A MAPE-K Approach to Autonomic Microservices[*]

Antonio Bucchiarone[3], Claudio Guidi[2], Ivan Lanese[1], Nelly Bencomo[4], and Josef Spillner[5]

[1] University of Bologna-INRIA, Italy, ivan.lanese@unibo.it
[2] italianaSoftware, Imola, Italy, cguidi@italianasoftware.com
[3] Fondazione Bruno Kessler, Italy, bucchiarone@fbk.eu
[4] Aston University, Birmigham, UK, nelly.bencomo@durham.ac.uk
[5] Zurich University of Applied Sciences, 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.

1 Introduction

Software architectures continue to move towards a *ubiquitously networked era*. Containerization technologies, together with the rapid spread of Cloud Computing, have changed the way a computational resource is perceived, no more a hardware machine, but just a *box* able to execute software [14]. The box then runs as container [15], which is a single package of software abstracted away from the host operating system, and hence, it can run across any platform or cloud.

This industrial fact has severe impact on the way software is and will continue to be conceived. Since computation is seen as a good that can be obtained and released fast and on demand, it is quite obvious that it must be requested only when needed. Costs follow the usage, indeed. For this reason, software is requested to be scalable, in other words it must be designed and implemented to be flexible enough 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 [4]. Continuous integration and delivery practices changed the way software production chains are conceived [7]. The central idea is to deliver as fast as possible new versions and new functionalities, layering upon an automatized infrastructure, which is able to enforce compilations, quality checks, tests and deployment with minimum human intervention, if at all. The combination of all these forces will bring us rapidly to the mentioned ubiquitously networked era, an age where software components would be easily deployed and integrated in both private and public clouds. The main objective is the reduction of costs and the increase of velocity while substantially reducing the need of human intervention. Every aspect of the software delivery should be automatized, by leaving to the human intervention only the intelligence support to provide and write the logic.

The increasing complexity of modern software, which requires to be flexible and rapidly deployable, demands for new approaches to architectural design and system modeling. Innovative

[*]This paper has been already accepted in the poster track session of 19TH IEEE International Conference on Software Architecture (ICSA 2022)

engineering is always looking for adequate instruments to model and verify software systems and support developers all along the development process to deploy correct software. *Microservices* [4] is an architectural style originating from Service-Oriented Architectures (SOAs) 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 among the composing services. Reasoning in terms of microservices is essentially a dynamic architectural decision problem to achieve the desired goals. As such, several areas of software engineering can underpin microservices [10], such as, among others, service composition and orchestration, runtime architectural adaptation, versioning, and Infrastructure as/from Code (IaC/IfC).

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 [5] to be added into the architecture. The autonomic approach allows the operator to provide high-level directives on the desired outcome of the adaptation and leaving to the system itself the tedious and error-prone work to transform it into actual architectural changes. A fully autonomic approach is not always possible or even desirable, hence, in most of the cases, autonomic capabilities complement manual interventions or directives from the IT personnel.

Such a perspective raises important issues related to the interactions between the running environment and the microservices, because some of the autonomic actions can be provided only by negotiating with the environment. As an example, let us consider a microservice that asks for being scaled to address a load increment. Such an activity is in charge to the running environment (Docker, Kubernetes, etc), thus the microservice must ask to the environment to provide another instance of itself. Thus, 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 [5, 6]. These solutions are generally underpinned by the fundamental ideas of feedback loops [3], 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 environment, including, e.g., the Cloud or the container where the microservices are running, can play a role, hence more trade-offs emerge and therefore they 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 [2].

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. We also highlight open issues and research directions that need to be tackled to transform autonomic microservices into viable industrial practice.

2 Our vision

We envision scenarios where microservice-based systems (MBS) are purposefully equipped with autonomic capabilities to manage issues such as self-protection, self-healing, self-optimization, self-reconfiguration and so on. This complements the microservice approach, which aims at automating aspects such as deployment and scalability, making a further step towards NoOps scenarios [11].

