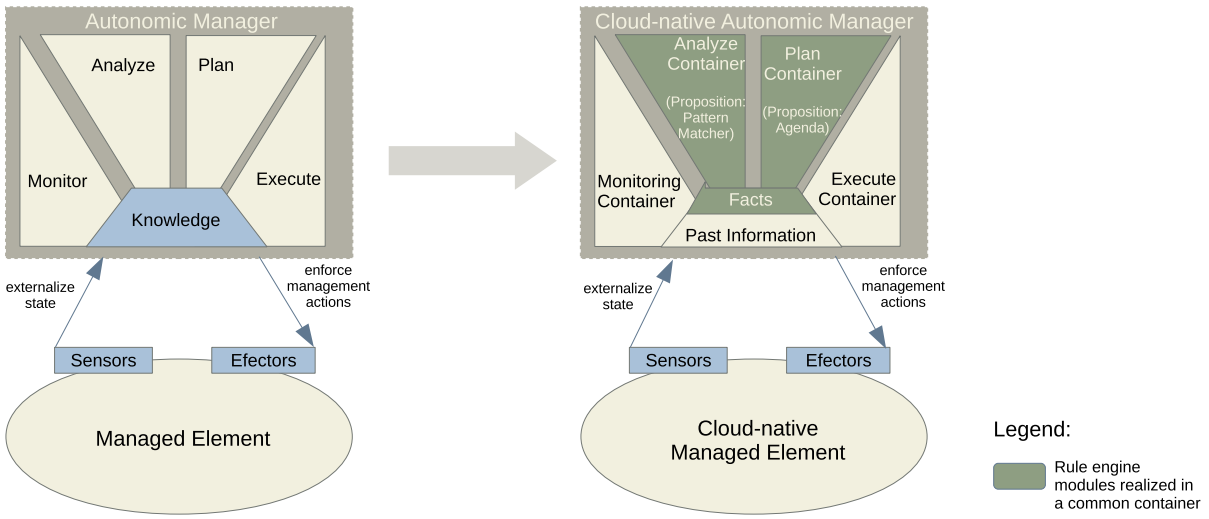

Fig. 2 Mapping the Autonomic Element from AC into the Cloud-native Autonomic Element

capabilities, AMoCNA also significantly enhances the execution environment. Figure 3 shows its simplified view. AMoCNA controllers operate in three different layers of the execution environment (depicted in Fig. 3). Kubernetes controllers know only the data related to the Containerization Layer, and the executed actions also concern that layer. The view of the entire CNApp stack enables AMoCNA to enforce any management action against the CNApp execution environment.

The examples in the distinguished layers include, among others:

- Infrastructure Layer – `create_snapshot`, `upgrade_OS`, `allocate_IPs`
- Containerization Layer – `join_cluster`, `list_images`, `create_network`
- Application Layer – `update_microservice`, `show_logs`

CNApps adhere to policies regarding all layers of the execution stack. In the AMoCNA framework, a declarative management policy process governs the management of the environment. Declarative management policies emphasize the end-user vision of the system expressed in a non-sophisticated way. These policies usually constitute high-level demands that seamlessly translate into low-level executable actions that, in turn, are mapped to the corresponding effectors. In addition, Cloud-native applications can be reconfigured at runtime according to the declared management policies. The published reconfigurations execute while the CNApp is instrumented, closing the control loop. On the other hand, Kubernetes controllers that reside in the control plane function in an event-driven way. But these events often occur as a response to the administrator's directives. These directives pertain only to components of the containerization layer. Kubernetes documentation specifies only policies for resources (limit ranges, resource quotas, process ID limits, reservations, etc.) and networks. In addition, only sufficiently skilled humans can produce them.

The architecture of AMoCNA controllers is very general, allowing the building of each controller according to its rules (Section 6). Stress is placed on its functionality, not on its internal structure. Kubernetes controllers are contrary to AMoCNA controllers. They follow a proper blueprint. Therefore, it is possible to develop a custom controller combined with a custom resource. A new resource must follow the Kubernetes Operator pattern [39] to extend the default architecture of the Kubernetes cluster with the additional one. Kubernetes operators introduce automation into cluster environments. They supervise repeatable tasks and replace human operators.

AMoCNA is based on the concept of cooperating autonomic elements that form a hierarchical structure and are linked. This paradigm is based on AC auto-

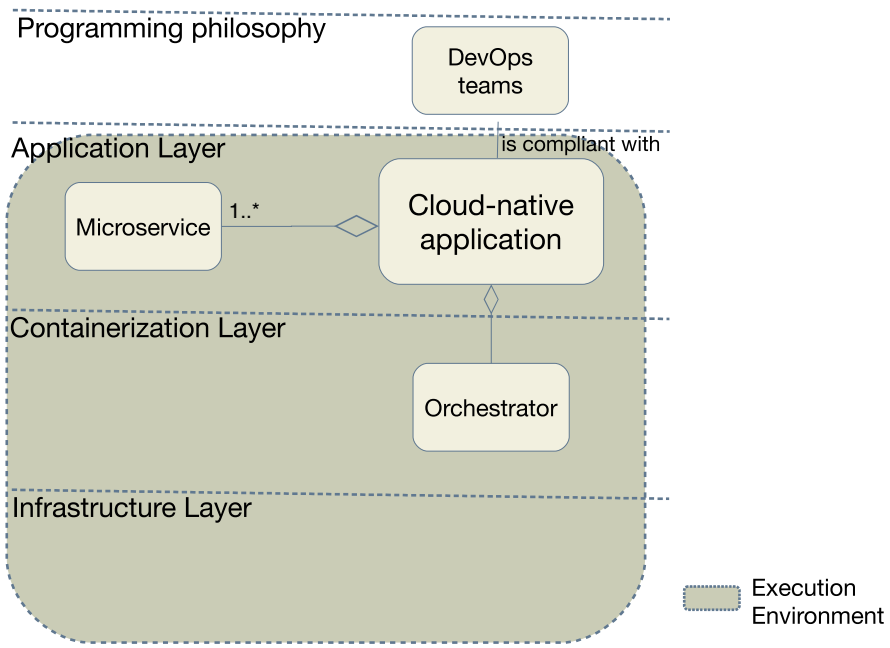

Fig. 3 The distinguished layers of a Cloud-native Execution Environment

nomic elements adjusted to the Cloud-native context. Our proposition presents Section 4. Cloud-native autonomic elements realize the MAPE-K loop (the MRE-K loop in this research). Because of the connection with each other, autonomic elements have access to their and other autonomic elements' knowledge. It is a significant advantage compared to Kubernetes, where the controllers do not share their knowledge. Access to comprehensive knowledge allows for broad control over the entire Cloud-native application execution environment. The benefits of such holistic knowledge are presented in the paper [35] where, in carried experiments, we show the management of the entire execution environment.

