Generality vs. Reusability in Architecture-Based Self-Adaptation: The Case for Self-Adaptive Microservices

Nabor C. Mendonça
Post Grad. Prog. in Applied Informatics
University of Fortaleza
Fortaleza, CE, Brazil
nabor@unifor.br

David Garlan
Institute for Software Research
Carnegie Mellon University
Pittsburgh, PA, USA
garlan@cs.cmu.edu

Bradley Schmerl
Institute for Software Research
Carnegie Mellon University
Pittsburgh, PA, USA
schmerl@cs.cmu.edu

Javier Cámara
Institute for Software Research
Carnegie Mellon University
Pittsburgh, PA, USA
jcmoreno@cs.cmu.edu

ABSTRACT

Why is it so difficult to build self-adaptive systems by reusing existing self-adaptation services and frameworks? In this paper, we argue that one possible explanation is that there is a fundamental mismatch between the adaptation needs of modern software systems, and the architectural models and adaptation mechanisms supported by current self-adaptation solutions. We identify and discuss the main reasons leading to this problem by looking into a number of representative self-adaptation solutions that have been proposed in recent years, including open source frameworks and cloud-based services, from two perspectives: *generality*, i.e., their ability to support a variety of architectural models and adaptation mechanisms, and *reusability*, i.e., their ability to be reused without requiring substantial effort from software developers. We then make the case that recent industry progress toward microservices and their enabling technologies can open the way to the development of more general *and* reusable self-adaptation solutions.

CCS CONCEPTS

• **Software and its engineering → Software architectures**;

KEYWORDS

self-adaptation, microservices, service mesh

ACM Reference Format:
Nabor C. Mendonça, David Garlan, Bradley Schmerl, and Javier Cámara. 2018. Generality vs. Reusability in Architecture-Based Self-Adaptation: The Case for Self-Adaptive Microservices. In *12th European Conference on Software Architecture: Companion Proceedings (ECSA '18), September 24–28, 2018, Madrid, Spain.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3241403.3241423

ECSA '18, September 24–28, 2018, Madrid, Spain

ACM ISBN 978-1-4503-6483-6/18/09…$15.00
https://doi.org/10.1145/3241403.3241423

1 INTRODUCTION

After a relatively slow start, research in the field of self-adaptation has picked up significantly in recent years and is now following the regular path of maturation [18]. Based on an analysis of the field from the perspective of Redwine and Riddle's Software Technology Maturation (STM) model [14], Weyns argues that self-adaptation is currently in the phases of internal and external enhancement and exploration, where the technology is used to solve concrete real problems involving a growing community to show evidence of its value and applicability [18]. Still according to Weyns, popularization, which is the next (and last) phase in the STM model, is in a very early stage for self-adaptation, with only a handful of self-adaptation techniques, such as automated server management, cloud elasticity, and automated data center management, having thus far found their way to industrial applications [18].

This contrast between the state-of-the-art and the state-of-the-practice is even more evident for self-adaptation models and techniques that deal with multiple quality attributes, e.g., performance, reliability and cost [11], or address more abstract quality concerns, such as security [21]. Since offering multiple quality guarantees, including security, is a basic extra-functional requirement of any modern software system, the question of why existing self-adaptation models and techniques have been largely neglected by practitioners deserves further attention.

One possible explanation might be that engineering production-quality self-adaptive software systems requires far more advanced adaptation models and techniques than what is currently available in the state-of-the-art [17, 19]. Another possibility is that current software engineering methods and tools lack adequate support for implementing practical self-adaptation solutions. For instance, Yuan *et al.* [20] claim that the lack of engineering principles and repeatable methods for the construction of self-protecting software systems has been a major hindrance to their realization and adoption by industry. While both reasons are likely to be true, at least to a certain extent, neither of them explains why existing architecture-based self-adaptation frameworks, such as Rainbow [6], K8Scalar [4] and ActivFORMS [7], which were originally developed with reusability as a key design concern, are not more widely adopted in practice.

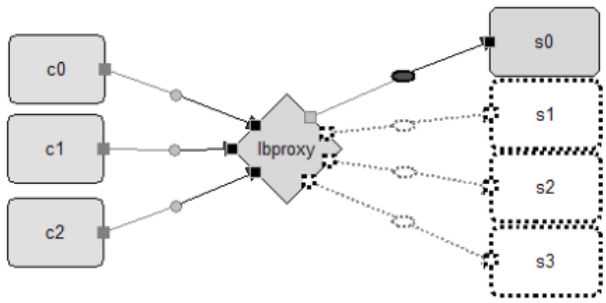

Figure 1: Znn's N-tiered architecture (extracted from [3]).

