Towards Self-Architecting Autonomic Microservices

Claudio Guidi
italianaSoftware s.r.l, Imola, Italy

Abstract

Autonomic computing is a key challenge for system engineers. It promises to address issues related to system configuration and maintenance by leaving the responsibility of configuration and reparation to the components themselves. If considered in the area of microservices, it could help in fully decoupling executing platforms from microservices because they permit to avoid coupling at the level of non functional features. In this paper, I explore the case of self-architecting autonomic microservices through the illustration of a proof of concept. The key points and the main challenges of such an approach are discussed.

2012 ACM Subject Classification Computer systems organization → Self-organizing autonomic computing

Keywords and phrases Autonomic computing, microservices, architectures

Digital Object Identifier 10.4230/OASIcs.Microservices.2020-2022.7

Supplementary Material *Software*: `https://github.com/klag/autonomic-microservices` [17]
archived at `swh:1:dir:8ddd029c52e9caa1743c0394a326b95c8623c422`

1 Introduction

In recent years, the push towards the digitisation of processes has led to an increase of system complexity, both in terms of the number of applications and integration processes. Such a fact had impacts both on the organizations and the system architectures.

As far as organizations are concerned, the necessity for software that is increasingly aligned with business needs, has led to the adoption of organizational processes targeted to minimize the delivering time and to increase the frequency of releases. The most important example in this case is represented by DevOps[12, 16] that is a software development approach used for reducing the distance between development and operational activities. On the other hand, if we consider the evolution of architectures, we have observed the raise of *microservices*, that is a distributed oriented architectural approach which introduces a new transformative force in the design and deployment phases of a software system. Following a microservices approach, functionalities are isolated by responsibility, independently deployed and distributed into the system in order to allow for independent scaling and management. Each microservice is designed, developed and managed by a different team where all the required competences are present. Both DevOps and microservices can be considered as two complementary forces that, when combined, aims to:(i) increase the organization's speed; (ii) create a software application, or more generally, a software system, by integrating multiple independent components. They contribute to make the final system more resilient, flexible and scalable. The price to pay is a general rise of the overall complexity of the systems. DevOps and microservice platforms, even if they are commercial or custom, play a fundamental role for addressing such a complexity. In real cases, these platforms are usually a mix of technologies that address different functionalities. As an example Jenkins[6] is used for programming automatic tasks for DevOps, GitLab[3] is used as a code repository, Docker[2] and Kubernetes[9] are used for managing the containerization layer, OpenShift[11] is used for addressing the infrastructural layer and so on.

Automation is a key aspect of DevOps, especially when applied in the context of microservices, as these considerably increase the number of deployed components and, therefore, increasingly require automated tools for their management. In this context, autonomic

Joint Post-proceedings of the Third and Fourth International Conference on Microservices (Microservices 2020/2022).
Editors: Gokila Dorai, Maurizio Gabbrielli, Giulio Manzonetto, Aomar Osmani, Marco Prandini, Gianluigi Zavattaro, and Olaf Zimmerman; Article No. 7; pp. 7:1–7:14
OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

computing is a key challenge for system engineers [21] and it may be considered as a further step forward in automating systems. It promises to address issues related to configuration, maintenance, updating and security by leaving all the responsibilities to the software itself without any human intervention. In the vision of autonomic computing, an autonomic component possesses all the capabilities for self-detecting errors, performance deterioration and security threats, and consequently take actions for repairing and adjusting their status. Moreover, they are able to detect when their internal modules need to be updated, and they are able to correctly install and configure the new versions of them. Differently from DevOps, that is developer and IT operator-oriented focusing on collaboration and automation in the software development lifecycle, autonomic computing aims to create fully autonomous and self-sufficient computing systems that can adapt, optimize, and protect themselves. While DevOps is a widespread system of practices applied in many different contexts, autonomic computing is still a new and little explored field, especially in production environments.

In this scenario, autonomic computing could be considered as a contribution to the evolution of DevOps, applied specifically to microservices, for reducing human intervention. In order to understand the role it could play, we can start by noting that, even if microservices should be agnostic with respect to the technologies used for developing, they are actually coupled with the platforms due to non functional constraints and limitations inherently present. For example the set of programming technologies could be limited because only some of them are managed in the existing DevOps pipelines. Indeed, developing and maintaining a DevOps pipeline for a given technology comes with an organizational cost that could be not convenient if the related stream of work is not relevant. Furthermore, since the observability of components and their fault management processes are often centralised and delivered exploiting different platforms, depending on how an organization approaches their management (e.g. following ITIL[5] strategies *Service Operation* and *Continual Service Improvement*), microservices must be equipped with specific connectors or even specially programmed to adhere to general guidelines of the organization. Summarizing, if from a functional point of view a microservice can be designed to be independent and decoupled with respect of the rest of the system, from a non functional point of view it could be strongly coupled with the platform where it is developed and deployed. Such a coupling could represent an issue when some modifications at the level of the platform must be performed. Depending on their impact, the risk is that all the microservices must be revised in order to adhere to the new platform standards. Moreover, in case of migration from a platform to another, an important refactor of the microservices must be considered and made. Minimising non-functional coupling between microservices and the platforms on which they are deployed can enable their truly independent design, development and deployment. Such a milestone could be achieved by introducing autonomic computing at the level of the microservices, thus making them independent and autonomous in managing non-functional properties w.r.t. the execution environment where they are deployed. At the present, in the current practices, microservices are not equipped with any self-adaptation logic, they are never aware of the context where they are executed. Every operational activity on a microservice, also those that are automatic and related to some non functional aspects like auto-scaling, are always demanded to the external platforms where they are executed.