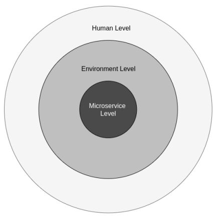
As discussed earlier, autonomicity can be obtained using the classical MAPE-K approach. The MAPE-K feedback control loop performs Monitor-Analyze-Plan-Execute phases over a shared Knowledge [2]. The Monitoring (M) phase acquires data from the system and its environment. Analysis (A) of the monitored data involves activities such as filtering or transforming data, e.g. to reduce noise or to put them in a form suitable for elaboration. 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 actuator(s) 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 different phases involved in the loop. Possibilities include having phases performed by the infrastructure (e.g., Containers, the Cloud, 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 various constraints and trade-offs. A main 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.

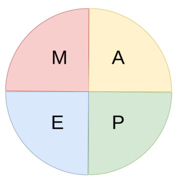
To categorize the possible approaches to introduce the envisioned autonomic behaviours in MBSs, we introduce a graphical notation to identify the responsibilities of intervention for the different phases of the MAPE-K loop. In particular, in Figure 1a, we represent the intervention responsibilities by actor (i.e. for the actors: 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 takes 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 on which the microservices are running, e.g. a Cloud platform or a container, or more in general of any automatic entity not part of the considered microservice system.

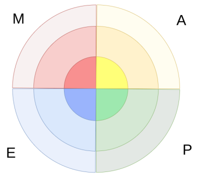
Since the MAPE-K phases form a loop, we represent them as different sectors of the same circle as reported in Figure 1b. Note that we do not explicitly represent the knowledge base: in many cases it is under the responsibility of the entity that takes care of the planning, since planning makes extensive use of the knowledge base. Other options are also possible, however we think that an explicit representation of the knowledge base would clutter the simple graphical representation presented here. Therefore, we leave further analysis of this issue for future work. 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.



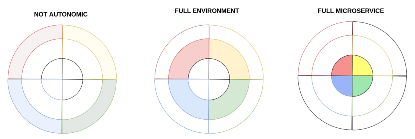
(a) Levels of responsibilities in autonomic microservices.



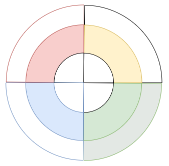
(b) MAPE phases as sectors.



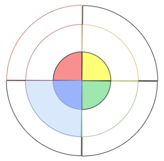
(c) Autonomic microservice landscape.



(d) Special cases about the degree of autonomicity.



(e) Self-scaling.



(f) Proof of concept.

Figure 1: A Vision towards Autonomic Microservices.

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.

Let us note that systems more coloured towards the center are more autonomic than systems more coloured at the border. Therefore, we argue that such a diagram can help 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 the main requirements.

Figure 1f depicts the scale diagram of the proof of concept described above. Here the microservice is close to be a fully autonomic microservice. Indeed, only the execution phase is performed together with the infrastructure.

The proof of concept above has been realized using the Jolie programming language [1], 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 for microservices. In general, building autonomic capabilities into microservices may not be easy, however the use of dedicated languages can help [9].

3 Roadmap

Service delivery is inherently characterised by differences in interest by engineers, providers and consumers. We identify below five *autonomic microservice challenges* that must be overcome in this context to be able to implement the vision previously outlined in business practice.

1. **Standardised discoverable APIs**. Interaction between the services and the infrastructure should take place via well-defined APIs. To allow deployment on multiple infrastructures and to reduce vendor-lockin such APIs should be standardized, so that the same interface would be available on different platforms. Ideally, custom implementations could be provided whenever the infrastructure does not cover them. A decision to use the MAPE-K control loop to approach autonomicity could give guidelines on how to define such a standardized API, which should reasonably involve functionalities for each of the 4 phases and for sharing the knowledge base K.

   This conveys awareness on the situation - whether self-managed changes are allowed and to which degree, versus immutable infrastructure requirements, from the provider to the hosted services. Recent de-facto standards like the Open Service Broker API [13] have demonstrated that a cross-provider discovery approach is possible.

2. **Intent-based specification of autonomicity**. Rather than specifying imperatively all technical instructions on which change to perform under which condition, high-level intents or rules (i.e. described instead of prescribed) that keep into account both technical aspects (e.g., throughput) and economic concerns (e.g., price of resources) are to be preferred. This allows for stating objectives such as "any change is allowed as long as the cost of service operation does not exceed a maximum threshold". As usual, moving from an imperative to a declarative approach makes the description of the autonomic behaviour more flexible and easy to understand; yet the translation of such a specification to code executable on top of the available API may become more complex.