AMoCNA control with feedback and the Kubernetes reconciliation loop have more similar features. For example, both solutions provide a similar control style based on the characteristics of the control loop. They also share differences. For example, the AMoCNA loop depends on feedback and observability, whereas only human directives trigger Kubernetes loops. AMoCNA control activity is far more general than only enforcement of the desired state. Table 1 pointed out only some characteristics of the controllers available in both solutions. However, it is possible to point out their pros and cons. This knowledge makes new areas for AMoCNA further improvements (e.g., research toward standardizing AMoCNA controllers to develop K8s AC operators, which would enable the treatment of AMoCNA as a pattern). This table and the comparison [35] of the AMoCNA framework with Kubernetes allows to state that the proposed approach to include autonomic features in a Cloud-native application improves the quality of Cloud-native application execution environment that existing orchestrators offer.

4 Autonomic Computing Paradigm in the context of Cloud-native

An autonomic element [51], a concept from AC, is a fundamental building block of any autonomic system. It aims at hiding the complexity of overall management of the system, particularly a Cloud-native system.

In the paper [35], we present our vision of a high-level view of the design of the AMoCNA management process. We divide the system into three logical parts, as shown in Fig. 4. These are (from the bottom up) Execution Environment, Autonomic Computing and the Management Policies part. This research focuses on the AC part. It is a middle part of the management process, and it is further explained. This part consists

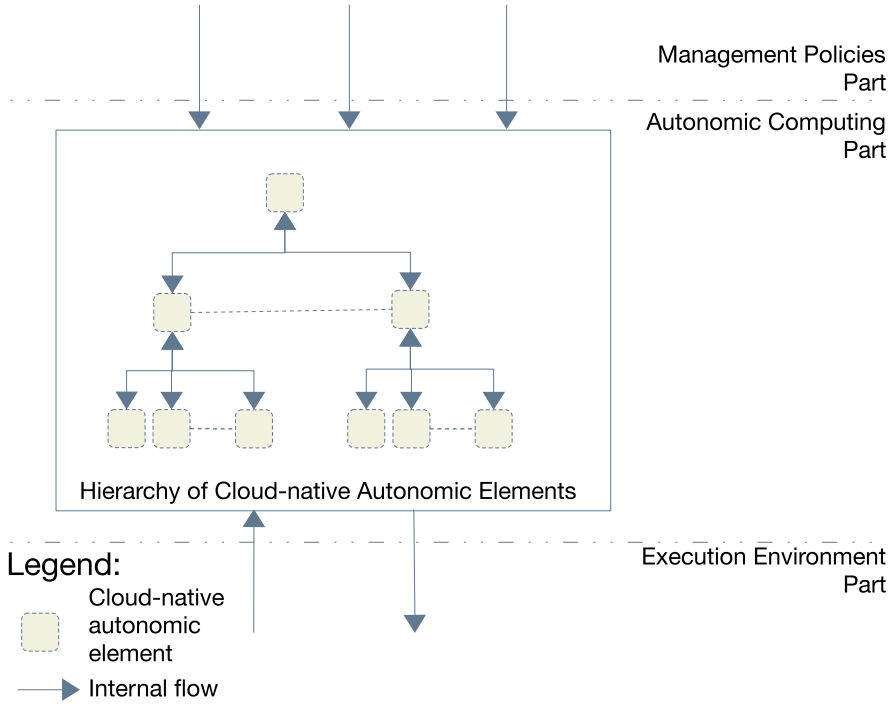

Fig. 4 Autonomic Computing (AC) part of the AMoCNA management process.

of a network of connected autonomic elements. AC states that an Autonomic Manager manages an Autonomic Element. The AC components are organized in a hierarchical fashion [30], which is shown in Fig. 4 as a hierarchy of Cloud-native Autonomic Elements. Consequently, the Cloud-native Autonomic Managers are connected and interoperate with other Cloud-native Autonomic Managers. Hence Fig. 4 also depicts a hierarchy of Cloud-native Autonomic Managers. The top of the tree structure constitutes a Cloud-native Autonomic Supervisor that is in fact AMoCNA's framework supervisor (see Fig. 5).

The number of Cloud-native autonomic elements depends on required accuracy and performance, and it may differ among particular components of the Cloud-native execution environment. Usually, AMoCNA's autonomic elements have mapped to the components of a Cloud-native execution environment in a ratio of 1:1. In particular, each layer contains exactly one Cloud-native autonomic element. Even the whole Cloud-native execution environment can map directly to one Cloud-native autonomic element.

Figure 5 shows the data flows of the AMoCNA autonomic elements. For clarity, the Knowledge part is not included in the figure. Such a simplification was possible because this paper does not elaborate on knowledge management. The hierarchical structure of elements of particular layers of Cloud-native execution environment is the foundation of the flows. In general, the communication is based on a notion of communication between Cloud-native Autonomic Managers. Figure 5 highlights two blue autonomic elements (located in different layers) and presents the data flow between them in detail. As can be seen, the information flow is between particular components of a conceptualized MRE-K loop. That is, the flow between the monitoring components and the rules engine components is directed from elements of lower-layer to upper-layer and in the opposite direction between the Execute components[1].

An example of a use case helps explain this procedure more precisely (Fig. 6). Let the autonomic element in the Containerization Layer represent an Orchestrator. In the Infrastructure Layer, the autonomic element