In this paper, I investigate the possibility to make a step forward in the direction of a non functional decoupling between microservices and platforms by exploring the idea of self-architecting autonomic behaviours in microservices. In particular, I propose a proof of concept where an autonomic microservice is able to negotiate the scaling, and the de-scaling, of one of its internal components with the execution environment. The main contribution of this paper is to show how a microservice of this kind could be developed, which are the key points to be considered and which are the main challenges to overcome for achieving these results.

Section 2 reports a conceptual view of what a self-architecting autonomic microservice is, Section 3 describes the proof of concept by focusing on those elements that are relevant for this paper; Section 4 and Section 5 report discussions on some key points and challenges that can be extracted from the proof of concept; finally Section 6 contains conclusions and comments about references.

2 Self-architecting conceptual view

This section is devoted to provide a conceptual overview of what is meant here by self-architecting autonomous microservices. The discussion is kept as abstract as possible in order to illustrate only the basic concepts, focusing on the architecture of the components of the microservice application, without taking into account other details and, above all, without considering the execution context in which the microservice is deployed.

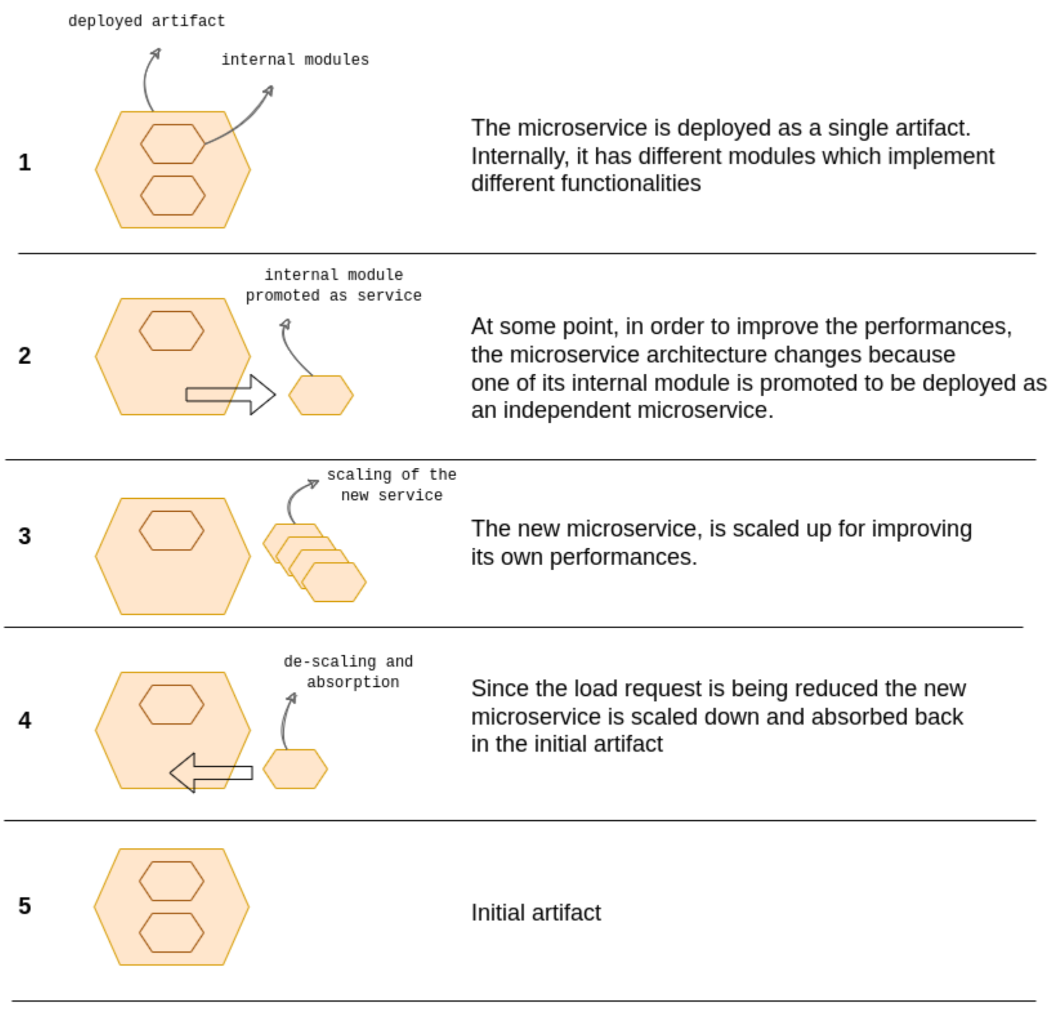
deployed artifact
internal modules
1
The microservice is deployed as a single artifact. Internally, it has different modules which implement different functionalities
internal module promoted as service
2
At some point, in order to improve the performances, the microservice architecture changes because one of its internal module is promoted to be deployed as an independent microservice.
scaling of the new service
3
The new microservice, is scaled up for improving its own performances.
de-scaling and absorption
4
Since the load request is being reduced the new microservice is scaled down and absorbed back in the initial artifact
5
Initial artifact

Figure 1 Expected architectural evolution of a microservice application which abstractly refers to what is going to be detailed with the proof of concept.

Such an abstraction will allow us to highlight the main contribution behind the adoption of an autonomic behaviour in modifying the architecture of a microservice application, and will make it easier to understand the description of the proof of concept that will be presented in the next section.

