A MAPE-K Approach to Autonomic Microservices

Antonio Bucchiarone
Fondazione Bruno Kessler
Trento, Italy
bucchiarone@fbk.eu

Claudio Guidi
italianaSoftware s.r.l.
Imola, Italy
cguidi@italianasoftware.com

Ivan Lanese
University of Bologna/INRIA
Bologna, Italy
ivan.lanese@gmail.com

Nelly Bencomo
Durham University
Durham, United Kingdom
nelly.bencomo@durham.ac.uk

Josef Spillner
Zurich University of Applied Sciences
Zurich, Switzerland
josef.spillner@zhaw.ch

***Abstract*—Microservices are an emerging architectural style advocating for small loosely-coupled services in order to maximize scalability and adaptability. In order to help IT personnel, adaptability can be put (completely or partially) under the responsibility of the system using autonomic techniques, e.g., underpinned by a MAPE-K control loop. This paper discusses possible trade-offs, challenges and support techniques for software architects involved in building autonomic microservice-based systems.**

***Index Terms*—Autonomic Computing, Microservices, MAPE-K control loop**

I. Introduction

Software architectures move towards an *intelligent no-ops era* with functionality encapsulated in networked containers [1], [2]. This has severe impact on the way software is conceived. Since computation is a good that can be obtained and released on demand, it must be requested only when needed. For cost reasons, software must thus be designed to follow the demand-and-load curves. New architectural approaches emerged for addressing such an issue: microservices and serverless architectures are today the main approaches for dealing with it [3]. Continuous integration practices [4] deliver as fast as possible new versions and new functionalities, layering upon a smart automatized infrastructure, which is able to enforce compilations, quality checks, tests and deployment with minimum human intervention, if at all. The increasing complexity of modern software demands for new approaches to architectural design and system modeling. *Microservices* [3] is an architectural style originating from Service-Oriented Architectures and introduced to provide: (i) high scalability, (ii) technology and language independence, (iii) simple maintainability and (iv) simple update and redeployment due to loose coupling. Reasoning in terms of microservices involves finding at runtime the best architectural configuration to achieve the desired goals. As such, several areas of software engineering can underpin microservices [5], such as, service composition and orchestration, runtime architectural adaptation, versioning, and Infrastructure as/from Code.

The uncertain and dynamic context of the ecosystems of microservices calls for adaptation support similar to that provided by autonomic (a.k.a. self-*) systems [6] to be added into the architecture. The autonomic approach lets the operator describe at the high-level of abstraction the desired outcome and leaves to the system itself the tedious and error-prone work to find the actual architectural changes needed to reach such an outcome. A fully autonomic approach is not always possible or even desirable, hence, in most cases, autonomic capabilities complement manual interventions or directives from the IT personnel. Such a perspective raises important issues related to the interactions between the runtime environment and the microservices, because some of the autonomic actions can be provided only by negotiating with the environment. Consider a microservice that asks for being scaled to address a load increment. Such an activity is in charge to the runtime environment (e.g., Kubernetes), thus the microservice must ask the environment to provide another instance of itself. As a consequence, the protocols for negotiating these kinds of actions between the microservices and the environment should be rationalised.

There are several mechanisms which can be adopted towards autonomicity [6], [7]. They are generally underpinned by the fundamental ideas of feedback loops [8], which comprise the activities of Monitor, Analyse, Plan and Execute (MAPE). The decision making for adaptation is made according to trade-offs between positive and negative effects as consequences of the adaptation actions. In the general setting of autonomic systems, two implementation strategies have been discussed: adding an autonomic manager in charge of it or embedding the autonomic capabilities within the managed system. In the context of microservices also the runtime plays a role, hence more trade-offs emerge and need to be discussed. In all the cases, the entity in charge of the autonomic behaviour must acquire sufficient knowledge to take on the activities that are to be automated. The knowledge to recognize the need for adaptation and to automatically decide and perform the actions required needs to be maintained in a knowledge base. The approach above is known as the MAPE-K loop [9].

The main aim of the present paper is to discuss different alternative solutions and trade-offs related to the application of an autonomic approach based on the MAPE-K loop in microservice scenarios. More in detail, we will present a vision using a graphical representation to highlight which entity (microservices, environment or IT personnel) takes

responsibility of each MAPE-K phase, and also discuss the trade-offs related to different design choices.

