Autonomic Management Framework for Cloud-Native Applications

Joanna Kosińska · Krzysztof Zieliński

Received: 27 November 2019 / Accepted: 2 August 2020 / Published online: 26 September 2020
© The Author(s) 2020

Abstract In order to meet the rapidly changing requirements of the Cloud-native dynamic execution environment, without human support and without the need to continually improve one's skills, autonomic features need to be added. Embracing automation at every layer of performance management enables us to reduce costs while improving outcomes. The main contribution of this paper is the definition of autonomic management requirements of Cloud-native applications. We propose that the automation is achieved via high-level policies. In turn autonomy features are accomplished via the rule engine support. First, the paper presents the engineering perspective of building a framework for Autonomic Management of Cloud-Native Applications, namely AMoCNA, in accordance with Model Driven Architecture (MDA) concepts. AMoCNA has many desirable features whose main goal is to reduce the complexity of managing Cloud-native applications. The presented models are, in fact, meta-models, being technology agnostic. Secondly, the paper demonstrates one possibility of implementing the aforementioned design

procedures. The presented AMoCNA implementation is also evaluated to identify the potential overhead introduced by the framework.

Keywords Autonomic Computing (AC) · Cloud-native · Resource management · Policy-driven management · Observability

1 Introduction

System components and software elements have been evolving for decades to deal with the increased complexity of system control, resource sharing and operational management [16]. While Cloud-native is not a novel concept, it remains at the forefront of software development. This approach has not yet been tested with Autonomic Computing (AC) [44], although its usage in this context seems to be a natural step. Such architectures can effectively address the overall complexity of resource management. Fundamentals of Autonomic Computing (AC) paradigm constitutes autonomic elements that are responsible for policy-driven self-management of particular system components. The present paper focuses on autonomic management of Cloud-native applications (abbreviated as CNApps) through observation of all their internal components that simply are represented by autonomic elements. The observations are consumed by appropriate sensors of autonomic elements which in turn we propose to realize with the support of a rule engine.

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

K. Zieliński
e-mail: kz@agh.edu.pl

CNApp represents a graph of communicating microservices running as containers. Its QoE and QoS are much determined by an orchestration process which is often defined [31] as the automated configuration, coordination, and management of computer systems and software. Orchestration process is so tightly coupled with a CNApp so setting it up can be considered as an integral part of the application. Such a holistic view adopted in this paper leads to the conclusion that the CNApp management has to address the whole CNApp execution environment. From this conclusion results that observations must be performed across all levels in the Cloud-native application stack.

The main contribution of this paper is (i) the definition of autonomic management requirements of Cloud-native applications, (ii) identification of principles leading to their fulfillment and (iii) presentation of an autonomic management framework for Cloud-native applications which involves technical implementation of the proposed concepts. Embracing automation at every layer of performance management enables us to reduce costs while improving outcomes. The outcomes regard mainly accomplishing administrative tasks such as configuration. The proposed extension of current Cloud-native environments is based on knowledge processing capabilities to detect, correlate and respond to diverse events across multiple real-time data sources. The measurements and the reasoning which bases upon them depend on the type and number of components being instrumented. Cloud-native environments provide a rich mixture of component types (computing resources, container engines, containers, orchestrators, etc.) whose observability is a must. Furthermore, the research analyses several technology stacks and standards, and then selects specific ones to build a development environment for the proposed solution.

The structure of this paper is as follows. First, the aim and contribution of the research is presented. Section 2 outlines current standards and technologies related to management of Cloud-native applications. On the basis of this knowledge, the following section proposes the requirements for the framework for autonomic management of CNApps. We then delimit the research area by providing an abstract model of a Cloud-native execution environment. Abstracting resource management solves problems related to continuous evolution of Cloud-native environment

components. This abstraction, bound with the Autonomic Computing (AC) paradigm, results in fully autonomic management of Cloud-native applications. Following a succinct description of the encountered obstacles, the paper introduces a framework for Autonomic Management of Cloud-Native Applications (AMoCNA). Its usefulness is assessed in overhead and performance tests and also through comparison with other models, such as the service mesh. Finally, the paper is summarized and we propose directions for further development and research.

2 Related Work

The Cloud-native concept is built around containerization philosophy, and the management of those containers through orchestration tools. Starting more and more containers and containerized applications, divided into hundreds of pieces, complicates their management and orchestration [19]. Orchestrator logically orders the loosely coupled microservices into a dependency graph and organizes their deployment. They solve potential issues that might rise among containers run across many machines. This includes high availability, scaling, replication, fault tolerance and isolation. The problem with heterogeneity already at the virtual machine level is interestingly solved in [3]. The proposed solution is implemented and used within the INDIGO-DataCloud project. The research provides blueprints for orchestration model using Docker containers in an homogeneous and transparent way providing consistent application deployment for users that can use hybrid clouds.

Three most popular open-source container orchestration tools are: Docker Swarm [10], Kubernetes [25] and Apache Mesos [36]. These orchestrators offer resource management mechanisms to prevent SLA violations and increase resource cost-efficiency in container-centric environments [50], but only considering a limited area of Cloud-native application stack. A comprehensive study of Docker cluster management tools and solutions for the cloud is presented in [41]. It analyzes software tools and gathers them in a table with marked fulfilled requirements. The research points out that there is no single tool fulfilling all requirements.

The autonomic behavior of the system calls for specification of the goal that should be achieved as

a result of adaptation or autonomous processes. This goal is usually expressed as a collection of Quality of Service (QoS) or QoE values [24]. These metrics are available only if the system's components are observable [26]. Cloud-native Computing Foundation (CNCf) treats this feature as a fourth, not-obligatory step in its Cloud-native landscape [6]. Different approach to monitoring measurements is proposed in [17]. Observability is addressed there with the method of data-driven monitoring that is based on the tracing the stream along the paths.