Figure 1 reports a conceptual architectural evolution of the microservice application targeted in the proof of concept. In Step 1, the microservice application is initially deployed: it is a single executable artifact, internally composed by different business logic modules which deal with different functionalities. In Step 2, in order to improve the performances due to an increase of load, a decision is taken: one of its internal module is promoted to become an independent microservice and it is deployed separately from the initial artifact. In Step 3, the new microservice is scaled up to specifically improve its own performances. In Step 4, since the external load is being reduced, the new microservice is scaled down and absorbed back in the initial artifact. In Step 5, the microservice is operating as it was initially.

The architectural evolution described in Figure 1 has been kept deliberately abstracted to focus on concepts about architecture modification, without specifying any actor which is responsible to perform the steps. Some questions easily emerge: in which steps there is a human intervention? In which steps does the microservice act autonomously? Moreover, which are the differences if we approach the same evolution in a conventional way, or using an autonomic approach? So far, a discussion on how such an architectural evolution could be approached and which impacts it could have on the microservice development and maintenance, has not been reported. The same evolution indeed, could be achieved following a conventional approach to the development of microservices, or using an autonomic one. A brief comparison between the conventional approach and the autonomic one, together with an analysis of the roles involved in each step, will help us to focus on better highlighting the impact of a self-architecting autonomic microservice, which is the subject of this paper. In Table 1, a comparison between the convectional approach and the autonomic one, is reported, whereas in Table 2, there is a more detailed analysis about the actors involved in each step where, for the sake of this discussion, the roles *developer* and *sysadmin* are merely indicative and they must be considered as just abstract references to two archetypal roles into an organization. A deep analysis on the impacts that an autonomic microservice could have on the different strategies adopted for managing software, is out of the scope of this paper.

As can be seen, as far as the autonomic microservice is concerned, quite all the steps are managed by the microservice itself which possesses the capability to dramatically modify its architecture, whereas in a conventional approach, all the steps involve developers, system administrators, or both. In particular, in the autonomic approach all the steps are managed by the microservice, with the exception of Step 1 that is related to the first release of the software and that is in charge to the developer in both cases. In the conventional case steps 2 and 4 are in charge to human roles, whereas Step 3, if auto-scaling feature is used, it is in charge to the execution environment. In any case, in the conventional scenario, the microservice does not take decisions or performs activities which implies a change on its own architecture, but all the actions are delegated to external actors. On the contrary, in the autonomic scenario all the actions are delegated to the microservice itself.

Table 1 Differences between a conventional approach and an autonomic one. Step 5 is reported together with Step 1 because they are equivalent.

Step	**Conventional Approach**	**Autonomic Approach**
1 (5)	(i) The developer implements the microservice in a standard way as a unique artifact, internally composing different business logic modules; (ii) The developer releases the artifact and automatically (or manually) the artifact is deployed and executed (as container or a process) into a target execution environment (e.g. a containerization layer).	(i) The developer implements the microservice as an autonomic one, by envisioning the possibility for the microservice to make some modifications about its own architecture; (ii) The developer releases the artifact and automatically (or manually) the artifact is deployed and executed (as container or a process) into a target execution environment (e.g. a containerization layer).
2	(i) The developer and the sysadmin analyze the performances of the microservice; (ii) the developer decides to divide the artifact into two by promoting one of its internal modules as a microservice. She extracts the code of the module to expunge, from the initial artifact, then she puts it into another project; (iii) the developer releases both the initial artifact, without the expunged module, and the new one; both of them are deployed replacing the former one. Optionally, the new one can be deployed together with some directives to the execution platform for auto-scaling it, if not the number of replica must be defined at deploying time.	(i-iii) The microservice auto-detects that its performance is deteriorating and it decides to promote one of its internal components as a microservice. Thus, it directly deploys the new microservice by interacting with the executing environment.
3	(i) The sysadmin analyzes the performances of the microservice; (ii) The sysadmin decides to scale up or down the new microservice, in order to tune its performances. If the auto-scaling feature has been set at the previous step, the execution platform does it automatically. Note that if it is the case of a manual intervention, such a decision should be a long-term one because it is not reasonable to manually change the number of replica day by day.	(i-ii) The microservice autonomously decides to scale up or down the new component by defining the number of current replicas by interacting with the execution environment.
4	(i) The sysadmin analyzes the performances of the microservice; (ii) The developer decides to restore the initial version of the microservice because the load is now very low and there is no need to have two microservices. Note that, in this case, usually, the developer would leave the last architecture (that of Step 3) with just one replica for the new microservice, in order to avoid the costs of a new release; (iii) The developer releases the previous artifact.	(i-iii) The microservice decides to absorb new microservice ang restoring the initial architecture.

Table 2 Detailed steps with the focus on the involved actors.

Step	Description	Conventional Approach	Autonomic Approach
1 (i)	Design and development	Developer	Developer
1 (ii)	Release and deployment	Developer	Developer
2 (i)	Analysis of the performance metrics	Developer or Sysadmin	Microservice
2 (ii)	Decision to modify the architecture, and its related implementation	Developer	Microservice
2 (iii)	Release (in case of auto-scaling, configuration of the environment)	Developer and Sysadmin	Microservice
3 (i)	Analysis of the performance metrics of the new component	Execution Environment (or Sysadmin, if auto-scaling is not set)	Microservice
3 (ii)	Decision and implementation of scaling up or down	Execution Environment (or Sysadmin, if auto-scaling is not set)	Microservice
4 (i)	Analysis of the performance metrics	Developer or Sysadmin	Microservice
4 (ii)	Decision to restore the initial version and its related implementation	Developer or Sysadmin	Microservice
4 (iii)	Deployment of the initial version	Developer	Microservice

3 Proof of concept

The main objective of the proof of concept described in this section is to show the basic mechanisms behind the implementation of an *Autonomic Microservice*, which is able to modify its own architecture depending on its own performances by negotiating it with the *Execution Environment*.