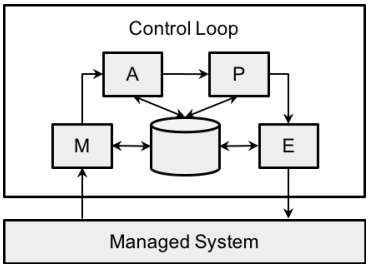

Figure 2: MAPE-based self-adaptation architecture.

In this paper, we argue that one major problem hindering a more systematic reuse of existing self-adaptation solutions is that there is a fundamental mismatch between the adaptation needs of modern software systems, and the architectural models and adaptation mechanisms supported by current self-adaptation services and frameworks. We discuss the main reasons leading to this problem by looking into a number of architecture-based self-adaptation services and frameworks that have been proposed in recent years from two design perspectives: *generality*, i.e., their ability to support a variety of architectural models and adaptation mechanisms, and *reusability*, i.e., their ability to be reused without requiring substantial effort from system developers. We frame our discussion by identifying multiple patterns for adding self-adaptation capabilities into existing systems. We then make the case that recent industry progress toward the microservices architectural style [10] and its enabling technologies can open the way to the development of more general *and* reusable self-adaptation solutions.

2 MOTIVATING EXAMPLE

To illustrate the issues involved in reusing existing self-adaptation solutions to add self-adaptation capabilities to an existing system, we will use Znn as an example. Znn is a simple web-based news service originally implemented as a benchmark system for Rainbow [3]. As Figure 1 indicates, Znn.com N-tiered architecture is composed of a load balancer and a pool of replicated web servers, some of which may be inactive. The load balancer balances client requests across the pool of replicated servers. The servers deliver to the clients static files (e.g., images and videos), as well as dynamic content (e.g., news populated from periodically-updated sources).

To extend Znn with self-adaptation capabilities, we consider the following adaptation needs:

Capacity It should be possible to dynamically adjust the number of active replicated servers.

Fidelity It should be possible to dynamically adjust the fidelity level of the content provided by servers to the clients. For instance, servers could be dynamically configured to provide different levels of content ranging from full multimedia (i.e., including all images and videos) to static text.

Protection It should be possible to dynamically enable protection mechanisms so that the service can protect itself against external attacks. For instance, the service may force clients to reauthenticate or pass a Turing test using a Captcha verification mechanism. Also, the service may blacklist or throttle the requests received from suspicious clients based on their IP addresses.

A self-adaptive version of Znn can benefit from the adaptation mechanisms put in place to satisfy the above needs to establish quality tradeoffs across multiple extra-functional dimensions, such as performance, cost, security, and user-experience. For example, increasing the number of active servers is likely to improve performance and make the service more resilient to Denial-of-Service (DoS) attacks. However, this also increases the service's operational cost. Alternatively, performance can be improved with no extra cost by only reducing content fidelity, but at the expense of user experience. Finally, forcing clients to reauthenticate or pass a Turing test may improve security, but also may negatively affect user experience and performance.

3 ARCHITECTURE-BASED SELF-ADAPTATION

A system is self-adaptive if it can reflect on its behavior at runtime and change itself in response to environmental conditions, errors, and opportunities for improvement [15]. Typically, self-adaptation capabilities are provided to some target system by adding a self-adaptation layer that reasons about observations of the system's runtime behavior, decides whether the system is operating outside its required bounds and what changes should be made to restore the system, and effects those changes on the system. This form of self-adaptation essentially involves adding a closed control loop layer onto the system. Adaptive control consists of four main activities: Monitoring, Analysis, Planning, and Execution (commonly referred to as the MAPE loop [9]), as shown in Figure 2.

Classical control loops use models of target physical systems to reason about control behavior. Similarly, self-adaptive software requires models to reason about the self-adaptive behavior of a system. Architecture models [16] represent a system in terms of its high level components and their interactions (e.g., clients, servers, etc.) reducing the complexity of the reasoning models and providing systemic views on their structure and behavior (e.g., performance, protocols of interaction, etc.). Much research in self-adaptive systems has therefore coalesced around using models of the software architecture of systems as the basis of reasoning about behavior and control, collectively termed architecture-based self-adaptive systems [11].