Results of gathered observations allow to form management policies. Some tools addressing observability and policy-based management in Cloud-native environments have been developed. It is worth to familiarize with research presented in [31]. The general idea is similar to ours, but the policies regard only autoscaling mechanisms and they are not defined by ordinary users, only by skilled professionals. The limitations result from using alert mechanisms that are available in most monitoring platforms. In turn we propose to accomplish management of policies by the support of a rule engine. Nonetheless there is still an inevitable need to transform collected metrics into executable actions in order to effectively improve customer experience (QoE) and ROI.

Regarding Cloud-native environment management, many of the analyzed solutions, including enterprise-class ones (e.g. Docker UCP [11], Sematext [22] Docker Agent, Newrelic [39], Dynatrace [12], etc.), propose also an alerting and simple prediction framework. However, none support policy-based management, especially when runtime reconfiguration features are required.

A different but still interesting approach to CNApps management involves the Service Mesh model [9]. Service Mesh provides a generic mechanism for intercepting inter-microservice communications. Cloud-native environments do not enforce this type of architecture, but the CNCf organization nevertheless recommends its usage. Solutions which have been successfully applied in this area include Envoy [13], Istio [23] and Linkerd [35]. Following an introductory description, the Autonomic management framework for CNApps will be compared with Service Mesh, revealing the pros and cons of the latter approach.

Many concepts and software realizations have been proposed to address specific elements of the presented research. However, these are not linked and cannot

be evaluated as a whole. The stated problem is not addressed in any known publications – although there is related research in the scope of specific concepts, and there also similar concepts which concern old technologies [1, 4, 20, 33, 45, 49].

3 Cloud-native Application Autonomic Management Principles

The goal of every management platform is to focus on application logic while at the same time tracking end-user feedback. These Continuous Integration/Continuous Delivery (CI/CD) [8] characteristics are respected in Cloud-native environments. Prioritizing user feedback and satisfaction provides the means to obey Level Agreement (SLA).. As the level of end-user satisfaction is a subjective concept, it is very difficult to propose its formal specification. On the other hand, it is much easier to directly observe any part of an execution environment and carry out reasoning on its basis. The proposed models are consistent with the latter approach. Their specification aims to prove the need for research in the scope of CNApp management.

The substance of the matter is that the management process takes place in the environment where heterogeneous CNApps execute together. To precisely determine the objects being managed we decided to specify a notion of a Cloud-native execution environment. The model of a Cloud-native Execution Environment, depicted in Fig. 1, outlines investigations in the area of CNApp management. It provides a non-exhaustive and prescriptive guide to identifying and implementing key components required in such environments. Every CNApp has its own Cloud-native Execution Environment comprising all components (with their configuration) necessary to its deployment and run. The Cloud-native Execution Environment proposed in Fig. 1 is also compliant with the DevOps team set of operation principles and practices (the topmost layer). In the present model, details of CI/CD aspects are intentionally omitted, as they constitute a complex topic – although one worthy of further extensive research.

First, however, it is necessary to identify the requirements of AC and a *resource management* system [43] in the context of Cloud-native operation and

Fig. 1 An abstract view of a Cloud-native application execution environment

observability of its components. These requirements are as follows:

- Unlike an SOA system [51], the proposed solution should be *goal-driven* and dynamically composable rather than being deployed as monolithic services (even though this is, in principle, possible). This approach provides immediate insight into a complex and highly distributed execution environment.
- Aggregation of logs and metrics needs to be performed at every layer of the proposed enhancement of Cloud-native environments. This yields the true view of a distributed execution environment and consequently, provides a wide array of capabilities such as application health management, optimization, failure cause analysis, etc.
- As a result of the above requirement, the designed architecture of the proposed solution should include monitoring [18] and analytical services. Apart from visualizing these parameters to administrators, such metrics underpin further autonomic management.
- The detailed observability of all components of the execution environment should help maintain the expected level of QoS. These measurements also help proactively overcome SLA violations.
- Autonomic management, an important aspect of the proposed solution, is the process of accomplishing administrative tasks without or with only modest human intervention. Additionally this process adapts to dynamic changes in Cloud-native environment. We propose that the automation is achieved via high-level policies. In turn autonomy features are accomplished via the rule engine support.

- The provided blueprint for autonomic management of CNApps contains a closed feedback control loop, referred (by IBM) as the MAPE-K loop [20].
- The proposed declarative policy management model should be extensible. Given that the policy model remains abstract and not tied to any specific implementation, it can easily capture many aspects such as connectivity, security, communication, etc.
- The proposed solution should take into account the dynamic nature of a Cloud-native environment, especially changes in resource requirements and fluctuations in the level of resource availability. This capability is needed to adjust SLA at runtime.

Analysis of CNApps management solutions (Section 2) and requirements of Cloud-native environments (listed previously) has enabled us to propose a high-level view of the concept of management of Cloud-native environments, based on observability data (this concept is depicted in Fig. 2 as the upper component of the sum). The main component of this part, namely the Cloud-native Execution Environment, has already been described (refer to Fig. 1). The association of enforce management

actions with Cloud-native Autonomic Manager covers all actions that can be performed. From denoted multiplicity it comes that each Cloud-native Execution Environment is under control of a single Cloud-native Autonomic Manager.

CNApps adhere to any declared policy regarding all layers of the execution environment. The management process of the environment is governed by the Declarative Management Policy. This component is so general that it can cover any type of policy. Policies determine the actions which need to be taken when an event occurs and certain conditions are met [46]. The actions are executed by a Cloud-native Autonomic Manager. First, the management process is driven by manual guidelines of the Declarative Management Policies component. Second, in addition to human effort, which is required to preserve operation of all components in Cloud-native Execution Environment, the executed actions are supported by Observability characteristics.