In Figure 2, a representation of the architecture developed in the proof of concept is reported. The *Autonomic Microservice* is deployed within a Docker[2] container, controlled using standard Docker API by the *Execution Environment*. Moreover, it is able to self-calculate the average response time of its own API and, depending on the results, it is able to negotiate with the *Execution Environment* a change of its architecture by scaling a specific sub-component, which takes the form of another microservice.

3.1 Architecture

Since it is a proof of concept, some assumptions have been made in order to keep it as simpler as possible:

- *Simplified model for the Autonomic Microservice*: The *Autonomic Microservice* models a microservice which implements some basic functionalities for managing a set of data, and it is assumed it implements autonomic features modelled following a MAPE-K loop[13, 20].

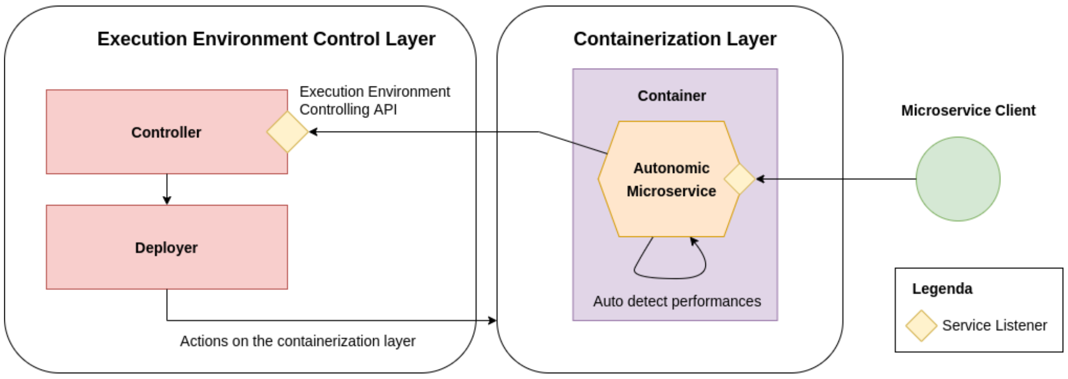

Figure 2 Logical representation of the architecture developed in the proof of concept.

In particular, all the different autonomic functionalities of the MAPE-K model have been simplified and condensed into a single internal component of the microservice. For the same reasons, all the algorithms for decision making and performance deterioration detection have been kept very basic and raw.

- *Simplified model for the Execution Environment*: the *Execution Environment* ideally models a general platform which is able to manage microservice deployment. A full representation of all of its parts is out of the scope of this paper. Here it has been modelled with a simple service that, on the one hand, it is able to interact with a containerization layer by invoking its standard API and, on the other hand, it exhibits a new set of API that are specific to be invoked by autonomic microservices in general.
- *Execution Environment agnosticism*: here we assume that the *Execution Environment* is agnostic with respect to the actual deployment an autonomic microservice may have at runtime. Apart from the first deployment, the *Execution Environment* does not own other container images nor it is aware about other components the autonomic microservice may request to have. Moreover, no specific rules for monitoring or scaling have been set in the *Execution Environment*. All the knowledge about the *Autonomic Microservice* management is in charge to the *Autonomic Microservice* itself.

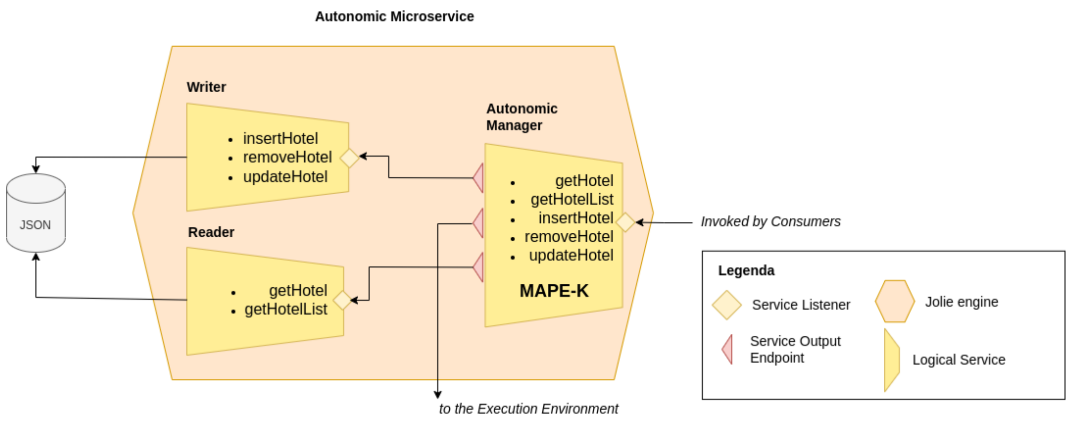

Figure 3 Autonomic Microservice inner logical architecture.