II. Our Vision

We envision scenarios where microservice-based systems (MBS) are purposefully equipped with autonomic capabilities to manage issues such as self-{protection, healing, optimization, reconfiguration} and so on. This complements the microservice approach which automates aspects such as deployment and scalability, making a further step towards NoOps scenarios [10]. As discussed earlier, autonomicity can be obtained using the classical MAPE-K approach. The MAPE-K feedback control loop performs MAPE phases over a shared Knowledge [9]. The Monitoring (M) phase acquires data from the system and its environment. Analysis (A) of the monitored data involves activities such as filtering or transforming data, e.g. to reduce noise. Then, Planning (P) of future actions is done, keeping into account the result of the analysis as well as the knowledge of the system held in the model. Finally, the Execution (E) of the planned actions should be performed. This consists of changing the values on the actuators in the system according to the developed plan. A MAPE-K loop stores the Knowledge (K) required for decision-making in what is called the Knowledge Base (KB).

When integrating a MAPE-K loop in an MBS there are various options related to which part of the system performs the phases involved in the loop. Possibilities include having phases performed by the infrastructure (e.g., containers, cloud runtime, etc.), by each microservice in addition to its functionalities, by ad-hoc microservices, or by groups of microservices. While different decisions are suitable in different contexts, they come equipped with constraints and trade-offs. The contribution of this paper is the analysis and discussion of such trade-offs, aiming at providing guidelines for designers of autonomic microservice systems, taking into account benefits and issues that come with different design choices.

To categorize the possible approaches to introduce autonomic behaviour in MBSs, we introduce a graphical notation to identify the responsibilities of intervention for the different phases of the MAPE-K loop. In Figure 1a, we represent the intervention responsibilities by actor (humans, environment, and microservices) as concentric circles:

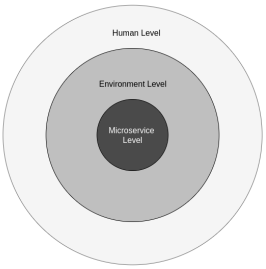
- the external area represents *human responsibility*, namely the fact that the IT personnel is in charge of the corresponding activity; of course if all the activities are under human responsibility then the system is not autonomic at all;
- the inner area represents *microservice responsibility*: either dedicated microservices or each microservice as part of its capability take care of the activity;
- the area in the center represents the responsibility of the *environment*. This case may denote that the activity is under the responsibility of the infrastructure, or more in general of any automatic entity not part of the considered MBS.

Since the MAPE-K phases form a loop, we represent them as different sectors of the same circle as reported in Figure 1b. We do not explicitly represent the KB; in many cases it is under the responsibility of the entity that takes care of the planning, since planning makes extensive use of the KB. Other options are also possible, however we think that an explicit representation of the KB would clutter the simple graphical representation presented here. We combine the two diagrams to describe an approach to autonomic microservices as a scale diagram in Figure 1c. Essentially, each area of the figure represents whether the actor corresponding to the circle takes some responsibility for the activity corresponding to the MAPE sector: if the sector is white then it takes no responsibility; if it is (arbitrarily) coloured then the actor has some responsibility. On the one hand, note that for each MAPE sector at least one segment (i.e., portion of circle in the sector) needs to be coloured since at least one actor must take the responsibility for the activity. On the other hand, more than one segment can be coloured in a sector, since multiple actors can cooperate in taking responsibility for the activity. Systems more coloured towards the center are more autonomic than systems more coloured at the border. Therefore, we argue that such a diagram helps architects, developers and sysadmins to grasp at the glance the degree of autonomicity of a running microservice system, as well as the main responsibilities involved in the chosen approach. Further, the same notation can also be used at the level of system design to highlight requirements.

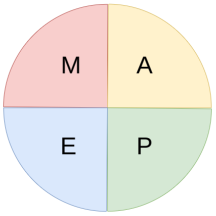
1) Examples: We introduce examples of our notation and trade-offs that we will discuss more deeply later on. Figure 1d highlights 3 special cases about the degree of autonomicity, with respect to the actors. On the left side, all segments of the external sector are coloured. Thus, all MAPE-K phases are performed by human actors, hence there is no autonomic behaviour. In the center, all the phases are automated by the environment. In this case the MBS is not really autonomic, since microservices are oblivious of the self-* behaviour. On the right side, all phases are managed by the microservices themselves. This is a fully autonomic MBS. Such an approach has the positive side of providing maximum flexibility, since microservices can react to changing conditions autonomously, and they can also be easily relocated since they do not rely on the environment for activities. Nevertheless such a scenario is difficult to obtain, since, microservices may not have access to useful information such as CPU and memory load, which is normally under the control of the infrastructure, and which may be useful to steer reconfiguration in the correct direction. Similarly, microservices may not have control on the actuators able to change policies related to infrastructure behaviour.

A solution would be found in a middle ground, represented by a more realistic scenario related to self-scaling (in Figure 1e). Also called auto-scaling, it is an attractive feature of microservice architectures, which allow them to autonomously react to changes according to the working load. To illustrate the reaction process, we exemplify it with Kubernetes [11], a widely used industry standard that supports scaling at the infrastructure level. Referring to a Kubernetes-based scenario, auto-scaling would be represented in our scale diagram as an environment-based scenario, where:

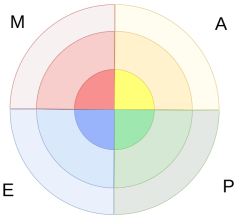
- **Monitoring** is performed by Kubernetes only, which continuously collects metrics from the deployed pods.