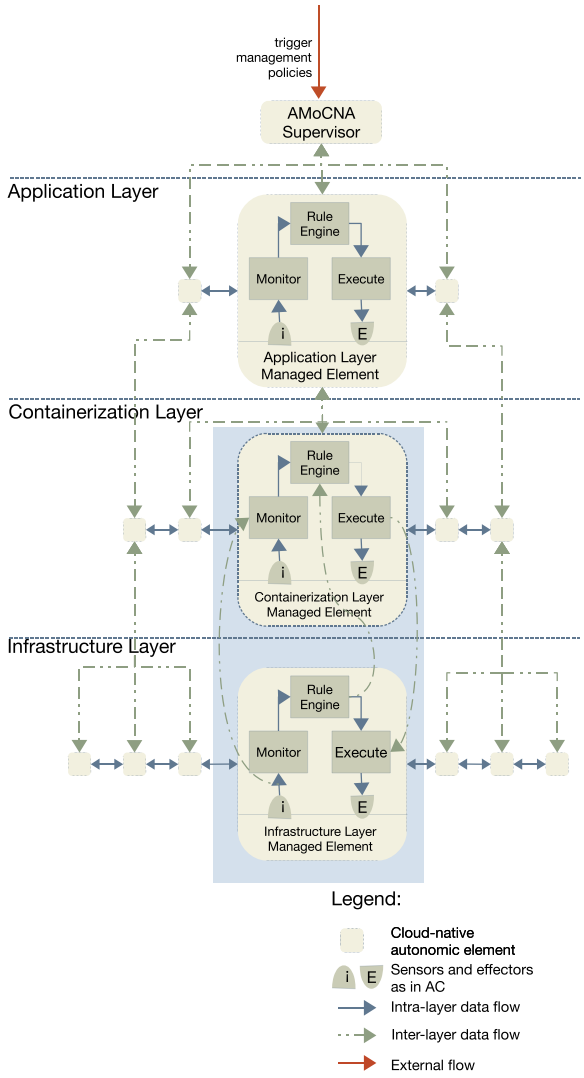

Fig. 5 AMoCNA autonomic elements' data flows

represents a corresponding (that composes the cluster) server (strictly a Computing Resource component from a Cloud-native execution environment), for example, a VM. The VMs sensors send monitoring information (health, number_of_cores, memory_used, etc.) to its Monitor part and the container's component Monitor part. The container starts on that particular VM, hence the need for its observation. In Fig. 6, the monitoring data flow mentioned above is marked as ①. The flow ② symbolizes the VM's Rule Engine output. It is an input to Orchestrator's Rule Engine[2].

[1] communication between Cloud-native autonomic elements residing in the same layer is realized similarly between the identical internal components. The difference is that the data flows are bidirectional

[2] Rule Engines of diverse components' types, have different declared rules.

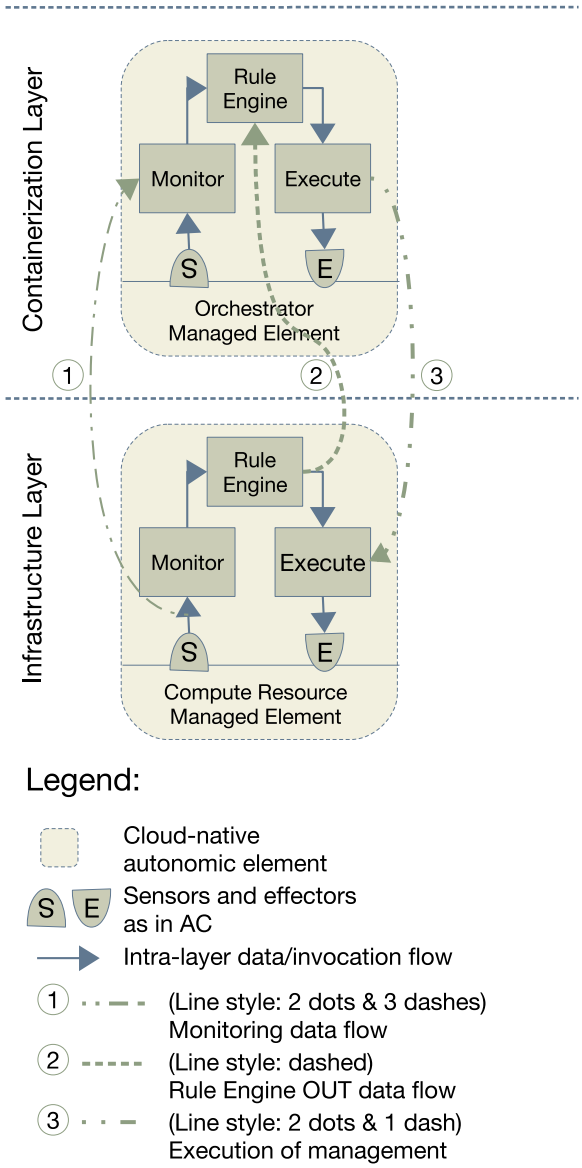

Fig. 6 Detailed description of AMoCNA flows

Policy 2 represents a rule that checks the health of a VM. It shows the appropriate steps to take if the VM detects a failure. First, the actions taken on lines 2 and 3 directly move to VM's Execute part. Line 4 propagates the action to the orchestrator of the present autonomic element, and the failed node is designated not to schedule any workload. The last ③ flow symbolizes the management actions flow between Execute parts of Cloud-native autonomic elements. The flow is top-down, meaning that the components of

Policy 2 A rule checking the VM's health. The rule is fired in VM's Rule Engine

```
1. if ¬ vm.isHealth() then
2.     vm.migrate()
3.     vm.recover()
4.     orchestrator.setDrain(node)
5. end if
```

the upper layers transfer the management actions to the lower layers. As depicted in Fig. 6, management actions from the orchestrator's autonomic element are refined in its Execute part and, if necessary, propagated to the VM's Execute part. For example, during the runtime of a microservice, one of its containers consumes more RAM than is available. Hence, Execute part from an Orchestrator invokes the VM's resize action. It should be stressed that in AC, autonomic elements interact with other autonomic elements through autonomic managers [30]. For clarity reasons, these components are omitted from Fig. 6, but should exist in the flow path ① ② and ③.

5 Inclusion of AMOCNA

This section enumerates all the steps towards integrating AMoCNA with the Cloud-native application environment. The internal integration between both systems leads to the facilitation of autonomic management.

AMoCNA enhances management of the concrete CNApp. However, to be used successfully, it imposes requirements on the CNApp. First, the CNApp must be running. Its execution environment should be exactly the one in which AMoCNA runs. And secondly, the CNApp should expose a HTTP metrics endpoint compliant with Prometheus [19] directives. This requirement is not obligatory for AMoCNA operation. Usually, the built-in metrics provided with different tools (such as the Prometheus Operator [38] project) are sufficient. The number of metrics is thousands. They cover many aspects, such as CPU or memory constraints. If CNApp does not expose a HTTP metrics endpoint, only observed are the default metrics.

Step-by-step guidance to include AMoCNA in a Cloud-native environment is as follows:

1. Recognition of autonomic management objectives - this process requires a deep analysis of AMoCNA capabilities. The high-level demands, correspond-

ing to the agreed SLA, should be assigned to low-level executors. This step determines the targets for autonomic management.
2. Identification of managed elements - in this step, the execution environment of a CNApp has to be divided into parts concerning the autonomic management targets determined in the previous step.
3. Initial setup - this step locates the MRE-K loop components inside the distinguished layers of the Cloud-native execution environment. The initial setup installs components that enable observability and monitoring of the appropriate Quality of Service (QoS) parameters and execution of specified management actions. A centralized, declarative management policies supervisor embraces the entire system. The supervisor is a mandatory prerequisite for CNApp autonomic management.

Following the above rules ensures AMoCNA fully integration with a CNApp execution environment. Hence, it enhances the autonomic management capabilities of the execution environment.