In Figure 3 the inner architecture of the *Autonomic Microservice* is reported. The microservice manages a generic set of data about a list of hotels that, for the sake of simplicity, has been modelled with a JSON[8] file[1]. Such a persistence layer is accessed by two internal services: *Writer* and *Reader*. The former is in charge to implement writing APIs (*insertHotel*, *updateHotel* and *removeHotel*), whereas the latter is in charge to implement the reading ones (*getHotel* and *getHotelList*). Neither of the two services directly exposes the APIs to the end consumer, but they are aggregated and embedded within the service *Autonomic Manager*, resulting in a single deployable artifact. A consumer can access all the APIs aggregated into the *Autonomic Manager*, by invoking its public listener where they are all available. Besides playing the role of a gateway for the reading and writing APIs listed above, the *Autonomic Manager* is also in charge to manage some autonomic features that allows for scaling the sub-service *Reader*. In particular, following a MAPE-K approach, the autonomic features of the *Autonomic Manager* can be summarized as it follows:

- *Monitor*. Since it is proxying all the requests to the inner services (*Reader* and *Writer*) it is able to capture all the metrics related to the API invocations, like invocation timestamp, reply timestamp and duration. In particular, it retrieves only those of the *Reader* because it is the component that can be scaled.
- *Analyse*. It calculates the average duration of the last ten invocation of the API of the *Reader*.
- *Plan*. It decides for a scaling or a de-scaling of the *Reader* depending on a threshold for API duration time.
- *Execute*. It interacts with the *Execution Environment* in order to ask for scaling or de-scaling the *Reader*.
- *Knowledge*. It manages the definitions of all the internal components (e.g. the *Reader*), their actual configuration (e.g. the number of active replica), and their configuration.

3.2 Runtime behaviour

In Figure 4 two scenarios, *before scaling* on the left and *after scaling* on the right, are reported. The *before scaling* represents a normal scenario where the *Autonomic Microservice* is simply deployed within a container. On the other hand, the *after scaling* represents a scenario where the *Autonomic Microservice* has been stressed with an extra load by a test consumer, and it negotiated with the *Execution Environment* for a scaling of the service *Reader*. In particular, it has been supposed that the *Autonomic Microservice* requested n instances of the service *Reader*. It is worth noting that all the instances of the service *Reader* are dynamically proxied by the *Autonomic Manager* which is in charge also to load the balance among them.

In Figure 5 the sequence chart, which describes the message exchanges between the *Autonomic Microservice* and the *Execution Environment* in case of scaling, is reported. A test client forces an extra load by continuously sending messages on the API *getHotelList* (1,2); concurrently the *Autonomic Microservice* calculate the average response time and detects a deterioration of such a metric (3). It is worth noting that, in order to trigger the scaling mechanism, the response time delay is simulated within the *Autonomic Manager* by augmenting the real measure with an extra delay. When the average time is greater than

[1] For the sake of simplicity the persistence layer has been mapped into a JSON file instead of using a structured one like a database. In the proof of concept, data consistency issues have been taken into account adding simple guards on writing and reading operations. A deep analysis about the impacts of self-architecting autonomic microservices and data consistency in a distributed scenario, is out of the scope of thsi paper.

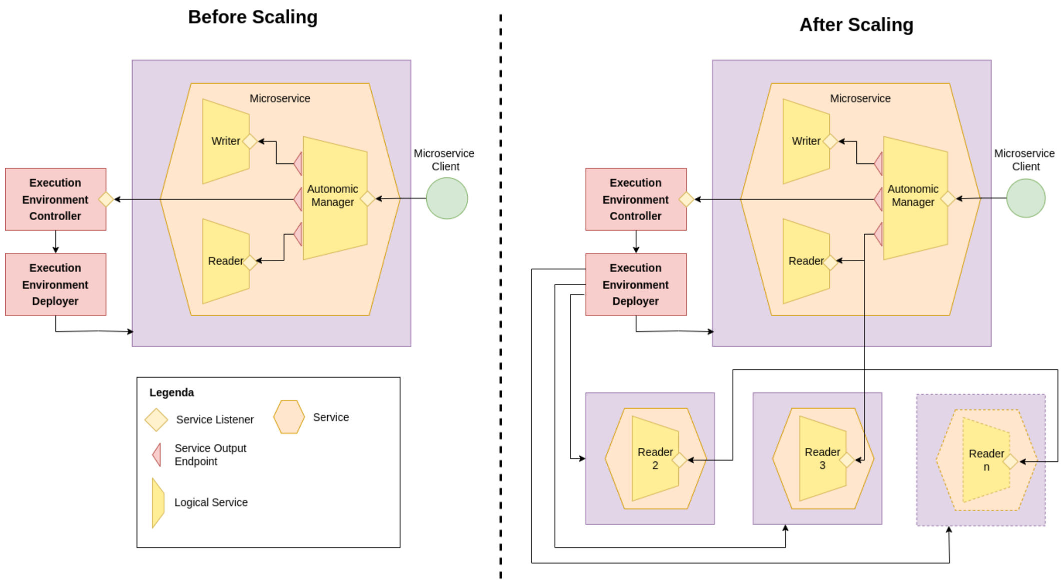

Figure 4 Microservice architecture before scaling and after scaling.

a threshold ($AVERAGE\_MAX$), the *Autonomic Manager* requests an instance of service *Reader* to the *Execution Environment* by sending also its definition (4). The *Execution Environment* creates the image for service *Reader* if it is not already stored in the internal catalogue of the containerization layer (5), and then run the container (6). Finally, it sends back the binding details of the new container to the *Autonomic Manager* (7). As a last step, the *Autonomic Manager* binds the new container into its gateway and starts to balance the load towards the new container too. Similarly, the *Autonomic Manager* can detect an improvement of the response times and request the removal of the containers that are not more necessary.

3.3 Implementation choices

The system has been realized using the service-oriented programming language Jolie[7] which has been chosen because it allows to easily implement the following aspects:

- *Embedding services.* It permits to dynamically embed a service into another. Thanks to the operator called *embedding*[19], a set of services can be executed in a distributed manner or run in the same engine. When executed within the same engine, the inner communication among the services are automatically resolved at the level of the memory without network exploitation. In the proof of concept, the *Autonomic Microservice* initially embeds both the *Writer* and the *Reader*.
- *Aggregating services.* Thanks to the operator *aggregate*[19], it permits to collect and expose APIs of different services into one single listener (in Jolie it is called *inputPort*), thus permitting to easily develop light API gateways. The aggregator plays the role of a proxy by receiving an API invocation and delivering it to the aggregated service that actually implements it. In the proof of concept, the *Autonomic Microservice* plays the role of the gateway by delivering all the incoming requests to the replica of the service *Reader*.