The self-management capabilities of the Cloud-native environment overcome its dynamic properties which tend to increase over time. The AC paradigm addresses this issue. An AC system is built around

Fig. 2 Realization of a Cloud-native autonomic element

the concept of an autonomic element. An autonomic element is a fundamental building block of any autonomic system. It aims at hiding the complexity of overall management of the environment, particularly a Cloud-native environment.

We propose to combine the concept of management of Cloud-native environments with the reference model of the IBM control loop. This combination results in the model of a Cloud-native autonomic element. This model is presented in Fig. 2 and represents this paper's principal contribution to Cloud-native environments. Cloud-native autonomic elements are applied in this paper to enforce the declared management policies. Similarly as in AC, all internal components of Cloud-native execution environment are represented by autonomic elements. The autonomic element's MAPE-K loop can be realized in various ways, using diverse algorithms and techniques. The authors propose that the architecture of the Cloud-native autonomic element be grounded on the concept of a rule engine to process the declared management policies (an explanation of this approach is provided in [2, 27, 47]). For this reason the loop is named MRE-K (as it contains Monitoring, Rule Engine and Execute containers; the letter "K" is left unchanged). The Cloud-native autonomic element uses information and semantics to link the SLA [7] with the services offered by every layer of the Cloud-native execution environment. This information, together with the declared management policies, is processed by the MRE-K loop. As a result, the MRE-K loop produces a new configuration, enabling the execution environment to be modified at runtime. The execution environment is therefore equipped with policy-driven management capabilities regarded as standard in modern management solutions [5, 37, 38].

The next section describes the architecture of an autonomic management framework compliant with the proposed concepts. The framework has a multilayered architecture, with each layer covering different aspects of AC.

4 AMoCNA Model

This section presents design of the framework of a system for **Autonomic Management of Cloud-native Applications**. By taking only the initials of each word, the lengthy name is condensed into simply

AMoCNA. The proposed framework has been thoroughly described in [30] and is available at AGH University's Main Library.

The following section decomposes the AMoCNA architecture according to AC paradigms, thus realizing autonomic management of CNApps. The AMoCNA architecture is not based on any implementation technology; thus the designed models are compliant with Platform Independent Model (PIM) [14]. The presented layers realize the control loop capabilities, referred to as the MRE-K loop. Moreover, the AMoCNA framework is compliant with the AS3 element model presented in [29]. AMoCNA components are divided into five logical layers representing their capabilities. These layers together comprise the AMoCNA microservices, and are preconditions for its successful operation. As depicted in Fig. 3 these layers are:

- **Instrumentation Layer** – This is the bottom layer of AMoCNA. It is the only layer with direct access to the specific CNApp that is being managed. Access is ensured through Cloud-native autonomic elements that provide a link between AMoCNA and the given CNApp. The *Instrumentation Layer* contains two components, namely *Collector* and *Actuator*. *Collector* gathers data from different sensors, and passes it – unmodified, but decorated e.g. with timestamps – for further processing in *Observation Layer*. On the other hand, *Actuator* retrieves actions from the *Management Layer* that have to be enforced on the concrete CNApp. It then (through its effectors) executes the management actions upon resources.
- **Observation Layer** – This layer is tightly coupled with monitoring which supports collection of up-to-date measurements. The inter-layer flow includes communication with the *Control Layer*, but from the *Instrumentation Layer*, which lies directly below. The *Observation Layer* consists of two components - *Observation Controller* and *Fault Detector* which interoperate in a loop. The former drives the overall observation process, while the latter can be thought as an alerting tool, which, upon detecting a failure, informs the appropriate entities. *Fault Detector* does not contain any logic

Fig. 3 Microservices architecture of AMoCNA framework

that correlates events to anticipate failures or analyze their root causes.

- **Management Layer** – This layer operates at the same level as the Observation Layer. It sits atop the Instrumentation Layer and communicates with it on the basis of information retrieved from the Inference Layer. The Management Layer consists of two components - *Management Controller* and *Runtime Adjustor* which interoperate in a loop. The former drives the overall management process, directing management actions to appropriate Cloud-native autonomic elements (the current autonomic element is not necessarily the one which executes the management action; indeed, it can also collaborate with other Cloud-native autonomic elements). The latter component is tightly coupled with the type of management action. It acts as its helper class by taking care of capacity planning, then

marshalling the required resources and adequately preparing the execution environment.

- **Inference Layer** – This is the most sophisticated AMoCNA layer. It implements core AMoCNA features that are triggered by Control Layer. The Inference Layer consists of four components - *Data Correlation*, *Data Transformation*, *Data Reasoning* and *Application Profiler* which together form a chain. The order of the components is important as the output of each serves as input for the successive component. The first component of the chain - *Data Correlation*, retrieves observation data and correlates data from different sources. Such correlation is performed according to the dependencies between metrics. The second component - *Data Transformation*, transforms the observation data (which is already correlated) into a format accepted by the rule engine, namely into facts. The rule engine is contained within

the third component of the Inference Layer chain. This component, named *Data Reasoning*, is responsible for data interpretation, analysis and inference. These steps provide input for the final component in the chain, namely *Application Profiler*. *Application Profiler* aids the reconfiguration of CNAApp in accordance with the predefined policies, but also in a manner consistent with the contracted SLA. It produces management actions that are passed to the Management Layer.