3. **Multi-cloud support**. If autonomous services are meant to automate human logic on what to run where, this includes necessarily decisions on when to activate additional providers or migrate between providers. For this to be technically feasible then, shared knowledge on what providers exist and what as-a-service models they support with which non-functional properties need to be available.

4. **Auto-versioning**. If a configuration change happens at runtime, as advocated in the autonomic approach, the previous configuration must continue to be available to ensure zero downtime. The concept of versioning, which is used inconsistently across cloud providers and frameworks [8, 12], needs to become a standard feature.

5. **Ecosystem of MAPE-K logic**. Many problems that are addressed by autonomy are shared across application domains. Engineers would benefit from accessing libraries and repositories containing custom logic for the analysis of monitoring data and the planning of next steps. This logic could itself be offered in the form of microservices, such as stateless functions delivering analytical results and planning decisions, for technological coherence and increased chance of adoption.

References

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

[2] An architectural blueprint for autonomic computing. Technical report, IBM, June 2005.

[3] Yuriy Brun, Giovanna Di Marzo Serugendo, Cristina Gacek, Holger Giese, Holger Kienle, Marin Litoiu, Hausi Müller, Mauro Pezzè, and Mary Shaw. *Engineering Self-Adaptive Systems through Feedback Loops*, pages 48–70. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[4] Antonio Bucchiarone, Nicola Dragoni, Schahram Dustdar, Patricia Lago, Manuel Mazzara, Victor Rivera, and Andrey Sadovykh, editors. *Microservices, Science and Engineering*. Springer, 2020.

[5] Betty Cheng et al. Software engineering for self-adaptive systems: A research roadmap. In *Software Engineering for Self-Adaptive Systems [outcome of a Dagstuhl Seminar]*, volume 5525 of *Lecture Notes in Computer Science*, pages 1–26. Springer, 2009.

[6] Shang-Wen Cheng, An-Cheng Huang, David Garlan, Bradley Schmerl, and Peter Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. volume 37, pages 276–277, 01 2004.

[7] Massimiliano 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*, ISEC 2020, New York, NY, USA, 2020. ACM.

[8] Sara Gholami, Alireza Goli, Cor-Paul Bezemer, and Hamzeh Khazaei. A framework for satisfying the performance requirements of containerized software systems through multi-versioning. In *Proc. of the Int. Conf. on Performance Engineering, ICPE '20*, page 150–160, NY, USA, 2020. ACM.

[9] Claudio Guidi, Ivan Lanese, Manuel Mazzara, and Fabrizio Montesi. Microservices: A language-based approach. In Manuel Mazzara and Bertrand Meyer, editors, *Present and Ulterior Software Engineering*, pages 217–225. Springer, 2017.

[10] Sara Hassan, Rami Bahsoon, and Rick Kazman. Microservice transition and its granularity problem: A systematic mapping study. *Software: Practice and Experience*, 50, 06 2020.

[11] Leonardo Leite, Carla Rocha, Fabio Kon, Dejan Milojicic, and Paulo Meirelles. A survey of devops concepts and challenges. *ACM Computing Surveys (CSUR)*, 52(6):1–35, 2019.

[12] L. Liu, X. He, Z. Tu, and Z. Wang. Mv4ms: A spring cloud based framework for the co-deployment of multi-version microservices. In *2020 IEEE Int. Conf. on Services Computing (SCC)*, pages 194–201, 2020.

[13] OSBA. Open Service Broker API. https://github.com/openservicebrokerapi/servicebroker/blob/master/spec.md, October 2020.

[14] C. Pahl, A. Brogi, J. Soldani, and P. Jamshidi. Cloud container technologies: A state-of-the-art review. *IEEE Transactions on Cloud Computing*, 7(3):677–692, 2019.

[15] Qi Zhang, Ling Liu, Calton Pu, Qiwei Dou, Liren Wu, and Wei 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*, pages 178–185. IEEE Computer Society, 2018.