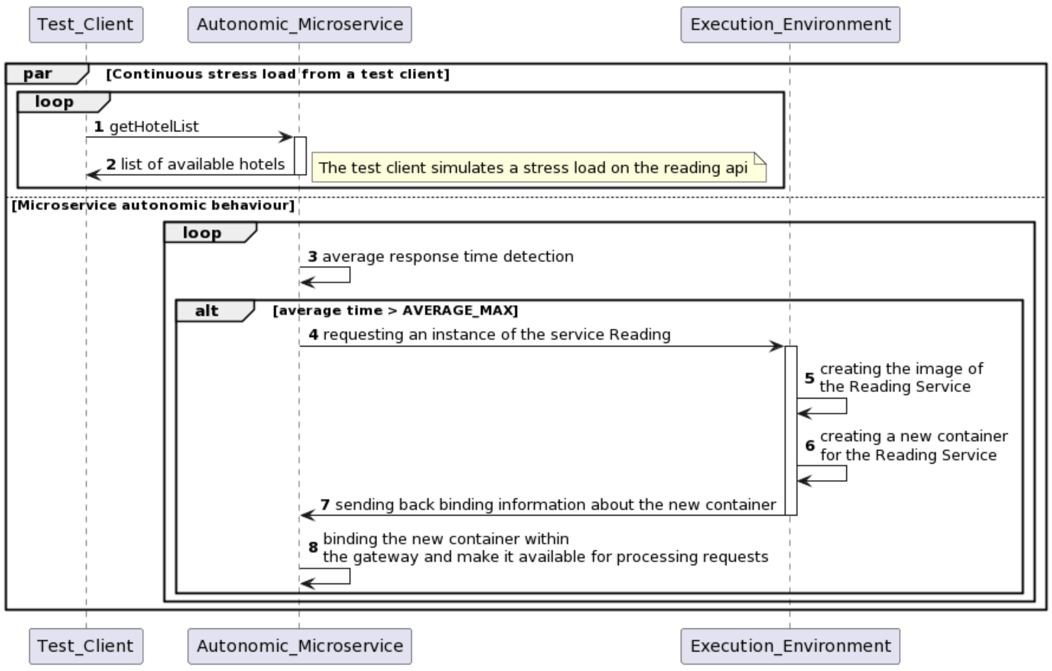
Test_Client
Autonomic_Microservice
Execution_Environment
par [Continuous stress load from a test client]
loop
1 getHotelList
2 list of available hotels
The test client simulates a stress load on the reading api
[Microservice autonomic behaviour]
loop
3 average response time detection
alt [average time > AVERAGE_MAX]
4 requesting an instance of the service Reading
5 creating the image of the Reading Service
6 creating a new container for the Reading Service
7 sending back binding information about the new container
8 binding the new container within the gateway and make it available for processing requests
Test_Client
Autonomic_Microservice
Execution_Environment

Figure 5 Sequence chart diagram for scaling.

- *Implements service functionality mobility.* It implements service functionality mobility[18] by permitting to easily send a service definition from a service to another by message. The engine which receives a service definition can dynamically embed and run it. In the proof of concept, the *Autonomic Manager* sends the definition of service *Reader* to the *Execution Environment* in order to instantiate a new replica.
- *Fast API modelling.* It permits to easily model an API using its specific syntax without following standards like openAPI[10] or gRPC[4], while preserving a comparable level of expressiveness w.r.t. them. Such a technological feature permits to reduce the technology stack burden and keep the proof of concept as simple as possible.

The code repository is public and available for inspection in [17].

4 Key points

Starting from the experience matured with the developing of the proof of concept, and following a MAPE-K loop approach, in the following some important key points that must be considered when designing and developing a self-architecting autonomic microservice, are reported:

1. **Monitor**: The *Autonomic Microservice* must collect all the required metrics for taking decisions about its own architecture that depends on the Service Level Agreements defined for that service. They can be application related, infrastructure related or both. In the former case, the microservice must be able to internally collect the metrics; in the latter case, the microservice must ask for them to the *Execution Environment*, thus implying that there are specific API for retrieving infrastructure metrics when needed. In general, the developer must be aware about the metrics to collect, and she has to know where and when they must be measured in the code.
   - In the proof of concept the *Autonomic Manager* directly retrieves the response time of each API invocation and it keeps on memory the last ten measures.