- **Control Layer** – This is the topmost AMoCNA layer. It realizes core AMoCNA functions that are triggered by declarative management policies usually compliant with SLA. The depicted *user data flow* is, in fact, processed by the Control Layer. The Control Layer consists of two components - *SLA Governance* and *Policy Controller* that operate in a loop. This layer retrieves data from the Inference Layer (as processed observation data) and, following a series of checks and processing, sends it in the form of a dataflow back to the Inference Layer, although to different components - namely, to the rule engine wrapper. The *Policy Controller* drives the overall AMoCNA processes, based on user-declared management policies. The *SLA Governance* component ensures compatibility between the contracted SLA and the enforced management actions.

AMoCNA design conforms to containerization rules present in Cloud-native environments. As depicted in Fig. 3, containers which comprise AMoCNA and the CNAApp download their images from a common container registry. The collection of containers includes two types of AMoCNA microservices. The primary one, the management policy microservice, is available as a singleton and governs the whole execution environment. The second type, autonomic element microservices, are composed of independent, managed elements that in turn together comprise the CNAApp's execution environment. The mapping between a microservice and the Cloud-native autonomic element is 1:1.

In Fig. 3 it is depicted that AMoCNA microservices manage one CNAApp. This is compliant with Fig. 2 where it is depicted that every Cloud-native Autonomic Manager corresponds to one Cloud-native Execution Environment.

Nonetheless, we do not see any obstacles for AMoCNA simultaneous management of more CNAApps.

It has to be noted that each layer is backed up and shadowed by additional layers. These are equivalent to the highlighted corresponding layer, but represent different Cloud-native autonomic elements composing the AMoCNA framework. The internal structure of each element is similar, hence the components which exist in each layer are simply replicated for every Cloud-native autonomic element, and can be treated as existing in one single layer. For reasons of clarity they are represented in the background. In particular, there can be one single autonomic element microservice in any given AMoCNA realization. This assumption is reflected by the next phase of implementation (Section 5).

The itemization provided on page 3 highlights the mandatory properties taken into account while designing the AMoCNA framework. The *Related work* section mentions the Service Mesh architecture as an optional approach to management of CNAApps. However, comparing both approaches (Table 1) reveals that they apply to very different scopes and aspects.

The ability to manage any application is indispensable. The autonomic nature of management is a virtue of modern applications. Depending on the required precision, the proposed solution can be general and cover all layers of the CNAApp stack, or application-oriented, covering only the topmost Application layer. The next compared feature involves the motivation underpinning both architectures. The primary intention of the Service Mesh pattern is to intercept intercommunication between microservices in a CNAApp scope. This results in the Service Mesh being communication-driven. On the other hand, AMoCNA is SLA- and QoS-oriented (Table 1). The QoS metrics determine the established SLA which defines service parameters such as availability, performance, operation, etc. A complex management solution involves monitoring and statistics services. These services, supported by observability characteristics, provide insight into the execution environment. Apart from visualizing such parameters to the administrator, they serve as the basis for further adaptation and enforcement of actions in the execution environment. The principal aim of observability is to make the execution environment more understandable in various aspects. To do so, data is gathered and analyzed from multiple

Table 1 Comparison of the Service Mesh model and the proposed AMoCNA framework

Feature	Service Mesh	AMoCNA
Subject of operation	(Highest) Application Layer	All layers of CNApp stack
Driven by	Communication	QoS and SLA
Observability (how)	Limited	Detailed
Observability (what)	Global view of the CNApp	Global view of the CNApp and its execution environment
Operation style	Limited number of communication patterns	Policy driven, based on open framework
Control style	Limited	Full implementation of closed loop of feedback control
Instrumentation	Sidecar model	Not specified

sources. The resulting information provides accurate details about performance metrics related to infrastructure resource utilization and also about the status of all elements of the system. Measurements are collected and aggregated to yield historic and current views of the execution environment. The requirements of AMoCNA, itemized in Section 3, emphasize the importance of detailed observations that are crucial for maintaining the expected level of QoS. Observations are performed across all layers of the execution environment and are not limited to the CNApp boundaries. The availability of comprehensive knowledge means that in AMoCNA it becomes possible to declare any kind of management action. This comprehensive knowledge, both current and past, offers a wide spectrum of online reasoning and prediction capabilities.

Declarative management policies emphasize the end-user vision of the system, which is expressed in a rather simplistic manner. These policies usually constitute high-level demands and are seamlessly translated into low-level executable actions, which, in turn, can be mapped to the appropriate effectors. The effectively unlimited number of declarative management policies means that the AMoCNA framework is applicable to many use cases. This stands in contrast with Service Mesh, which proposes only a limited number of communication patterns (e.g. the circuit breaker pattern).

In addition, in AMoCNA, Cloud-native applications are capable of runtime reconfiguration according to the declared management policies. This feature helps proactively overcome SLA violations. Example included in [30] contains definition of a migration policy in case the CNApp's microservices consume too much resources and some other hosts are underutilized. The published reconfigurations are consumed during the

observation process, effectively closing the control loop. The novelty of this paper includes the proposal to apply the MRE-K control loop, itself based on the notion of the AS3 pattern [29]. The Inference and Control layers (as depicted in Fig. 3) of Cloud-native autonomic elements can not only be realized through rule engines, but for example with support of machine learning paradigm.

One advantage of the Service Mesh model is that it provides a detailed and unified definition of instrumentation methods in the form of a sidecar model. AMoCNA also acknowledges this layer, but does not impose any specific terms.

Thus, rather than being in competition with each other, both solutions are essentially complementary and implementing both seems to enable one to address more complex scenarios.

5 AMoCNA Implementation Aspects

The transformation process from PIM to Platform Specific Model (PSM) requires to select a technology stack used while implementing the AMoCNA framework. The decision has impact on architecture design and influence its features. It has been decided to use open source software. Their functionality do not stand out from commercial products, whereas they are publicly available and the usage for research purposes is non complicated [34]. A result of this selection is here provided.