6 Autonomic element microservice

Figure 1 depicts two types of microservices that make up the AMoCNA framework. Namely the autonomic element microservices (bottom) and a single management policies microservice (top). The capabilities of each microservice align with distinguished layers. The prior microservice includes instrumentation, observation, low-level management aspects, processing of measurement data, and reasoning over them. The latter microservice focuses on declaring and governing management policies.

This section focuses on the microservice of the autonomic element. The structure of this microservice and its capabilities influence the autonomic features of the Cloud-native execution environment. Among others, its components divide according to the loop letters:

- Monitoring Controller – this controller gathers the measurements from the Cloud-native execution environment, including CNApp metrics, and then exposes the collected data for further processing.
- Reasoning Controller – its task is to facilitate reasoning over metrics exposed by prior controller. The metrics build facts inserted into a rule engine.
- Execution Controller – it strongly depends on the management actions possible to be enforced in the Cloud-native execution environment. The Execution Controller closes the MRE-K loop and is the last link in the runtime reinforcement of declarative management policies.

The further description focuses on the mentioned controllers (that relate to the distinguished layers of AMoCNA architecture depicted in Fig. 1), with particular emphasis on the execution controller.

The orchestrator comprises the execution environment of CNApps. We chose Kubernetes [39] as a basis of the AMoCNA framework. It has a proper level of abstraction, and the most important, CNCF recommends its usage. Also, it is industry accepted. It is worth mentioning that AMoCNA was also successfully tested with Docker Swarm [16].

6.1 Monitoring Controller

In case of the present Platform Specific Model (PSM) (a concept borrowed from the (MDA) [7] technology), it is recommended to strictly adhere to the monitoring metrics of the *Prometheus* directives. For provisioning the monitoring stack, we highly recommend using kube-prometheus [37]. The gained virtue is automation. Additional enhancements are the already defined Grafana dashboards that attractively illustrate the current state of all components of the execution environment.

6.2 Reasoning Controller

A rule engine forms the central core of the reasoning controller. As mentioned in our paper [34], Drools KIE Server [17] is a rule engine. It is deployed in a standalone Docker container[3]. It is a web application hosted in a JBoss Application Server (Wildfly). It exposes REST, JMS, and Java interfaces to client applications. The current prototype uses the Java API to insert Fact objects to working memory of the KIE Execution Server and to instantiate and execute rules. Covering with existing Java API, communication with KIE Server omits the need to parse JSON response as during communication with Prometheus Server.

[3] https://hub.docker.com/r/jboss/kie-server/

After reasoning against collected measurements and declared policies, the next step is to execute the resulting actions.

6.3 Execution Controller

Figure 7 shows the characteristics of an execution controller whose primary objective is to enforce and execute the declared management policies. Crucial steps include observations of all components of the Cloud-native Execution Environment.

In a nutshell, the execution controller is a server that listens on a socket. The incoming clients' requests result from matching defined rules against the metric facts. The request is then passed to the proper `Handler` responsible for executing the given management actions. When the execution controller container starts, it goes through two stages. First, it launches a TCP server that listens only for policy execution requests. Second, it generates a hashmap that transforms all declared policy actions into `Executor` objects.

The execution controller structure conforms to the Executor notion depicted in Fig. 8. The definitions of the entities presented are as follows:

Definition 1 **Execution Controller** ($ExeCtrl$) is a set of Executors ($e \in ExeCtrl$). It bears the burden of enforcement and execution of all declarative management policies in the Cloud-native execution environment.

$$ExeCtrl = \{e_1, e_2, \ldots e_i\} \tag{1}$$

Definition 2 The **Executor** (e_i) is an element responsible for executing a single declared management policy.

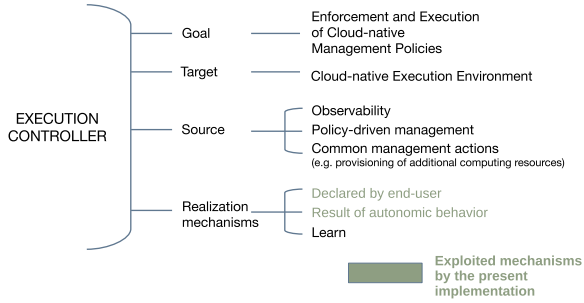

Fig. 7 Characteristics of the Execution Controller

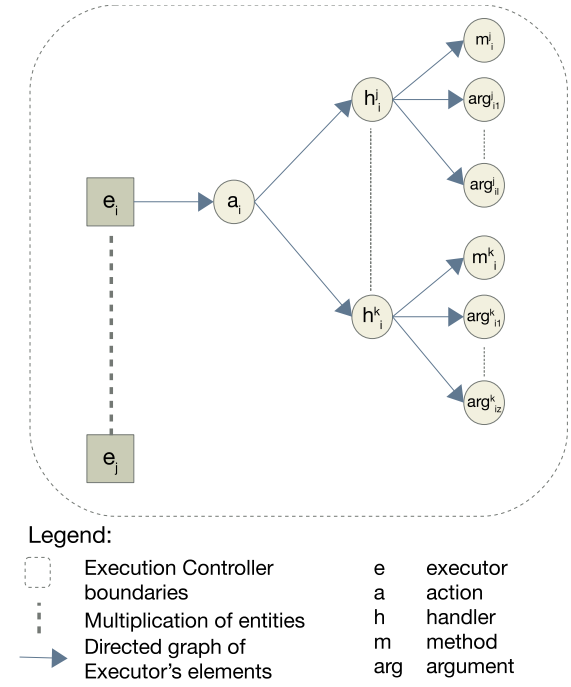

Fig. 8 Foundations of the Execution Controller model

An Executor is dependent on the enforced management action (a_i). Examples of management actions include `scale_nodes`, `increase_memory_limits`, etc.

Definition 3 A **handler** (h_i^j) is composed of a method (m_i^j) that invokes operations on the effectors of components of the Cloud-native execution environment. Also, composed of the arguments (arg_i^{jl}) that this method takes. The following equation expresses these relationships:

$$h_i^j := \left(m_i^j, arg_i^{j1}, \ldots, arg_i^{jl}\right) \tag{2}$$

To sum up, Fig. 8 shows that a single declared management policy ($\eth$) can be enforced by a single Executor:

$$p(\eth) \quad \rightarrow \quad ExeCtrl = \{e_1, e_2, \ldots e_i\} \tag{3}$$

where p is a procedure that maps declared policies to a set of Executors. The Executor, in turn, reflects the management action. The action can be assigned to multiple handlers. All defined variables of the Execution Controller model are described in Table 2.