3.1 Generality vs. Reusability in Self-Adaptive Systems

Many types of architectural models have been used to manage the behavior of self-adaptive systems across multiple levels of abstraction. Irrespective of the architectural models used for that purpose, the managed system inevitably has to be implemented or changed in a way to accommodate the adaptation mechanisms required to enable such models to be monitored and acted upon by the control loop layer. For example, to allow managing the fidelity level of the content provided by a Znn server, that server must offer a mechanism (such as an environment variable or a management API) for allowing its fidelity level to be dynamically changed by the control loop components. In addition, the control loop components must be implemented with that specific fidelity tuning mechanism in mind. This two-way dependency between the managed system and the control loop layer is arguably the main challenge developers face when integrating existing self-adaptation solutions with an existing system [2].

To better decouple the managed system from its control loop layer, some architecture-based self-adaptation frameworks, such as Rainbow, use separate system-level components to support the monitoring and execution activities of the MAPE loop, called probes and effectors, respectively [6]. In this way, system developers are only required to provide the probes and effectors necessary to monitor and change the behavior of their system according to a given architectural model, without the need to implement or change any control loop component directly. However, while this solution certainly facilitates the reuse of the control loop components across self-adaptive systems that share the same architectural model, and of probes and effectors across systems that share the same adaptation mechanisms, it is of little help when developers need to define new self-adaptation strategies based on architectural models not previously supported by the self-adaptation framework at hand. In that case, developers have no option but to extend the existing control loop components to support the new architectural models, as well as to provide the probes and effectors necessary to enable those models to be monitored and changed at runtime.

3.2 MAPE Design Space

As we have mentioned previously, in this paper we focus on two specific design attributes of a self-adaptation solution, namely *generality* and *reusability*. By generality we mean the ability of a given self-adaptation solution to support a variety of architectural models and adaptation mechanisms. By reusability we mean the ability of a given self-adaptation solution to be applied and reused across systems and domains without requiring substantial development effort from system developers. Clearly, those two attributes are in conflict with each other, as the extension mechanisms that allow a given self-adaptation solution to support new architectural models and adaptation mechanisms usually make it difficult to reuse, while a reusable self-adaptation solution that only supports a fixed set of architectural models and adaptation mechanisms is by definition of limited generality. Figure 3 depicts our view of the design space for architecture-based self-adaptation solutions from the perspective of the generality and reusability attributes. We chose to represent the design space as resembling a power law distribution, to reflect the competing nature of those two attributes.

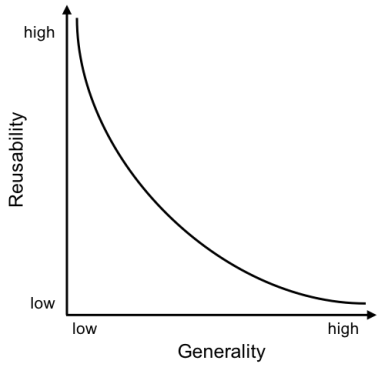

Figure 3: MAPE design space as a power law distribution.

In the next section, we will overview some representative architecture-based self-adaptation solutions from the perspective of their potential generality and reusability. To this end, we will group those solutions based on their MAPE integration strategies or *patterns*, and discuss whether and the extent to which each of them could be (re)used to satisfy the three Znn's adaptation needs described in Section 2.

4 MAPE INTEGRATION PATTERNS

Most modern software systems are delivered into production by relying on a number of infrastructure management services, such as those provided by tools like Docker and Kubernetes. Those services offer a number of benefits to both system developers and operators, as they greatly accelerate the tasks of packaging, testing, deploying, monitoring, and operating system components across multiple execution environments (e.g., staging, canary release, production) [5]. In this paper, we refer to those infrastructure services as belonging to the *infrastructure layer*.

Despite their increasing popularity in recent years, infrastructure tools such as Kubernetes have not (yet) fully embraced more sophisticated self-adaptation models, i.e., architectural models involving multiple quality dimensions and multiple resource types, as found in Rainbow as well as in many other research-based self-adaptation frameworks. For this reason, the question of how to better integrate existing research-based self-adaptation solutions within the context of existing industry-driven infrastructure management tools remains to be properly addressed by both the industry and research communities. Some early work in that direction will be discussed next.