Enhanced Cloud Computing (CC) [32] platform, constitutes an indispensable component of a Cloud-native environment in form of a Cloud-native platform. Cloud-native platforms accelerate development of Cloud-native applications. They abstract the underlying

infrastructure while provisioning computing resources as VMs, bare metal servers, etc., provisioning storage resources as databases, disks, etc. and also provisioning network as SDN controllers, load balancers, virtual switches, service discovery, etc. Cloud-native platforms center the application while provisioning and maintaining the parts of technology stack needed to achieve Cloud-native aspect. They accomplish this by undertaking tasks that must be done (e.g. creating, scheduling and provisioning orchestrated environments, providing high availability, fault tolerance and resiliency) but that are not directly related to the development of the application. Thorough analysis of CC platforms reveals the crucial pros and cons of diverse solutions and their support for Cloud-native. This results in choosing Openstack [28, 40] for that purpose. It significantly influence the provision and management of resources available for this research.

The proliferation of containers risen the need of their orchestration. Orchestrator comprises the execution environment of CNApPs (Fig. 1). Preferably, the concrete orchestrator is used, depending on its destiny. Taking this into account and the fact that this research execution environment regards diversity of CNApPs, the *Kubernetes* [25] has been chosen as a basis of this framework. It has a proper level of abstraction, and the most important, it is recommended by CNCF and is accepted by industry. It is worth mentioning that AMoCNA was also successfully tested with Docker Swarm.

This paper assumes that autonomic management of CNApP is realized by a set of Cloud-native autonomic elements. The MAPE-K loop (adequately, in this paper MRE-K loop) components are organized in a hierarchical fashion [27]. The number of Cloud-native autonomic elements (nodes in the hierarchy) depends on the level of required accuracy and performance, for example each component of a Cloud-native execution environment can be mapped to AMoCNA's autonomic elements in the ratio of 1:1. However, usually Cloud-native execution environment are in a whole represented by a single Cloud-native autonomic element. This means that there is a single MRE-K loop that in turn is accomplished with support of a single rule engine. *Drools* solutions serve this purpose. It is also used as a policy management tool, allowing their definition, not complicated authoring, maintenance and re-use.

Figure 4 depicts a development environment constructed for the research proof of concepts. The environment is actually a mapping of an abstract view of a CNApP execution environment (depicted in Fig. 1) into a concrete model. The Infrastructure Layer is centered around an OpenStack with multiple physical and virtual servers. The configuration assumes that controller node (runs most OpenStack services) and network node (runs neutron service) functionality is realized by a single VM and additional servers are supplied to augment the pool of compute resources. The Containerization Layer is orchestrated by a *Kubernetes* cluster. The AMoCNA testbed contained a cluster consisted of one master, and then 6 additional nodes has joined the cluster to carry the workload. The topmost layer of the CNApP execution environment is the Application Layer. This layer is mostly comprised of CNApP microservices.

Figure 4 presents a view from the selected technology stack perspective and shows the placement of AMoCNA components. Most of the components are realized through adopting entities existing in those technologies. This embracement requires little coding effort, assuring the adjustment and linkage between the technologies.

The components marked with red color, are the ones that need to be coded, to adjust the chosen technologies to AMoCNA's requirements. The Instrumentation Layer of AMoCNA framework (depicted in Fig. 3) is general, leaving to concrete PSM the implementation of its principles of operation. However in case of the present PSM, it is recommended to adhere monitoring metrics strictly to *Prometheus* directives. It is a well known and opensource standard. Firstly, the mapping between Prometheus metrics and Drools facts is done, and secondly, Management Controller dispatches the inferred actions to proper entities of AMoCNA. It should be stressed that more components need coding efforts (e.g. SLA Governance of Control Layer as depicted in Fig. 3), but the two aforementioned are the most important one to concatenate different technologies to adopt them in AMoCNA PSM transformation.

Such obvious choice from cutting-edge solutions of the technology stack, poses that present PSM is an enterprise-class ready framework. Most AMoCNA advantages are gained by leveraging the potential of selected technologies. Among, a never-ending list of

- Online visualization of current state of a Cloud-native application.
- Transparent inference over Cloud-native application execution environment,
- Declarative autonomic management of supervised Cloud-native application. The approach offers runtime SLA exchange and accordingly the execution environment reconfiguration.

6 Evaluation of AMoCNA

Figure 5 shows that OpenStack handles the creation of a Kubernetes cluster for CNApp and AMoCNA

¹Weaveworks (<https://www.weave.works>) and Container Solutions (<https://container-solutions.com>) have developed a demo microservices-based application - Sock Shop [48], which is open-source and available for free under Apache License 2.0. The application is used to carry out the experiments presented in this paper.

Fig. 5 Evaluation infrastructure hardware configuration Network and Service Layout

purposes (through the Magnum OpenStack API service). In this figure the AMoCNA framework validation infrastructure is comprised by a Kubernetes cluster with seven members. The cluster consisted of one master (labeled kube-master in the figure) and 7 nodes (labeled kube-node).

The purpose of the presented experiments is two fold. First, they estimate the overhead associated with infrastructure resources generated by deploying autonomic management solutions (Experiment 1) proposed in this paper. Secondly, they evaluate the performance of AMoCNA, i.e. the influence of policy enforcement by a rule engine (Experiment 2). Such as-

sumptions determine that the intervals of monitoring refresh time (60 seconds in this case) are not important. The experiments do not take the quality of input data into consideration.