(a) Levels of responsibilities in autonomic microservices.

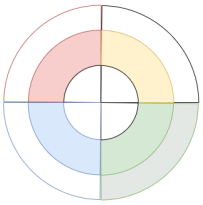


(b) MAPE phases as sectors.

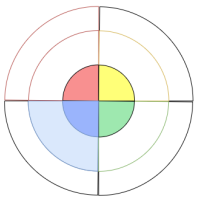


(c) Autonomic microservice landscape.

(d) Special cases about the degree of autonomicity.



(e) Self-scaling.



(f) Proof of concept.

Fig. 1: A MAPE-K concept and notation vision towards Autonomic Microservices.

- **Analysis** is also performed by Kubernetes, which extracts statistics from the metrics.
- **Planning** is performed by Kubernetes as well, depending on the initial customization provided by the sysadmin. Here we highlight as well the sector of human responsibility as planning may require a previous configuration performed by a human actor. Only when the planning is always applied without customization, planning can be put in charge exclusively to the environment.
- **Execution** is provided by Kubernetes, which autonomously scales the components and takes care of load balancing.

2) General considerations: Having a system such as the one described above, where all the phases of the MAPE-K loop are in charge of the environment would lead to an autonomic infrastructure managing non-autonomic microservices. This would offer the disadvantage of increasing platform lock-in, i.e. it would be difficult to move the MBS across different platforms since autonomic capabilities would be lost, unless the new platform offers support for the MAPE-K phases as well. Another general observation is that if both microservices and the environment concur to the same or to different activities of the MAPE-K loop then an interface allowing the interaction among them is needed. Such an interface allows them to agree or negotiate about the results of the various phases of the MAPE-K loop. If such an interface is not standardized, then platform lock-in will indeed be more severe. Another aspect to take into account is that communication among microservices and between microservices and the environment is frequently asynchronous, hence care is needed to ensure coherence of their information and behavior. Finally, when phases are in charge of microservices, different implementations are possible. Indeed, there could be sidecar microservices which play some specific roles like monitoring or planning, or each microservice could take care of all the aspects. The first possibility will create centralization, which is against the microservice philosophy, yet the second option would duplicate efforts and increase the need for coordination.

3) Trade-offs for each phase: We describe in detail each phase of the MAPE-K loop, discussing the involved trade-offs as well as possible choices and their consequences. Consider first the **monitoring phase** which is in charge of retrieving information about the microservice(s) and their hosting environment. We argue that the microservices are in the best position to monitor themselves, since they have the knowledge about their structure and internals, while the environment does not. Microservices can, e.g., inspect useful information such as the kind and amount of requests they receive, or which functionalities are used. The infrastructure instead controls the environment where the microservices run, hence it is in the best position to monitor global metrics such as the percentage of CPU used. Only an interaction between the microservices and the environment can provide a complete view of the status of the system. Of course, the discussion above calls for a standardized interface to weaken vendor lock-in.

The **analysis phase** is in between Monitoring and Planning, hence it can be done closer to either phase, in between, or in any combination of the above. A single microservice monitoring some metric can directly filter the data by taking averages over time using a sliding window and send the result to the infrastructure. The infrastructure can take the average across microservices and update the KB accordingly. Asynchronous communications should be taken into account, e.g., analysis filters out values which are obsolete due to delays in communication or to microservices with excessive load.

The **planning phase** ideally would be done in a centralized way, that is by one entity only, i.e. a dedicated component of the environment or a dedicated system of microservices. The chosen entity can take all the information available and establish a suitable plan. This is also the place where the **knowledge base KB**, used for planning, should be kept. However, such a centralized approach may be at odds with the desired distributed and loosely-coupled nature of MBS. A main challenge here is how to distribute the planning as well as the KB (allowing, e.g., for scalability and robustness) while preserving the quality of the developed plans. Compositional approaches are possible, where each microservice subsystem takes care of local planning, and global coordination ensures compatibility and synergy among local plans. The global plan could even result as an emergent property of the system.

The **execution phase** aims at actuating on various settings and configurations of the system and of its environment. As before, microservices are in the best position to act internally, e.g., by changing the used algorithm, while the infrastructure is in a better position for more global actions, such as replicating the microservices or changing the resource provision.