Table 2 Variables defined by the Execution Controller model

Symbol	Name	Description
$ExeCtrl$	Execution Controller	Its objective is to enforce and execute the declared management policies.
e	Executor	It is responsible for executing a single declared management policy.
a	Action	It is a component of a declared management policy.
h	Handler	It handles the execution of a part or whole action.
m	Method	It invokes operations on the proper effectors.
arg	Argument	It is one of the arguments taken by the above method.

As already stated, the execution controller's functionality comes from declared management policies, and in this PSM are configured externally through a `json` file, mounted as a volume to the execution controller pod. The ConfigMap abstraction decouples coarse-grained information from the execution controller and injects the Pod with configuration data. A fragment of the `executor-config` ConfigMap object is attached as the Listing 1.

```
kind: ConfigMap
apiVersion: v1
metadata:
  creationTimestamp: 2019–04–18T19:14:38Z
  name: executor–config
  namespace: monitoring
data:
   executionController.json: |–
     {
        "executors": [
           {
             "action":"resize_cluster",
             "handler":"ResizeClusterHandler",
             "method":"addNode",
             "arg":"nodeIP"
          },
          {
            "action":"change_pod_cpu_request",
            "handler":"PodCPURequestHandler",
            "method":"changeRequest",
            "arg":"newValue"
          },
```

Listing 1 Fragment of `executor-ConfigMap.yaml` manifest file

The Executor objects, created from the declared ConfigMap, have four properties that sufficiently describe their operation. The properties are: (i) "*action*" – is a mapping from management policies into actions, which are enforced in a Cloud-native execution environment, (ii) "*handler*" – defines the class providing for handling the action, (iii) "*method*" – specifies the particular implementation of the action, and (iv) "*arg*" – stores the argument of method itemized in the previous point.

The rule engine evaluates the particular rule at runtime and executes its actions. The actions create a new object that acts as a client of the execution server. The accomplishment of the declared management policies in the Cloud-native context is highly dependent on the configuration of the orchestration environment. Kubernetes Objects API, available via REST, has official clients in the programming languages [12] that publish most of the functions exposed by Kubernetes objects. The Execution Controller closes the MRE-K loop and is the final link in the runtime reinforcement of declarative management policies.

7 Evaluation of AMoCNA

The AMoCNA framework is thoroughly evaluated in [34] and [35]. The results obtained positively assess this framework. The framework monitors not only the Cloud-native application but based on observations of the entire Cloud-native application execution environment, AMoCNA triggers actions across the Infrastructure, Containerization, and Application layers, enabling a holistic control of the Cloud-native application's execution performance. In the carried experiment, we will show this ability. The evalua-

tion of AMoCNA regards runtime adjustments of the Cloud-native execution environment. The experiment will show the autonomic management accomplished by AMoCNA in the Containerization and Infrastructure Layers of the execution environment.

To provide a baseline for the experiments, we set up a Kubernetes cluster consisting of one master node and six worker nodes. Table 3 presents the installed software. In this testbed, we deployed a Sock Shop Cloud-native application under AMoCNA supervision. We assess the Sock Shop microservices from the latency perspective. Among the microservices, the highest latency has the front-end microservice. We chose it as a representative one. To simulate user behavior, we used the Locust load generator tool. In every rerun of the experiment, we used the same values of the load, i.e., 3000 users that totally generated 15000 requests. We repeated the experiment ten times, each providing the same results. We established the example of SLA criteria. They state:

1. Each microservice latency is less than 2.5 s.
2. Cluster CPU utilization is less than 60%.

Th above configuration caused a violation of SLA. The situation is depicted in both Fig. 9 (left graph) and Fig. 10 (left peak in the graph). It indicates that the CNApp was flooded with user requests. The situation is also observed in the metrics gained that exceed the SLA threshold. It was necessary to trigger the appropriate actions. The actions proceeded with microservice (strictly container) redeployment and cluster autoscaling. Container redeployment with different settings is an obvious functionality of orchestration systems. This functionality is an example of management in the Containerization Layer. AMoCNA goes one step further. It proceeds with the redeployment autonomously and based on the observations. For the management in the Infrastructure Layer and cluster autoscaling, Kubernetes provides a Cluster Autoscaler[4] controller that automatically adjusts the cluster size. AMoCNA augments the possibilities of this controller, and instead of taking into account only the resource requests and limits specified before starting the CNApp, the decision is made based on the observations of resources actual utilization. The cluster can also scale at runtime.

For the proof of concepts, we composed three rules. Two rules trigger (Listing 2) and enforce (Listing 3) the modification in the microservice CPU request. The third rule (Listing 4) manages cluster autoscaling. In this PSM, we used Jboss Drools [17] as a rule engine. The listings are in the .drl language.

```
import translator.metrics.ContainerMetric;
import translator.metrics.Metric;
import java.lang.Double;
import translator.SLA;
import pl.edu.agh.informatyka.amocna.CPU;
import translator.metrics.SockShopMetric;

rule "trigger_pod_CPU_request"
  dialect "mvel"
  when
        ssMetric : SockShopMetric( service
        == "front-end" && namespace == "sock-shop"
        && name == "request_duration_seconds_sum"
        && eval( Double.parseDouble
        (ssMetric.getValue()) > SLA.FRONTEND_LATENCY))
        m : ContainerMetric( container_name
        == "front-end" , namespace == "sock-shop" ,
        name == "container_cpu_usage_seconds_total" ,
        eval( Double.parseDouble(m.getValue()) >
        CPU.request("front-end") ))
  then
        ContainerMetric insertedFact = new
        ContainerMetric();
        insertedFact.setContainer_name( "front-end" );
        insertedFact.setNamespace( "sock-shop" );
        insertedFact.setName(
        "avg_container_cpu_usage_seconds_total" );
        insertedFact.setValue( m.query("front-end",
        "sock-shop") );
        insertLogical( insertedFact );
end
```

Listing 2 A rule adjusting Pod's requests for CPU. First rule.

```
import translator.metrics.ContainerMetric;
import java.lang.Double;
import pl.edu.agh.informatyka.amocna.CPU;
import pl.edu.agh.informatyka.amocna.executor.
    ExecutorServerCli;

rule "enforce_pod_CPU_request"
  dialect "mvel"
  when
        c : ContainerMetric( name ==
        "avg_container_cpu_usage_seconds_total" ,
        eval( Double.parseDouble(c.getValue()) >
        CPU.request("front-end","sock-shop") ))
        client : ExecutorServerCli( )
  then
        client.execute( "change_pod_cpu_request" );
end
```

Listing 3 A rule adjusting Pod's requests for CPU. Second rule.

[4] as an additional component.