Both experiments are conducted with a common case study. Its successive steps are as follows (the best way to track these steps is illustrated in Fig. 6, experiment 1): prior to timestamp 10:01 (marked on the horizontal axis) CNApp runs together with AMoCNA on 7 nodes (these nodes are listed in the legend of particular subfigures). Subsequently, pods realizing (i) translation of metrics, (ii) rule engine capabilities, (iii) support for declarative management policies,

Fig. 6 Overhead of the main compute resources measured while equipping Cloud-native application with autonomic management capabilities

and (iv) their execution are deleted (that means that all meaningful to AMoCNA management pods are deleted). Starting at 10:02, once the Cloud-native execution environment has settled down (note the peak on the “Network Utilization” graph, indicating increased network traffic between Kubernetes master and workers in order to set the new network topology), only CNApp and monitoring tools are still running.

Presented case study is a background to the following experiments.

6.1 Experiment 1: AMoCNA overhead

This experiment uses two methodologies for analyzing the overhead of the presented Cloud-native execution environment. Both measure the overhead

related to utilization of computing resources. Initially, employing the *USE*² method, the overall cluster utilization is measured from its nodes perspective. The second methodology provides performance information from the perspective of Kubernetes objects (mainly the Pods). Results of both methods are complementary, emphasizing the quality of proposed concepts.

Results of observations according to the first methodology are depicted in Fig. 6. In these observations measurements are captured in two steps (two different periods) reflecting consecutive cluster states, as explained below:

- Autonomic management of CNApp - measures resource consumption during CNApp operation with autonomic management enforcement enabled (state prior to timestamp 10:01).
- Monitoring - the CNApp and its execution environment is merely instrumented and the collected measurements do not trigger any management actions (state after timestamp 10:02).

A next step where monitoring is disabled may be desirable, but it is also pointless as information about the cluster needs to be retrieved anyway. The difference between both steps (i.e. the change in values plotted on the vertical axis after 10:02 compared to their initial values) is low, indicating that the proposed concepts of autonomic management in CNApp have already been efficiently implemented. The most notable difference concerns the cluster's memory utilization. We notice that memory utilization of worker nodes drops by 4% at this point.

The second methodology collects performance indicators from the perspective of Kubernetes objects. As shown in previous methodology, the scenario where all AMoCNA components are running gives more substantial results. Hence, in second methodology we put emphasis on period before 10:01. The collected information complements our understanding of the execution environment.

This methodology provides data only for the two aforementioned parameters (i.e. without network utilization). Kubernetes manages access to CPU and memory, allowing users to specify their requests and limits. The corresponding reference values are provided by considering all computing resources available to the cluster. Table 2 contains a summary

Table 2 The aggregation of consumption of computing resources from the perspective of Kubernetes objects

Computing resource	Utilization	Requests	Limits
CPU	4.626%	20.5%	21.5%
Memory	14.92%	6.40%	5.883%

of collected data. Results are similar (4.626% for CPU utilization and 14.92% for memory utilization) to those collected earlier (in Fig. 6 these are illustrated prior to timestamp 10:01 and correspond to the summarized Y-values depicted in the graphs). This confirms that the measurements gathered in both methodologies are correct. The quality of results, especially while turning off most Pods (from 10:02 onward), enables us to consider the AMoCNA prototype as an efficient realization of the proposed concepts.

6.2 Experiment 2: Performance of MRE-K loop concept

The intent behind the following experiment is to obtain better insight into the realization of the declared management policies and their processing performance.

An important concept proposed in this paper concerns the utilization of a rule engine for the purposes of autonomic management. This concept, referred to as the MRE-K loop, enables enforcement of declared management policies in a Cloud-native context. At its core lies the notion of using a rule engine as a means of enforcing management policies. This experiment reveals the effectiveness of such an approach.

To assess the AMoCNA response time it is necessary to measure (i) the overall delay imposed by translating metrics into facts (ii) the time required to insert metric facts into the rule engine's working memory, and (iii) the overall time of matching declared management policies against the metric facts. The results of the aforementioned measurements are discussed below.

In the present AMoCNA configuration, the overall delay imposed by translating metrics into facts is $1,1 \cdot 10^9$ ns. This is merely 1 second – however, hypothetical translation of all monitoring [42] metrics introduces a delay of $24,2 \cdot 10^9$ ns, equivalent to

²Utilization, Saturation and Errors

around 24 seconds. These results underscore the need to filter the metrics in advance.

The second step calls for timing the insertion of facts into the rule engine's working memory. Its results are depicted in Fig. 7.

The horizontal axis depicts the number of fact objects. The relation between facts and metrics illustrates the following example where e.g. metric `node_name` has 8 facts (there are 8 nodes in the cluster). The maximum value in Fig. 7 is for 258 metrics – above that number AMoCNA generates an exception indicating the need for memory tuning. The time required to insert metric facts (in seconds) is depicted on the vertical axis. The present AMoCNA configuration defines 32 metrics, yielding 1207 fact objects. Comparing these values to the graph reveals that the insertion time is below 2 seconds.

Finally, the last step in this experiment entirely relies on the performance of the selected rule engine implementation. Research in this area is out of scope of the present paper; however, some relevant data is captured. A discussion of best practices regarding construction of rules can be found in [15, 21]. In the presented experiment, with only two rules defined - as shown in Listings 1 and 2, the delay is just 0.8 seconds and rises very slowly as the number of rules increases.