4.1 System-Level MAPE Pattern

This pattern deploys all MAPE components as part of the system layer (see Figure 4). Indeed, from the perspective of infrastructure operators, all MAPE components are seen as an integral part of the managed system's architecture. This pattern is used by all traditional self-adaptation solutions that were originally developed without current infrastructure technologies in mind, as is the case of Rainbow.

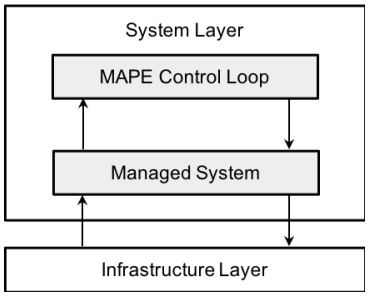

Figure 4: System-Level MAPE pattern.

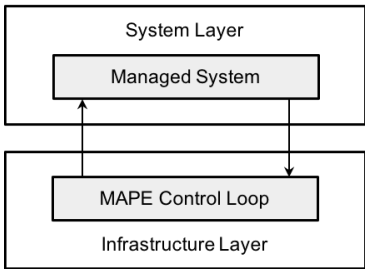

Figure 5: Infrastructure-Level MAPE pattern.

Self-adaptation solutions that follow this pattern are highly general, as all MAPE components are fully accessible to system developers. This means that developers are free to change or extend those components to support any architectural model and adaptation mechanism of interest. For example, Rainbow natively supports the three Znn's adaptation described in Section 2, and has been further extended to support other types of architectural models and adaptation mechanisms [2, 15].

The System-Level MAPE pattern does not fare well in terms of reusability though. The fact that all MAPE components are available at the system level also implies that it is up to system developers to test, deploy, monitor and operate them at runtime, alongside the other components that make up the managed system's architecture. This increases the developers's responsibility, requiring a great deal of effort from them in order to integrate those components into their system's architecture. Another drawback of this pattern is that it may refrain developers from reusing potentially more effective MAPE services provided at the infrastructure layer (e.g., a native autoscaling service) if similar services are also part of the self-adaptation solution being reused at the system layer.

4.2 Infrastructure-Level MAPE Pattern

In contrast to the System-Level MAPE pattern, this pattern deploys all MAPE components at the infrastructure layer (see Figure 5). Therefore, with this pattern MAPE services are fully integrated with other infrastructure services. Examples of self-adaptation solutions that follow this patterns include AWS Auto Scaling, which provides a rule-based reactive autoscaling service for systems deployed in the Amazon cloud, and Horizontal Pod Autoscaler (HPA), which provides a similar service for Kubernetes.

MAPE services provided at the infrastructure level are fully managed by the infrastructure management software. This makes them very easy to configure and reuse, as the managed system is required, by design, to comply with the runtime model imposed by the underlying infrastructure. On the other hand, since those services are not fully visible at the system level, developers are provided with only a fixed set of (possibly configurable) architectural models and adaptation mechanisms, i.e., those natively supported by the infrastructure management software at hand, which diminishes their generality. For example, both AWS Auto Scaling and HPA only support adding/removing replicas to/from a replicated service based on user-defined scalability rules. In that regard, both solutions could easily be reused to implement Znn's Capacity need, as long as the service itself is packaged and deployed in adherence to the AWS's and Kubernetes's runtime models, respectively. Nevertheless, those two solutions would be of no use to implement Znn's Fidelity and Protection needs, as their adaptation models are confined to manage a single type of resource, i.e., service replicas. Another issue with this pattern is that, without proper configuration support, infrastructure-level adaptation mechanisms are oblivious to system-specific goals. So, having the infrastructure layer make the right tradeoffs when adapting might be particularly problematic when the infrastructure is shared by multiple systems with disparate goals.

We note that both AWS and Kubernetes offer other infrastructure services that could be useful to implement some of the attack mitigation capabilities mentioned as part of Znn's Protection need. However, those services are not currently managed by any self-adaptation service provided by either AWS or Kubernetes, and, as such, must be manually enabled/disabled by system developers.