Table 3 The software used in the evaluation

Name	Short description
OpenStack	an open-source cloud platform [43].
Kubernetes	also known as K8s is an open-source system for orchestrating containerized applications [39].
Prometheus Operator	provides Kubernetes deployment and management of monitoring components based on the Prometheus [38].
JBoss Drools	is a Business Rules Management System (BRMS) solution written in Java [17].
Sock Shop Microservices	a Microservices Demo Application that simulates an e-commerce website that sells socks [50].
Stress	A Docker container that generates CPU, memory, I/O, and disk loads. It simulates increases in resource consumption and, therefore, failures to meet SLA commitments [1].
Locust	An open-source load-testing tool that simulates user behavior and swarms the system with millions of simultaneous users. Therefore enables to present flooding microservice with requests and hence simulates failures to meet SLA commitments with regard to response latency [40].

The result of execution of rules (Listings 2 and 3) operating in the Containerization Layer is shown in Fig. 9. The graph on the left side that presents the state of the CNApp before the AMoCNA adjustment indicates that the SLA threshold is significantly overcome. Its front-end microservice latency reached 5 s (2.5 s is allowed). The observability capabilities of AMoCNA (rule in Listing 2) detect this situation. As a consequence, AMoCNA triggers and then invokes ExecutorController that modifies the CPU request (this Controller is defined in Listing 1 as a second element of the `executors` table). As a result, the microservice front-end latency decreases to 2.5 s (depicted in the right-hand graph of Fig. 9).

```
import translator.metrics.ClusterMetric;
import translator.metrics.Metric;
import java.lang.Double;
import translator.SLA;
import pl.edu.agh.informatyka.amocna.
executor
    .ExecutorServerCli;

rule "SLA_cluster_cpu_utilization"
  dialect "mvel"
  when
       m : ClusterMetric( name ==
       "cluster_cpu_utilisation:lm" ,
       eval( Double.parseDouble
       (m.getValue()) > SLA.CLUSTER_CPU ))
    client : ExecutorServerCli( )
  then
       client.execute("cluster_resize");
end
```

Listing 4 A rule guarding cluster CPU utilization

On the other hand, the result of execution of rule presented in the Listing 4 operating in the Infrastructure Layer is shown in Fig. 10. The first peak, which shows the state of CNApp before AMoCNA adjustment, indicates that the SLA threshold has been overcome. The CPU utilization of the cluster exceeds the SLA threshold (60% is allowed). The AMoCNA capabilities (rule in Listing 4 detect this situation. Consequently, it triggers and then invokes ExecutorController that proceeds cluster autoscaling (this Controller is defined in Listing 1 as the first element of the `executors` table). As a result, a new node was added to the cluster. After joining another node (`worker-7` in this case), the utilization of the entire cluster automatically decreases below the SLA level. Its value is less than 60%, which denotes the second peak.

7.1 Evaluation summary

The experiment shows the autonomic management accomplished by AMoCNA in the Containerization and Infrastructure Layers of the execution environment. We present the reduction of the latency and CPU utilization. The functionality is possible through properly defined executors. Hence it is significant to put more emphasis on its development.

The experiments carried out proved the importance of comprehensive observations. The knowledge gained in this process is a good authority on information on the past and current state of the Cloud-native Applications. Based on this information, insights are provided, and appropriate autonomic management actions are triggered.

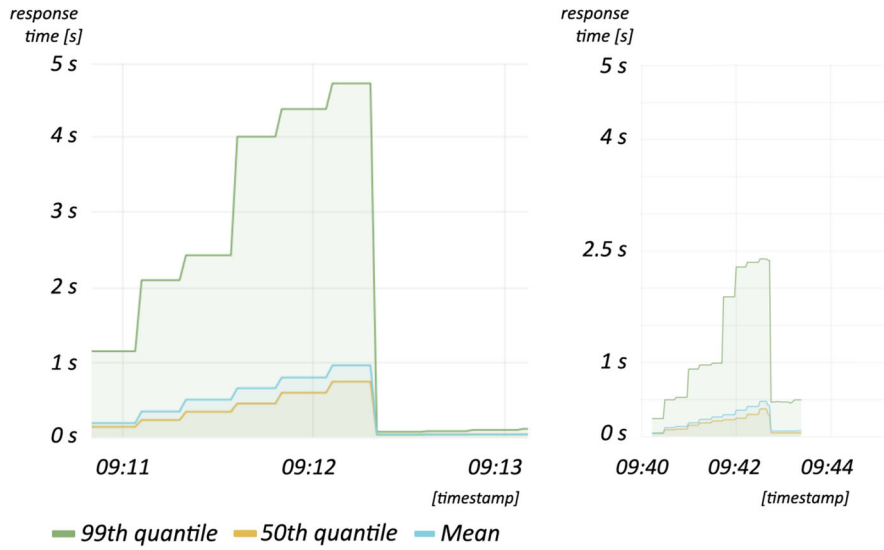

Fig. 9 Front-end microservice latencies before (left graph) and after (right graph) AMoCNA adjustments

8 Conclusions

In this paper, we discuss some details of the AMoCNA operation model. The development of the AMoCNA system is a significant achievement in proving the correctness of the proposed concepts. This system supports dynamic, flexible, and scalable Cloud-native environments. We show how easily AMoCNA can be combined with Kubernetes and used as a conceptual framework for Kubernetes extensions.

In conclusion, the AMoCNA structure corresponds to the core concepts of Kubernetes platform design. However, AMoCNA offers more general and robust management functionalities than the elemental Kubernetes reconciliation loop, which we present in Section 3. We propose to extend Kubernetes with AMoCNA

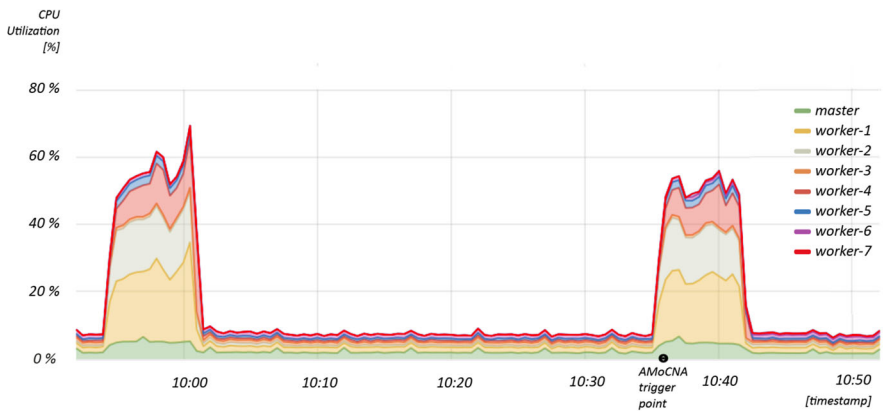

Fig. 10 CPU Utilization of the cluster.Before the AMoCNA trigger point, node `worker-7` does not exist. The CPU Utilization of the node is 0% and is not part of the CPU Utilization of the entire cluster. It is highlighted with the topmost (red) line in the figure. Note that the CPU Utilization of a node is not an area between its top boundary and X-axis but the top boundary of the node before it. The AMoCNA trigger point starts the worker-7 node. At first, the new node is underutilized. The current workload running in the cluster does not immediately migrate to it. The CPU Utilization of the `worker-7` node is still near 0%. Its line on the graph does not change significantly. However, we can notice its existence in the new CPU Utilization of the cluster, which is less than 60%.

and its autonomic elements that operate according to feedback control and perform declared policies. We give the guidelines for including autonomic features in Cloud-native environments. They are described in Section 5 and are the main contribution of our research.