To summarize our assessment of AMoCNA's response time or its imposed latency, it should be stated that in the present AMoCNA configuration it remains relatively low; definitely below 2 seconds. The results of performed tests conclusively prove that the AMoCNA response time is closely dependent on the number of collected metrics (which must remain limited) and also dependent on the number of defined rules. The latter value, in turn, depends on the contracted SLA. The rules should remain simple (ideally

```
import translator.metrics.ContainerMetric;
import translator.metrics.Metric;
import java.lang.Double;
import translator.SLA;
import pl.edu.agh.ki.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 1 Policy adjusting Pod's requests for CPU. First rule

```
import translator.metrics.ContainerMetric;
import java.lang.Double;
import pl.edu.agh.ki.amocna.CPU;
import pl.edu.agh.ki.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 2 Policy adjusting Pod's requests for CPU. Second rule

Fig. 7 Insertion of facts into the rule engine's working memory – time elapsed

without additional calculations) and concise, preferably involving single metrics [21].

Nevertheless, the presented experiments provide only a superficial estimate of the performance and overhead of AMoCNA. Further detailed studies are required.

7 Conclusions

The present paper addresses issues of autonomic management in the context of Cloud-native applications. To validate the proposed concepts, the AMoCNA framework was developed and its prototype implemented.

The proposed autonomic management that features Cloud-native autonomic elements seems to be the right choice. These elements we propose to realize through MRE-K loop approach that is with support of rule engine.

The presented research represents an enhancement of existing Cloud-native environments with observability that supports autonomic management using a declarative policy-based approach. The use of cutting-edge techniques across different parts of the proposed solution, especially within the scope of its foundation and observability part, enriches the Cloud-native application's capability for dynamic reconfiguration at runtime, depending on current state. The runtime reconfiguration is accomplished by declarative policy-based management that enables, among others, the execution of management tasks submitted by ordinary users with deference to resource QoS constraints (such as CPU or memory capacity).

The AMoCNA framework underscores the importance of autonomic management of Cloud-native application and enables mapping high-level demands into low-level effectors. Thereby it emphasizes the end-user vision of the environment.

Naturally, the proposed concepts, are not complete and can be improved in many aspects. Some new, interesting challenges involving autonomic management of CNApPs are being addressed in order to integrate AMoCNA with the service mesh model. There are many further directions of development (e.g. security issues) worth looking into. However, all these directions share a common goal, namely improving the quality of Cloud-native application management.

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.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

References

- Adamczyk, J., Chojnacki, R., Jarzab, M., Zielinski, K.: Rule engine based lightweight framework for adaptive and autonomic computing. In: Bubak, M., van Albada, G., Dongarra, J., Sloot, P. (eds.) *Proceedings of International Conference on Computational Science – ICCS 2008*, Lecture Notes in Computer Science, vol. 5101. Springer (2008)
- Agrawal, D., Calo, S., won Lee, K., Lobo, J., Verma, D.: *Policy Technologies for Self-Managing Systems*, 1 edn. IBM Press (2008)
- Caballer, M., Zala, S., García, L.A., Moltó, G., Fernández, O.P., Velten, M.: Orchestrating complex application architectures in heterogeneous clouds. *Journal of Grid Computing* (2018)
- Calinescu, R.: Resource-Definition Policies for Autonomic Computing. In: *Proceedings of 5Th International Conference on Autonomic and Autonomous Systems (ICAS'09)*. IEEE Computer Society Press (2009)
- Cisco Identity Services Engine. <https://www.cisco.com/c/en/us/products/security/identity-services-engine/index.html>. Last seen on April, 2020
- Cloud Native LandScape. <https://github.com/cncf/landscape>. Last seen on March, 2019
- Cuomo, A., Di Modica, G., Distefano, S., Puliafito, A., Rak, M., Tomarchio, O., Venticinque, S., Umberto, V.: An sla-based broker for cloud infrastructures. *J. Grid Comput.* **11**, 1–25 (2012). <https://doi.org/10.1007/s10723-012-9241-4>
- Debroy, V., Miller, S., Brimble, L.: Building lean continuous integration and delivery pipelines by applying devops principles: a case study at varidesk. In: *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 851–856. ESEC/FSE 2018. Association for Computing Machinery, New York. <https://doi.org/10.1145/3236024.3275528> (2018)
- Dobaj, J., Schuss, M., Krisper, M., Boano, C.A., Macher, G.: Dependable mesh networking patterns. In: *Proceedings*