4.3 Cross-Layer MAPE Pattern

This pattern offers a tradeoff between the System-Level and Infrastructure-Level MAPE patterns, by deploying MAPE components across both layers (see Figure 6). The idea is to extend the basic MAPE services provided by the underlying infrastructure with more sophisticated MAPE services provided at the system layer. In this way, MAPE services at the infrastructure layer are completely unaware of MAPE services at the system layer, with the former providing infrastructure-level sensing and actuating components to the latter. An example would be a system-level protection service that reuses basic monitoring and replica management services provided at the infrastructure level. To implement this pattern, system-level MAPE services must be designed targeting specific infrastructure-level MAPE services. This strategy has the benefit of avoiding redundant MAPE services being deployed at both layers. The downside is that the implementation of system-level MAPE services becomes tightly coupled to services of a particular infrastructure, which makes them hard to reuse across infrastructures. An example of a self-adaptation solution that follows this pattern is K8-Scalar [4], which implements a container-based autoscaling service on top of the native replica management services provided by Kubernetes.

Compared to the System-Level MAPE pattern, this pattern offers higher reusability, as the part of the MAPE layer that is provided at the infrastructure-level is completely hidden from system developers. On the other hand, this pattern offers lower reusability when

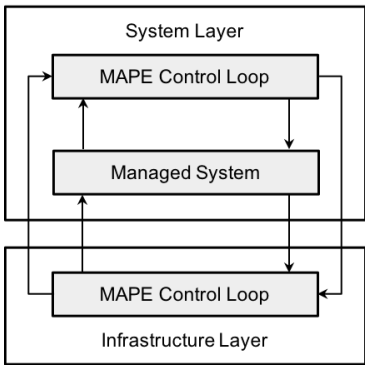

Figure 6: Cross-Layer MAPE pattern.

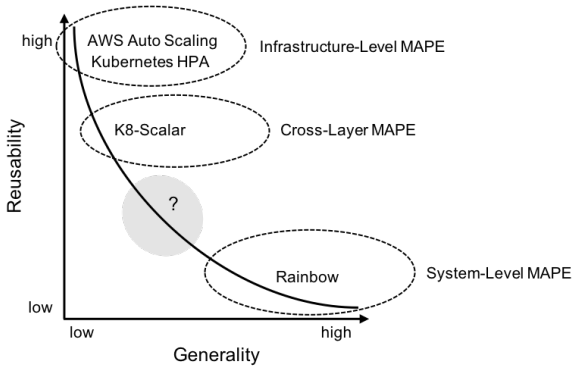

Figure 7: MAPE integration design space annotated with the names of the three patterns and their solutions.

compared to the Infrastructure-Level MAPE pattern, as developers now have to take responsibility for those MAPE components that are deployed at the system layer. In terms of generality, this patterns fares better than the Infrastructure-Level MAPE pattern, as developers have the option of implementing new architectural models and adaptation mechanisms by reusing or extending existing system-level MAPE components, and worse than the System-Level Integration pattern, as the types of models and mechanisms developers can create are to some extent coupled to infrastructure-specific MAPE services.

With respect to adding self-adaptation capabilities to Znn, solutions that follow this pattern could only be reused in the case that they already provide MAPE services supporting the architectural models and adaptation mechanisms necessary to satisfy Znn's adaptation needs. Otherwise, the missing models and mechanisms would have to be implemented by extending existing system-level MAPE services. In the case of the K8-Scalar system, its current version only supports autoscaling services, which could thus be reused to satisfy Znn's Capacity need only—again, assuming that Znn is adequately ported to run in Kubernetes.

4.4 MAPE Design Space Revisited

Figure 7 shows an updated version of the MAPE design space depicted in Figure 3, now annotated with the names of the three MAPE integration patterns and their respective self-adaptation solutions. The name of each pattern/solution is located along the Generality × Reusability curve, so that their position reflects their expected degrees of generality and reusability, as discussed above.

Note that we did not position any of the four solutions within the region highlighted in gray, which represents potential self-adaptation solutions that manage to strike a good balance between generality and reusability. We make the case for one such solution in the next section.

5 LOOKING AHEAD: THE CASE FOR SELF-ADAPTIVE MICROSERVICES

By examining the self-adaptation solutions discussed in the previous section, as well as their respective MAPE integration patterns, it is clear that, from a reuse and ease-of-use perspective, the best approach is to shield developers from all MAPE services by having them fully provided and managed at the infrastructure level. However, this approach also yields the lowest generality, as current infrastructure management technologies support only a limited set of architectural models and adaptation mechanisms. Instead of splitting MAPE services across both the system and infrastructure layers, as prescribed by the Cross-Layer MAPE pattern, we believe that a more effective approach to strike a balance between generality and reusability in the design of self-adaptation solutions is to widen the adaptation scope of the architectural models and adaptation mechanisms natively supported at the infrastructure level.