In contrast to Kubernetes management, resources do not simply maintain the desired state. Their state continuously adapts according to the declared management policies. The policies trigger the appropriate management actions that proceed according to the execution controller (described in Section 6). The concept of an execution controller is also our contribution to Cloud-native. The positive results of the evaluation show that it is beneficial to enhance the Cloud-native applications with autonomic features and develop suitable executors.

AMoCNA operation is not only limited to container resources. Kubernetes operates solely in the containerization layer. On the other hand, AMoCNA actions perform across the infrastructure, containerization, and application layers, allowing holistic control of Cloud-native applications execution. The proposed hierarchical composition of autonomic elements makes it possible to implement multilayer management policies. This capability influences three system layers (application, containerization, and infrastructure). However, the experiment described shows only the influence of the containerization layer. Future research can propose a specification of a strongly reconfigurable controller that can be valuable in many different contexts. Worth exploring is its usage among all layers of the execution environment and showing how the others, not only the containerization layer, are influenced.

Acknowledgements The research presented in this paper was supported by the funds assigned to AGH University of Science and Technology by the Polish Ministry of Science and Higher Education.

Author contributions This research work is part of Joanna Kosińska Ph.D. work, which was conducted under the supervision of Krzysztof Zieliński. This paper is a guide to including autonomic features in Cloud-native environments. For this purpose, we recommend using our AMoCNA framework. We specify all the steps needed to integrate the proposed AMoCNA framework into the Cloud-native application environment. We have carried out the work presented in this paper over the past 4 years. All authors read and approved the final manuscript.

Funding Information Funding information is not applicable / No funding was received.

Data Availibility Statement Data sharing is not applicable to this article, as no data sets were generated or analyzed during the current study.

Declarations

Conflicts of interest The authors declare that they have no conflict of interest.

Ethical Approval Not applicable.

References

1. A Docker container for stress, a tool for generating workload (Last seen on March 2021) https://hub.docker.com/r/progrium/stress
2. Abeywickrama, D.B., Ovaska, E.: A survey of autonomic computing methods in digital service ecosystems. Service Oriented Computing and Applications **11**(1), 1–31 (2017)
3. Achilleos, A.P., Kritikos, K., Rossini, A., Kapitsaki, G.M., Domaschka, J., Orzechowski, M., Seybold, D., Griesinger, F., Nikolov, N., Romero, D., Papadopoulos, G.A.: The cloud application modelling and execution language (camel). Journal of Cloud Computing **8**(1), 20 (2019)
4. Agrawal, D., Calo, S., Lee, K.w., Lobo, J., Verma, D.: Policy Technologies for Self-Managing Systems, 1st edn. IBM Press, USA (2008)
5. Antonopoulos, N., Anjum, A., Gillam, L.: Intelligent techniques and architectures for autonomic clouds: introduction to the itaac special issue. J Cloud Comput. p 1:18, (2012) https://doi.org/10.1186/2192-113X-1-18
6. Arcaini, P., Riccobene, E., Scandurra, P.: Modeling and analyzing mape-k feedback loops for self-adaptation. In: 2015 IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, pp 13–23, (2015) https://doi.org/10.1109/SEAMS.2015.10
7. Arlow, J., Neustadt, I.: Enterprise Patterns and MDA: Building Better Software with Archetype Patterns and UML. Addison Wesley Longman Publishing Co., Inc, USA (2003)
8. Aydemir, F., Başçiftçi, F.: Building a performance efficient core banking system based on the microservices architecture. J Grid Comput. **20**(4), 37 (2022)