- of the 24th European Conference on Pattern Languages of Programs, EuroPlop '19. Association for Computing Machinery, New York. <https://doi.org/10.1145/3361149.3361174> (2019)
10. Docker site. <https://www.docker.com>. Last seen on November, 2019
 11. Universal Control Plane overview. <https://docs.docker.com/ee/ucp/>. Last seen on April 2020
 12. Dynatrace: Monitor, optimize, and scale every app, in any cloud. <https://www.dynatrace.com> Last seen on April 2020
 13. Envoy is an open source edge and service proxy, designed for Cloud-native Applications. <https://www.envoyproxy.io>. Last seen on April 2020
 14. Frankel, D.: Model driven architecture: Applying MDA to enterprise computing. Wiley, USA (2002)
 15. Friedman, E.: Jess in Action: Rule-based Systems in Java. Manning Publications Co., Greenwich (2003)
 16. Ganek, A., Corbi, T.: The dawning of the autonomic computing era. *IBM Syst. J.* **42**(1), 5–18 (2003). <https://doi.org/10.1147/sj.421.0005>
 17. Géhberger, D., Mátray, P., Németh, G.: Data-Driven Monitoring for Cloud Compute Systems. In: 2016 IEEE/ACM 9Th International Conference on Utility and Cloud Computing (UCC), pp. 128–137 (2016)
 18. Gutierrez-Aguado, J., Alcaraz Calero, J.M., Diaz Villanueva, W.: Iaasmon: Monitoring architecture for public cloud computing data centers. *J. Grid Comput.* **14**(2), 283–297 (2016). <https://doi.org/10.1007/s10723-015-9357-4>
 19. Haff, G., Henry, W.: From Pots and Vats to Programs and Apps: How Software Learned to Package Itself, 1St Edn. CreateSpace Independent Publishing Platform, USA (2017)
 20. Huebscher, M.C., McCann, J.A.: 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>
 21. IBM Developer: IBM Operational Decision Manager On Cloud Trial. <https://developer.ibm.com/odm/> Last seen on February, 2019
 22. Inc., S.G.: Monitoring and Logging for Docker Datacenter. Reference Architecture. <https://sematext.com/publications/reference-architecture-monitoring-and-logging-for-docker-datacenter.pdf> (2016)
 23. Connect, secure, control, and observe services. <https://istio.io>. Last seen on April 2020
 24. Jarzab, M., Kosiński, J., Kosińska, J., Zieliński, K., Zieliński, S.: Virtualized Infrastructure for SOA, pp. 211–234. Poznań University of Economics Press (2010)
 25. Kubernetes site. <https://kubernetes.io>. Last seen on April, 2020
 26. Kalman, R.: On the general theory of control systems. *IRE Trans. Autom. Control* **4**, 110–110 (1960)
 27. 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>
 28. Khedher, O., Chowdhury, C.: Mastering OpenStack. Packt Publishing. <https://books.google.pl/books?id=G0IwDwAAQBAJ> (2017)
 29. Kosinski, J., Szymacha, R., Szydło, T., Zielinski, K., Kosinska, J., Jarzab, M.: Adaptive SOA solution stack. *IEEE Trans. Serv. Comput.* **5**(2), 149–163 (2012). <https://doi.org/10.1109/TSC.2011.8>
 30. Kosińska, J.: Autonomic Management Framework for Cloud-native Applications. Ph.D. thesis, AGH-UST (2019)
 31. Kovács, J.: Supporting programmable autoscaling rules for containers and virtual machines on clouds. *J. Grid Comput.* **17**(4), 813–829 (2019). <https://doi.org/10.1007/s10723-019-09488-w>
 32. Kratzke, N.: A brief history of cloud application architectures. *Appl. Sci.* **8**(8), 1368 (2018). <https://doi.org/10.3390/app8081368>
 33. Kratzke, N., Peinl, R.: Clouds - A Cloud-native Application Reference Model for Enterprise Architects coRR. arXiv:1709.04883 (2017)
 34. Lindberg, V.: Intellectual Property and Open Source: A Practical Guide to Protecting Code, 1 edn. O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, pp. 95472 (2008)
 35. Ultralight, security-first service mesh for Kubernetes. <https://linkerd.io>. Last seen on April 2020
 36. Mesos site. <http://mesos.apache.org>. Last seen on April, 2019
 37. Broadband System Management. https://www.motorolasolutions.com/en_us/products/lte-broadband-systems/operations-support-systems/system-management.html. Last seen on April, 2020
 38. Apply intelligence across all your data with SQL Server 2019. <https://www.microsoft.com/en-us/sql-server>. Last seen on April, 2020
 39. New Relic APM solution. <https://newrelic.com>. Last seen on April 2020
 40. Openstack site. <https://www.openstack.org/>. Last seen on April, 2019
 41. 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>
 42. Prometheus site: From metrics to insight. <https://prometheus.io> Last seen on April, 2020
 43. Röblitz, T., Schintke, F., Reinefeld, A., Barring, O., Lopez, M., Cancio, G., Chapeland, S., Chouikh, K., Cons, L., Poznanski, P., Defert, P., Iven, J., Kleinwort, T., Panzer-Steindel, B., Polok, J., Rafflin, C., Silverman, A., Smith, T., Eldik, J., Leiva, R.: Autonomic management of large clusters and their integration into the grid. *J. Grid Comput.* **2**, 247–260 (2004). <https://doi.org/10.1007/s10723-004-7647-3>
 44. 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. <https://books.google.pl/books?id=XHeoSgAACAAJ> (2004)
 45. Singh, S., Chana, I.: Qos-aware Autonomic Resource Management in Cloud Computing: A Systematic Review. *ACM Comput. Surv.* **48**(3), 42:1–42:46 (2015). <https://doi.org/10.1145/2843889>
 46. Sloman, M.: Policy driven management for distributed systems. *J. Netw. Syst. Manag.* **2**(4), 333–360 (1994)
 47. Sloman, M., Lobo, J., Lupu, E. (eds.): Policies for Distributed Systems and Networks, International Workshop, POLICY 2001 Bristol, Proceedings, Lecture Notes in Computer Science, vol. 1995. Springer. <https://doi.org/10.1007/3-540-44569-2> (2001)

48. Sock Shop - A Microservices Demo Application. <https://microservices-demo.github.io>. Last seen on April 2020
 49. Toffetti, G., Brunner, S., Blöchliger, M., Spillner, J., Bohnert, T.M.: Self-managing cloud-native applications: design, implementation, and experience. *Fut. Gener. Comp. Syst.* **72**, 165–179 (2017)
 50. Verreydt, S., Beni, E.H., Truyen, E., Lagaisse, B., Joosen, W.: Leveraging kubernetes for adaptive and cost-efficient resource management. In: *Proceedings of the 5th International Workshop on Container Technologies and Container Clouds, WOC '19*, pp. 37–42. Association for Computing Machinery, New York. <https://doi.org/10.1145/3366615.3368357> (2019)
 51. Wilde, N., Simmons, S., Pressel, M., Vandeville, J.: Understanding features in soa: Some experiences from distributed systems. In: *Proceedings of the 2nd International Workshop on Systems Development in SOA Environments, SDSOA '08*, pp. 59–62. Association for Computing Machinery, New York. <https://doi.org/10.1145/1370916.1370931> (2008)
- Publisher's Note** Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.