To achieve that goal, we argue that developers should adopt modern, industry-driven design approaches, particularly the microservices architectural style [10] and their enabling container-based technologies. The autonomous and independent nature of microservices makes them especially attractive as the locus for self-adaptation. Since a typical microservice is much smaller than a traditional (monolithic) system, developers could easily and rapidly develop alternative versions for each microservice, and have those versions being automatically replaced at runtime upon specific system conditions by the underlying infrastructure software. For example, in the case of Znn, the code responsible for generating the content to be sent to the clients could be refactored into a microservice with multiple parallel versions, with each version generating content with a different fidelity level. In addition, infrastructure services could be configured in a way to automatically switch the version of all instances of the content generation microservice, depending on factors such as the system's current workload or operational cost. Indeed, container orchestration tools like Kubernetes already provide features to automatically roll out and roll back service versions without the need to take the service down, which could be easily reused to provided a version switch adaptation service as described above. As the delay for adding new containerized instances to a service is considerably lower than when using traditional VMs, such a version switch service could be further integrated into an existing autoscaling service, thus offering a more flexible adaptation solution to manage both the performance and cost of each individual microservice.

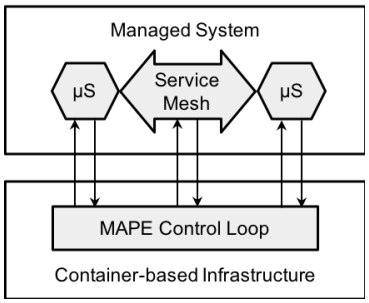

Figure 8: Microservices and Service Mesh MAPE pattern.

Another advantage of adopting microservices as first-class self-adaptation entities is that system developers can directly benefit from a fast growing number of (mostly open source) technologies being released as part of the microservice ecosystem [8]. Of particular interest are the so called "service mesh" technologies, such as Linkerd and Istio [1]. Those tools build on well-established reliable communication libraries, such as Finagle, and service proxies, such as Envoy, to provide a fully manageable service-to-service communication platform. Among the most useful features provided by a service mesh are service discovery, load balancing, fault tolerance, traffic monitoring, message routing, and authentication and access control [12]. For this reason, service mesh technologies are well-suited to address most of the communication-related security concerns raised as part of Znn's Protection need. In that direction, we envision a new MAPE integration pattern, where the system layer is composed of a collection of containerized microservices communicating exclusively via a service mesh (see Figure 8). In that pattern, both the containerized microservices and the service mesh are fully managed by MAPE services provided by the underlying infrastructure. Such a pattern would enable developers to automatically manage any architecturally relevant aspect of their systems, from the scalability, recovery and functional diversification of individual services to the monitoring and protection of their message-based interactions.

Ultimately, the Microservices and Service Mesh MAPE pattern could provide a uniform architectural style to develop, deploy and operate general-purpose self-adaptive distributed applications, without developers having to go through the burden of implementing or explicitly managing system-level MAPE services.

6 CONCLUSION

Despite the prevalence of architecture-based self-adaptation models and techniques in both academia and, more recently, industry, self-adaptation as a general and reusable software solution is yet to go mainstream. In this paper, we have argued that this is largely due to limitations of existing self-adaptation services and frameworks, which are either too difficult to reuse or too narrow in scope. In that regard, we have made the case that adopting a microservice architecture and its enabling technologies can be the way forward to address some of those limitations. We hope this discussion contributes to promote a better understanding of the interplay between reusability and generality in current and future self-adaptation solutions.

As future work, we plan to develop and empirically validate the Microservices and Service Mesh MAPE pattern on top of Kubernetes. We are also investigating how Kubernetes could be extended to support a general-purpose self-adaptation framework, such as Rainbow. Another interesting line of research would be to relate the three MAPE integration patterns to the different types of runtime uncertainty that self-adaptation is usually used to deal with [13]. Finally, we envision applying the microservices style to decompose the MAPE architecture itself, essentially turning each MAPE component into an autonomous service that could be independently deployed, updated and scaled.