9. Bennett, S.: A History of Control Engineering 1930–1955, 1st edn. Peter Peregrinus, GBR (1993)
10. Capodieci, N., Hart, E., Cabri, G.: Designing selfaware adaptive systems: from autonomic computing to cognitive immune networks. In: Proceedings of the 7th International Conference on SelfAdaptation and Self-Organizing Systems Workshops, SASOW , IEEE Computer Society, Conference Publishing Service, Los Alamitos, California USA, pp 59–64, (2013)
11. Carrión, C.: Kubernetes as a standard container orchestrator - A bibliometric analysis. Journal of Grid Computing **20**(4), 42 (2022). https://doi.org/10.1007/s10723-022-09629-8
12. Client Libraries for Kubernetes API (Last seen on May, 2023) https://kubernetes.io/docs/reference/using-api/client-libraries/
13. Cloud Native Computing Foundation (Last seen on March 2022) https://www.cncf.io
14. Cloud Native LandScape (Last seen on March, 2021) https://github.com/cncf/landscape
15. Cohen, I.R.: Discrimination and dialogue in the immune system. Seminars in Immunology **12**(3), 215–219 (2000). https://doi.org/10.1006/smim.2000.0234
16. Docker Site (Last seen on November, 2021) https://www.docker.com
17. Drools Site - A Business Rules Management System (BRMS) solution (Last seen on May, 2022) https://www.drools.org
18. Ehrlinger, L., Rusz, E., Wöß, W.: A survey of data quality measurement and monitoring tools. CoRR abs/1907.08138, (2019) arxiv preprint arxiv:1907.08138
19. From metrics to insight (Last seen on January, 2022) https://prometheus.io
20. Ganek, A.G., Corbi, T.A.: The dawning of the autonomic computing era. IBM Systems Journal **42**(1), 5–18 (2003). https://doi.org/10.1147/sj.421.0005
21. Gill, S.S., Buyya, R.: Resource provisioning based scheduling framework for execution of heterogeneous and clustered workloads in clouds: from fundamental to autonomic offering. Journal of Grid Computing **17**(3), 385–417 (2019)
22. Gonzalez, N.M., Carvalho, T.C.M.D.B., Miers, C.C.: Cloud resource management: Towards efficient execution of large-scale scientific applications and workflows on complex infrastructures. J Cloud Comput **6**(1), (2017) https://doi.org/10.1186/s13677-017-0081-4
23. Gonçalves, G., Endo, P.T., Santos, M., Sadok, D., Kelner, J., Melander, B., Mångs, J.E.: Cloudml: An integrated language for resource, service and request description for d-clouds. IEEE Third International Conference on Cloud Computing Technology and Science pp 399–406 (2011)
24. Hightower, K., Burns, B., Beda, J.: Kubernetes: Up and Running Dive into the Future of Infrastructure, 1st edn. O'Reilly Media, Inc., USA, (2017)
25. Hitzler, P., Krtzsch, M., Rudolph, S.: Foundations of Semantic Web Technologies, 1st edn. Chapman & Hall/CRC (2009)
26. Huebscher, M.C., McCann, JA.: A Survey of Autonomic Computing – Degrees, Models, and Applications. ACM Comput Surv **40**(3),7:1–7:28, (2008) https://doi.org/10.1145/1380584.1380585
27. Ibryam, B., Huß, R.: Kubernetes Patterns: Reusable Elements for Designing Cloud-Native Applications. O'Reilly Media, USA, (2019) https://books.google.pl/books?id=8WmRDwAAQBAJ
28. Jahan, S., Riley, I., Walter, C., Gamble, R.F., Pasco, M., McKinley, P.K., Cheng, B.H.: Mape-k/mape-sac: An interaction framework for adaptive systems with security assurance cases. Future Generation Computer Systems **109**, 197–209 (2020). https://doi.org/10.1016/j.future.2020.03.031
29. Kalman, R.: On the General Theory of Control Systems. IRE Trans Autom Control **4**, 110–110 (1960)
30. Kephart, J.O., Chess, D.M.: The Vision of Autonomic Computing. Computer **36**(1), 41–50 (2003). https://doi.org/10.1109/MC.2003.1160055
31. Khan, Z., Ludlow, D., McClatchey, R., Anjum, A.: An architecture for integrated intelligence in urban management using cloud computing. In: Fourth IEEE International Conference on Utility and Cloud Computing, pp 415–420, (2011) https://doi.org/10.1109/UCC.2011.69
32. Kim, G., Debois, P., Willis, J., Humble, J.: The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations. IT Revolution Press, USA (2016)
33. Koehler, M.: An adaptive framework for utility-based optimization of scientific applications in the cloud. Journal of Cloud Computing **3**(1), 4 (2014)
34. Kosińska, J., Zieliński, K.: Autonomic management framework for cloud-native applications. Journal of Grid Computing **18**(4), 779–796 (2020)
35. Kosińska, J., Zieliński, K.: Experimental evaluation of rule-based autonomic computing management framework for cloud-native applications. IEEE Trans Serv Comput **16**(2), 1172–1183 (2023). https://doi.org/10.1109/TSC.2022.3159001
36. Kritikos, K., Zeginis, C., Iranzo, J., Gonzalez, R.S., Seybold, D., Griesinger, F., Domaschka, J.: Multicloud provisioning of business processes. J Cloud Comput. **8**, 18 (2019). https://doi.org/10.1186/s13677-019-0143-x
37. Kubernetes cluster monitoring with Prometheus using the Prometheus Operator. (Last seen on February) (2022) urlhttps://github.com/prometheus-operator/kube-prometheus
38. Kubernetes native deployment and management of Prometheus and related monitoring components. (Last seen on February) (2022) https://github.com/prometheus-operator/prometheus-operator
39. Kubernetes Site (Last seen on July) (2022) https://kubernetes.io
40. Locust Homepage (Last seen on June) (2022) https://locust.io/
41. Mesos Site (Last seen on April) (2021) http://mesos.apache.org
42. Newman, S.: Building Microservices: Designing Fine-Grained Systems, 1st edn. O'Reilly Media, USA (2015)
43. OpenStack Site (Last seen on April) (2022) https://www.openstack.org/
44. Pahl, C., Jamshidi, P., Weyns, D.: Cloud architecture continuity: Change models and change rules for sustainable cloud software architectures. Journal of Software: Evolution and Process **29**(2), e1849 (2017). https://doi.org/10.1002/smr.1849

45. Park, S., Park, S., Park, Y.B.: An architecture framework for orchestrating context-aware it ecosystems: A case study for quantitative evaluation. Sensors (Basel, Switzerland) **18**(2), 562 (2018)
46. Peinl, R., Holzschuher, F., Pfitzer, F.: Docker cluster management for the cloud - survey results and own solution. J Grid Comput **14**(2),265–282, (2016) https://doi.org/10.1007/s10723-016-9366-y
47. Ramanathan, R., Raja, K.: Handbook of Research on Architectural Trends in Service-Driven Computing, 1st edn. IGI Global, Hershey, PA, USA (2014)
48. Redbooks, IBM and International Business Machines Corporation International Technical Support Organization. A Practical Guide to the IBM Autonomic Computing Toolkit. IBM Redbooks, IBM, International Support Organization, USA, (2004) https://books.google.pl/books?id=XHeoSgAACAAJ
49. Singularity Site (Last seen on April) (2021) https://sylabs.io/singularity/
50. Sock Shop - A Microservices Demo Application (Last seen on April) (2022) https://microservicesdemo.github.io
51. Sterritt, R., Parashar, M., Tianfield, H., Unland, R.: A Concise Introduction to Autonomic Computing. Adv Eng Inform **19**(3), 181–187 (2005). https://doi.org/10.1016/j.aei.2005.05.012
52. Tomarchio, O., Calcaterra, D., Modica, G.D.: Cloud resource orchestration in the multi-cloud landscape:A systematic review of existing frameworks. J Cloud Comput **9**(1), 49 (2020)
53. Turnbull, J.: The Art of Monitoring. James Turnbull, USA, (2014) https://books.google.pl/books?id=w5QfDAAAQBAJ
54. Vassev, E., Hinchey, M.: Knowledge representation and awareness in autonomic service-component ensembles - state of the art. In: 14th International Symposium on Object/Component/ServiceOriented Real-Time Distributed Computing Workshops, ISORC Workshops 2011, Newport Beach, CA, USA, March 28-31, IEEE Computer Society, USA, pp 110–119, (2011) https://doi.org/10.1109/ISORCW.2011.21
55. Wardm, J.S., Barker, A.: Observing the clouds: A survey and taxonomy of cloud monitoring. J Cloud Comput **3**(1), 24 (2014)
56. Zhou, N., Georgiou, Y., Pospieszny, M., Zhong, L., Zhou, H., Niethammer, C., Pejak, B., Marko, O., Hoppe, D.: Container orchestration on hpc systems through kubernetes. Journal of Cloud Computing **10**(1), 16 (2021)