III. Proof of concept summary

A proof of concept about how an autonomic microservice could work has been reported [12]. A functionality is originally provided by a single microservice μ. Then, a load increment and a corresponding slow down in the response time are simulated on its listening endpoints. The microservice, that is autonomously able to detect a deterioration of the response time of its operations, then invokes the environment to scale up one of its cores μ into the current infrastructure. After some delay, the load is simulated to decrease, thus improving the response time, resulting in μ asking for scaling down the extra instances. In this example, the infrastructure transforms just the requests from the autonomic microservice into operational activities on the underlying Docker layer, which is able to provide new containers inside the infrastructure. This example neatly highlights, even if in a minimal and raw setting, several of the points discussed in the previous sections:

- μ implements a simple monitor of its operations, particularly it collects the response time average. Thus phase M is under the responsibility of the microservice;
- μ detects a decrease in response time due to high load. Thus phase A is under the responsibility of the microservice;
- μ decides when to ask for a new instance for scaling the load. Thus phase P is under the responsibility of the microservice;
- μ negotiates with the environment the release of a new instance. Thus, phase E is under shared responsibility of the microservice and the environment.
- μ holds all the knowledge about the component to be deployed. The infrastructure is not aware of the new component to be deployed neither it is aware of its container image, which is built at runtime.

Figure 1f depicts the scale diagram of the proof of concept described above. Here the microservice is close to be fully autonomic. Indeed, only the execution phase is performed together with the infrastructure. The proof of concept has been realized using the Jolie programming language [13], where microservices can be deployed either together into a unique monolith or in a distributed manner. The autonomic microservice can fragment itself and promoting one of its internal components to become a scalable microservice by sending its definition to the environment. Such an aspect may not be easy to implement when using other more mainstream technologies. In general, building autonomic capabilities into microservices may not be easy, however the use of dedicated languages helps [14].

References

[1] Q. Zhang, L. Liu, C. Pu, Q. Dou, L. Wu, and W. Zhou, "A comparative study of containers and virtual machines in big data environment," in *11th IEEE Int. Conf. on Cloud Computing, CLOUD 2018, San Francisco, CA, USA*. IEEE Computer Society, 2018, pp. 178–185.

[2] C. Pahl, A. Brogi, J. Soldani, and P. Jamshidi, "Cloud container technologies: A state-of-the-art review," *IEEE Transactions on Cloud Computing*, vol. 7, no. 3, pp. 677–692, 2019.

[3] A. Bucchiarone, N. Dragoni, S. Dustdar, P. Lago, M. Mazzara, V. Rivera, and A. Sadovykh, Eds., *Microservices, Science and Engineering*. Springer, 2020.

[4] M. Di Penta, "Understanding and improving continuous integration and delivery practice using data from the wild," in *Proc. of the 13th Innovations in Software Engineering Conf. on Formerly Known as India Software Engineering Conference*, ser. ISEC 2020. New York, NY, USA: ACM, 2020.

[5] S. Hassan, R. Bahsoon, and R. Kazman, "Microservice transition and its granularity problem: A systematic mapping study," *Software: Practice and Experience*, vol. 50, 06 2020.

[6] B. Cheng *et al.*, "Software engineering for self-adaptive systems: A research roadmap," in *Software Engineering for Self-Adaptive Systems [outcome of a Dagstuhl Seminar]*, ser. Lecture Notes in Computer Science, vol. 5525. Springer, 2009, pp. 1–26.

[7] S.-W. Cheng, A.-C. Huang, D. Garlan, B. Schmerl, and P. Steenkiste, "Rainbow: Architecture-based self-adaptation with reusable infrastructure." vol. 37, 01 2004, pp. 276–277.

[8] Y. Brun, G. Di Marzo Serugendo, C. Gacek, H. Giese, H. Kienle, M. Litoiu, H. Müller, M. Pezzè, and M. Shaw, *Engineering Self-Adaptive Systems through Feedback Loops*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 48–70.

[9] "An architectural blueprint for autonomic computing," IBM, Tech. Rep., Jun. 2005.

[10] L. Leite, C. Rocha, F. Kon, D. Milojicic, and P. Meirelles, "A survey of devops concepts and challenges," *ACM Computing Surveys (CSUR)*, vol. 52, no. 6, pp. 1–35, 2019.

[11] F. Rossi, V. Cardellini, and F. L. Presti, "Hierarchical scaling of microservices in kubernetes," in *IEEE Int. Conf. on Autonomic Computing and Self-Organizing Systems, ACSOS 2020, Washington, DC, USA*. IEEE, 2020, pp. 28–37.

[12] C. Guidi, "Towards Autonomic Microservices, Proof of concept," https://github.com/jolie-storm/autonomic-microservices, 2020.

[13] "Jolie, the service-oriented programming language," https://jolie-lang.org.

[14] C. Guidi, I. Lanese, M. Mazzara, and F. Montesi, "Microservices: A language-based approach," in *Present and Ulterior Software Engineering*, M. Mazzara and B. Meyer, Eds. Springer, 2017, pp. 217–225.