2. **Analysis**: In general, the analysis of the metrics should be condensed in few parameters calculated at runtime, thus avoiding to persist a log storage. In order to not interfere with the business logic, such a calculation should be performed concurrently. Some kind of extra volatile memory components could be introduced for persisting a limited buffer of logs. In any case, as for the monitor phase, also for the analysis, the developer must be aware of the parameters to calculate and their algorithms, and she has to design and implement a proper architecture for dealing with them.
   - In the proof of concept, after each invocation call, the *Autonomic Manager* asynchronously calculates the average duration of the last ten invocations and, depending on the result, it sends a request for a new replica of the service *Reader*.
3. **Planning**: This phase can be as complex as desired depending on the level of transformation that the service can achieve. In general, the implementation of these algorithms requires a deep knowledge about the possible evolution of the architecture. The developer must identify the degrees of freedom of the microservice and must be familiar with all the possible architectures that can be derived from it. The more degrees of freedom there are, the more the system can reach unexpected and unpredictable configurations.
   - In the proof of concept, the microservice had just one degree of freedom: it can choose to scale or remove instances of the service *Reader*. As a first glance, it looks very simple and straightforward. But it is worth noting that, in the example, there is no programmed upper limit in the algorithm. This means that the planning relies solely on the assumption that scaling the service *Reader* will eventually lead to a decrease in response times. However, if, for some reason, the variation of the response times in the real system is not strictly dependent on the number of running instances of the service *Reader*, the microservice may potentially require an infinite number of its instances, thus harming the entire system.
4. **Execution**: The *Autonomic Microservice* must implement the mechanics for transforming its own architecture. It must be able to perform the right calls to the *Execution Environment* for requesting new instances and removing the existing ones, but it also needs to provide all the components for integrating the new instances within its execution boundary.
   - In the proof of concept, the *Autonomic Manager* plays also the role of proxy and load balancer for correctly dispatching the requests to the instances of the service *Reader*. In this case, the load balancing strategy has been encoded at developing time, but it is possible to imagine making it configurable or even negotiable with the *Execution Environment*. Moreover, it is reasonable to assume that also the *Autonomic Microservice* must exhibit a set of API for being invoked by the *Execution Environment*, thus permitting a two-sided negotiation. Indeed, some architectural changes could be triggered by the environment (e.g. for optimizing the resources).
5. **Knowledge**: the *Autonomic Microservice* must manage the knowledge about its own architecture and its dynamic modification at runtime, thus it must be able to reconstruct the state of the architecture at any given time and in any condition (e.g. after a malfunctioning).
   - In the proof of concept the *Autonomic Manager* actually collects all the definitions of the components it asks to instantiate and it is able to properly configure them. But, in the current implementation, the mapping of the replicas of the service *Reader* are managed only in the volatile memory and in case of crash, the service is not able to restore such a list, thus making the existing replicas useless.

5 Main Challenges

Starting from what was outlined in the previous section, here I highlight three main challenges that need to be addressed in order to envision an engineered utilization of self-architecting autonomic microservices. In particular, these three challenges specifically focus on three different aspects: *Development activities*, *Preparation of the execution environment* and *Security management*. *Development activities* were considered because of the huge impacts autonomic computing could have on the development phase; *Adaptation of the execution environment* was considered because the adoption of an autonomic computing strategy is strongly coupled with an execution environment capable of managing it; finally *Security management* was considered because of the high level of risks that could be raised by shifting the responsibilities of many controls to the component itself.

It is important to bear in mind that the following list is not intended to cover all the critical aspects, but it is the result of a first internal evaluation about applying the approach presented in this paper, on products and applications of the author's company.

1. **Development activities:** *alleviating developer's cognitive burden.* In general, it is possible to state that the implementation of a self-architecting autonomic microservice requires an increment of the cognitive burden in charge to the developer that must be aware of all the aspects regarding the autonomic features: monitor, analysis, planning, execution and knowledge. The topic of tests deserves a special mention, because it will be necessary to test the different architectural configurations achievable by the microservice by simulating the various expected triggers and possibly mocking the execution environment, thus increasing the complexity of this task. Such a challenge could be addressed by introducing a development framework that already takes into account the various aspects necessary for the implementation of an autonomous service and partially manages them on behalf of the developer, moreover we could imagine to define the autonomic behaviour by using a specific declarative language which could help in better defining and controlling it.
2. **Adaptation of the execution environment:** *standardization of API.* In general, the *Execution Environment* should be enabled for accepting autonomic microservices, and the message exchange protocols between it and the autonomic microservice must be previously defined. Thus, the API of the *Execution Environment*, but also those that must be possibly offered by the microservice, should be standardized in order to make them equally available in any execution context where autonomic capabilities are accepted. This challenge requires a shared understanding among the developer community and, above all, among platform providers. A manifesto could be prepared and shared in order to attract valuable stakeholders for paving the way for standardization.
3. **Security management:** *security must be guaranteed by the Execution Environment.* Since an autonomic microservice is potentially able to completely change its initial architecture, thus transforming a service that it is initially safe into an harmful software artifact, the *Execution Environment* must take the responsibility to perform security checks on the autonomic microservices. In particular, the *Execution Environment* should be able to inspect the microservice and all its components before creating running instances, thus determining if the components contain malicious code. Such an aspect could be addressed by avoiding the execution of pre-compiled code, but postponing the compilation inside the *Execution Environment* and installing a microservice from sources. Constraints could be added in the allowed programming languages, thus reducing the security checks to formal ones as much as possible. As an example, languages like Ballerina[1] and Jolie[7] directly provide a linguistic tool for programming services that are then interpreted by an underlying engine that, like it happens in the proof of concept, could be directly provided by the *Execution Environment*.

6 Conclusions

The main contribution of this article is to take a step forward in the investigation of autonomous microservices capable of dynamically transforming their architecture at runtime to respond to a change in execution context. In literature there exist some general overviews about challenges and opportunity of autonomic components[15] and microservices[22] too, but a specific insight about self-architecting microservices is not reported. Other authors explored the possibility to implement autonomic microservices[25], but they focus on self-healing and versioning instead of self-architecting autonomic features.

The main benefit of the introduction of autonomic behaviours in microservices is the fully decoupling between execution environments and microservices. Such an objective is ambitious and disruptive, because it can potentially change the way microservices are developed and deployed. At the same time, however, in the long term, it could permit to reduce maintenance costs and platform's lock-in. In particular, self-architecting autonomic microservices could simplify the deployment phases because almost all the steps are delegated to the microservice. As a counterpart, issues like security, standardization and the increase of complexity on the developments side must be considered. In general new models and references are needed like in [14], where the authors propose a MAPE-K loop based reference for identifying the different responsibilities between the execution environment and the autonomic microservice.