ACKNOWLEDGMENTS

Nabor C. Mendonça is partially supported by CNPq under grants 313553/2017-3 and 207853/2017-7.

REFERENCES

[1] P. Calçado. 2017. Pattern: Service Mesh. http://philcalcado.com/2017/08/03/pattern_service_mesh.html. (2017). [Online; last accessed on July 18, 2018].
[2] J. Cámara et al. 2016. Incorporating architecture-based self-adaptation into an adaptive industrial software system. *Journal of Systems and Software* 122 (2016), 507–523.
[3] S.-W. Cheng, D. Garlan, and B. Schmerl. 2009. Evaluating the Effectiveness of the Rainbow Self-Adaptive System. In *Proc. of the ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE, 132–141.
[4] W. Delnat et al. 2018. K8-Scalar: a workbench to compare autoscalers for container-orchestrated database clusters. In *Proc. of the 13th Int. Symp. Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*.
[5] David F. and Jez H. 2010. *Continuous Delivery: Reliable software releases through Build, Test and Deployment Automation.* Addison-Wesley Professional.
[6] D. Garlan et al. 2004. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *Computer* 37, 10 (2004), 46–54.
[7] M. U. Iftikhar and D. Weyns. 2017. ActivFORMS: A Runtime Environment for Architecture-Based Adaptation with Guarantees. In *2017 IEEE Int. Conf. Software Architecture Workshops (ICSAW)*. IEEE, 278–281.
[8] P. Jamshidi et al. 2018. Microservices: The Journey So Far and Challenges Ahead. *IEEE Software* 35, 3 (2018), 24–35.
[9] J. O. Kephart and D. M. Chess. 2003. The Vision of Autonomic Computing. *Computer* 36, 1 (2003), 41–50.
[10] J. Lewis and M. Fowler. 2014. Microservices. https://martinfowler.com/articles/microservices.html. (2014). [Online; last accessed on July 18, 2018].
[11] S. Mahdavi-Hezavehi et al. 2017. A systematic literature review on methods that handle multiple quality attributes in architecture-based self-adaptive systems. *Information and Software Technology* 90 (2017), 1–26.
[12] W. Morgan. 2017. What's a service mesh? And why do I need one? (2017). https://buoyant.io/2017/04/25/whats-a-service-mesh-and-why-do-i-need-one/ [Online; last accessed on July 18, 2018].
[13] D. Perez-Palacin and R. Mirandola. 2014. Uncertainties in the Modeling of Self-adaptive Systems: A Taxonomy and an Example of Availability Evaluation. In *Proc. of the 5th ACM/SPEC Int. Conf. Performance Engineering (ICPE)*. ACM, 3–14.
[14] S. T. Redwine Jr and W. E. Riddle. 1985. Software Technology Maturation. In *Proc. of the 8th Int. Conf. Software Engineering (ICSE)*. IEEE Computer Society Press, 189–200.
[15] B. Schmerl et al. 2014. Architecture-Based Self-Protection: Composing and Reasoning about Denial-of-Service Mitigations. In *Proc. of the 2014 Symposium and Bootcamp on the Science of Security (HotSoS)*. ACM.
[16] M. Shaw and D. Garlan. 1996. *Software Architecture: Perspectives on an Emerging Discipline*. Vol. 1. Prentice Hall Englewood Cliffs.
[17] S. Shevtsov et al. 2017. Control-theoretical software adaptation: A systematic literature review. *IEEE Trans. Soft. Eng.* (2017).
[18] D. Weyns. 2017. *Software Engineering of Self-Adaptive Systems: An Organised Tour and Future Challenges.* Springer.
[19] D. Weyns and T. Ahmad. 2013. Claims and Evidence for Architecture-Based Self-Adaptation: A Systematic Literature Review. In *Proc. of the 7th European Conf. on Software Architecture (ECSA)*. Springer, 249–265.
[20] E. Yuan et al. 2013. Architecture-Based Self-Protecting Systems. In *Proc. of the 9th Int. ACM SIGSOFT Conf. Quality of Software Architectures (QSA)*. ACM, 33–42.
[21] E. Yuan et al. 2014. A Systematic Survey of Self-Protecting Software Systems. *ACM Trans. Aut. Adap. Syst.* 8, 4 (Jan. 2014), 17:1–17:41.