As an evolution of this work, it could be interesting to investigate the relationship of self-architecting microservices with infrastructure as a code (IAC) approach[23] that is exploited for increasing automation in DevOps contexts. In particular, it could be interesting to apply a self-architecting behaviour over a IAC layer, thus extending the autonomic behaviour, so far restricted at the containerization level, to the infrastructure. Moreover, a non-functional decoupling between microservices and execution platforms could potentially impact internal organizational processes based on established standards, as for example ITIL[24]. Therefore, a potential area of investigation could involve analyzing how autonomic computing might influence these standards.

References

1 Ballerina. URL: https://ballerina.io/.
2 Docker. URL: https://www.docker.com/.
3 Gitlab. URL: https://about.gitlab.com/.
4 grpc. URL: https://grpc.io/.
5 Itil: Information technology infrastructure library. URL: https://www.axelos.com/certifications/itil-service-management.
6 Jenkins. URL: https://www.jenkins.io/.
7 Jolie - the service oriented programming language. URL: https://www.jolie-lang.org.
8 Json. URL: https://www.json.org/.
9 Kubernetes. URL: https://kubernetes.io/.
10 Openapi specification. URL: https://swagger.io/specification/.
11 Openshift. URL: https://openshift.com/.
12 Patrick debois on the state of devops. URL: https://www.infoq.com/interviews/debois-devops/.
13 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. In Betty H. C. Cheng, Rogério de Lemos, Holger Giese, Paola Inverardi, and Jeff Magee, editors, *Software Engineering for Self-Adaptive Systems*, pages 48–70, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. doi:10.1007/978-3-642-02161-9_3.

14 Antonio Bucchiarone, Claudio Guidi, Ivan Lanese, Nelly Bencomo, and Josef Spillner. A MAPE-K approach to autonomic microservices. In *IEEE 19th International Conference on Software Architecture Companion, ICSA Companion 2022, Honolulu, HI, USA, March 12-15, 2022*, pages 100–103. IEEE, 2022. doi:10.1109/ICSA-C54293.2022.00025.

15 Betty H. C. Cheng, Rogério de Lemos, Holger Giese, Paola Inverardi, Jeff Magee, Jesper Andersson, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic, Giovanna Di Marzo Serugendo, Schahram Dustdar, Anthony Finkelstein, Cristina Gacek, Kurt Geihs, Vincenzo Grassi, Gabor Karsai, Holger M. Kienle, Jeff Kramer, Marin Litoiu, Sam Malek, Raffaela Mirandola, Hausi A. Müller, Sooyong Park, Mary Shaw, Matthias Tichy, Massimo Tivoli, Danny Weyns, and Jon Whittle. Software engineering for self-adaptive systems: A research roadmap. In Betty H. C. Cheng, Rogério de Lemos, Holger Giese, Paola Inverardi, and Jeff Magee, editors, *Software Engineering for Self-Adaptive Systems [outcome of a Dagstuhl Seminar]*, volume 5525 of *Lecture Notes in Computer Science*, pages 1–26. Springer, 2009. doi:10.1007/978-3-642-02161-9_1.

16 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. doi:10.1145/3385032.3385059.

17 Claudio Guidi. Autonomic microservices - proof of concept code repository. URL: https://github.com/klag/autonomic-microservices.

18 Claudio Guidi and Roberto Lucchi. Formalizing mobility in service oriented computing. *J. Softw.*, 2(1):1–13, 2007. doi:10.4304/JSW.2.1.1-13.

19 Claudio Guidi and Fabrizio Montesi. Reasoning about a service-oriented programming paradigm. In Maurice H. ter Beek, editor, *Proceedings Fourth European Young Researchers Workshop on Service Oriented Computing, YR-SOC 2009, Pisa, Italy, 17-19th June 2009*, volume 2 of *EPTCS*, pages 67–81, 2009. URL: http://arxiv.org/abs/0906.3920.

20 IBM. An architectural blueprint for autonomic computing. Technical report, IBM, jun 2005.

21 Jeffrey Kephart and D.M. Chess. The vision of autonomic computing. *Computer*, 36:41–50, feb 2003. doi:10.1109/MC.2003.1160055.

22 Nabor Chagas Mendonça, Pooyan Jamshidi, David Garlan, and Claus Pahl. Developing self-adaptive microservice systems: Challenges and directions. *IEEE Softw.*, 38(2):70–79, 2021. doi:10.1109/MS.2019.2955937.

23 K. Morris. *Infrastructure as Code.* O'Reilly Media, 2016.

24 Manuel Mora Tavarez, Jorge Marx Gómez, Rory V. O'Connor, Mahesh S. Raisinghani, and Ovsei Gelman. An extensive review of it service design in seven international itsm processes frameworks: Part ii. *Int. J. Inf. Technol. Syst. Approach*, 8:69–90, 2015. URL: https://api.semanticscholar.org/CorpusID:31968040, doi:10.4018/IJITSA.2015010104.

25 Yuwei Wang. Towards service discovery and autonomic version management in self-healing microservices architecture. In Laurence Duchien, Anne Koziolek, Raffaela Mirandola, Elena Maria Navarro Martínez, Clément Quinton, Riccardo Scandariato, Patrizia Scandurra, Catia Trubiani, and Danny Weyns, editors, *Proceedings of the 13th European Conference on Software Architecture, ECSA 2019, Paris, France, September 9-13, 2019, Companion Proceedings (Proceedings Volume 2),*, pages 63–66. ACM, 2019. doi:10.1145/3344948.3344952.