# Exploring Emergent Microservice Evolution in Elastic Deployment Environments

Roberto Rodrigues-Filho[a], Iwens Sene-Júnior[b], Barry Porter[c], Luiz F. Bittencourt[d], Fabio Kon[e] and Fábio M. Costa[b]

[a]Federal University of Santa Catarina, Araranguá, SC, Brazil
[b]Federal University of Goiás, Goiânia, GO, Brazil
[c]Lancaster University, Lancaster, United Kingdom
[d]University of Campinas, Campinas, SP, Brazil
[e]University of São Paulo, São Paulo, SP, Brazil

ARTICLE INFO

*Keywords*:
microservices, emergent software systems, self-adaptive systems, smart cities

ABSTRACT

Microservices have become an important technology to enable the dynamic composition of large-scale self-adaptive systems. Although modern microservice ecosystems provide a variety of autonomous adaptation mechanisms, when focusing on the microservice itself, they can only account for changes in the sheer increase in workload volume. On the other hand, when workload patterns change, efficient treatment requires the intervention of DevOps experts to manually evolve the internal architecture of services. Given the need to quickly adapt systems to respond to changes, solely relying on DevOps to react to workload pattern changes becomes a bottleneck for future systems. To address this issue, we advance the concept of emergent microservices, that autonomously adapt and evolve their internal architectural composition to better handle changes in the pattern of incoming requests without human intervention. We demonstrate the effectiveness of our approach by exploring this novel concept in the context of a microservice-based Smart City platform.

## 1. Introduction

The ability to cope with constantly changing operating environments is one of the key challenges in creating modern distributed systems. Popular domains such as general IoT applications, smart cities, data centre-based applications and web-based services are relevant examples of systems that are characterised by constant changes in their environments. All these changes create uncertain operating conditions that affect systems performance in unexpected ways [14].

To address the volatility issue ever more present in contemporary systems, the use of self-adaptive techniques has gained popularity. Self-adaptive systems are software systems that actuate on themselves via parameter tuning or architectural adaptation to accommodate changes to maintain the system at a desired level of performance [9]. Thus, self-adaptation is increasingly becoming an indispensable property to improve the dependability of contemporary systems.

Microservices [15, 18] represent a key technology to support the creation of self-adaptive systems. Besides their advantages in creating highly reusable and maintainable software, their inherently modular architecture also enables systems to be highly adaptable. In the industry, microservice-based systems can be adapted in a number of ways. Regarding the adaptation of functional properties, microservices are often used as building blocks to dynamically compose and recompose systems through central orchestration or through the enactment of predefined choreographies [11, 19].

Regarding performance, microservices are adapted when changes occur in two main ways: (i) workload volume; and (ii) workload pattern. Concerning the former, tools such as horizontal and vertical autoscalers are frequently used to autonomously create replicas or to increase the amount of resources available to each microservice instance in response to increases in request rate [28, 3]. This is currently the principal way to autonomously react to changes and quickly respond when new conditions arise. On the other hand, for changes in *workload patterns*, which are characterised by different mixtures and frequency of request types, the dominant approach to optimise the performance of microservices is to rely on experts (i.e., DevOps) to identify new ways to improve a service's implementation, and then manually rewrite, test and redeploy the service through continuous delivery [6]. This strategy is time-consuming and inefficient.

Thus, resource management through horizontal and vertical autoscaling is currently the only widely used approach to autonomously compensate workload changes. Although these mechanisms work well when changes involve only workload volume, they are less efficient (in terms of resource consumption) to cope with changes in workload patterns. These changes often affect the performance of microservices, making them slower to process the incoming requests. This performance degradation leads to a decrease in the volume of requests that each service instance is capable of processing, thus requiring more replicas than the system would otherwise need if the services were optimised according to the pattern. Therefore, for pattern changes, it is necessary to optimise the microservice implementation considering the pattern characteristics. Otherwise, scenarios where workload patterns frequently change may, e.g., result in equally frequent resizing of the number of microservices

ORCID(s): 0000-0002-3323-0246 (R. Rodrigues-Filho)

replicas, resulting in an (avoidable) negative impact on the overall system's performance.

In this paper, we explore a finer-grained level of adaptation, based on a novel approach to microservice engineering, called Emergent Microservices (EM) [26]. The EM concept enables microservices to autonomously adapt to workload pattern changes, automating, to a certain degree, the role of DevOps in microservice evolution. For that, we use the Emergent Software Systems (ESS) approach [23], originally proposed for the creation of self-adaptive systems, as part of the design of microservices.

This novel engineering approach entails the development of a collection of very small components that are used to compose a microservice. At runtime, replacing any of the constituent components enables adaptation of the microservice's internal architecture, with a possible boost in its performance. To illustrate, if a sudden workload change requires a microservice to repeatedly retrieve large amounts of historical data, adding a component that performs caching may significantly increase performance. On the other hand, if the workload changes again to require the retrieval of recently produced data with a low tolerance for staleness, that component may rather be replaced by one that does not cache data. The online learning of which component to use lies at the core of our proposed approach.

The paper presents the following contributions:

1. A demonstration of the impact that workload patterns have on microservice performance even when the request volume remains unchanged. In this regard, we also show that different microservice implementations exhibit distinct levels of performance when subjected to the same workload patterns, and that it is not obvious which implementation is the most suitable in each case;

2. A demonstration of emergent microservices being able to identify workload patterns and learn, at runtime, the most suitable implementation in a case-by-case fashion.

3. A demonstration of the potential of combining EM and horizontal autoscaling to handle changes in both the volume and the pattern of workloads. We show that, for a given workload, optimising the microservice composition saves resources.

The remainder of this paper is organised as follows. We consider related work in Section 2. Section 3 describes our methodology to build emergent microservices. Section 4 describes a microservice-based smart city platform that will serve as a single-case mechanism experiment for the use and evaluation of EM. The evaluation, in turn, is presented in Section 5, demonstrating the feasibility and effectiveness of the approach. Section 6 discusses our findings, and Section 7 concludes the paper.

## 2. Related Work

Traditionally, microservices are opaque. This means that their implementation details, including business logic details, employed frameworks, and programming languages are often not relevant as long as they implement their expected functionality. Our approach, on the other hand, takes an unexplored route to investigate the details of microservices implementations. We add a specific framework capable of autonomously changing the implementation of a single microservice while maintaining its external microservice appearance. For that reason, we employ specific terminology to refer to important parts of the system to avoid confusion with other widely used terms in software architecture and microservices communities.

Particularly, in this paper, we use the terms *microservice's internal architecture* or *micro-architecture* and *macro-architecture*. The microservice's internal architecture refers to the implementation details of a single microservice, whereas the term macro-architecture refers to the system's architecture as a whole, which is often referenced as **microservice architecture** in the literature (see [20, 29, 11]). The remainder of this paper employs this terminology to avoid confusion with previously established terminology.

We identified two categories of research that depict runtime self-adaptive behaviour in microservice-based systems. Such research efforts (i) discuss tools that are part of microservice platforms and aim at providing elasticity support for microservice instances; and (ii) focus on adaptation at a macro-level, using microservices as building blocks of a macro-architecture where adaptation is realised. We also survey DevOps practices as a way to quickly and continuously respond to new user demands, providing a different perspective for design-time adaptation in microservice-based systems. In this section, we review relevant related work in these two categories and compare them with our approach.

Microservices are often deployed in environments that provide supporting software to ensure quality of service (QoS) and service-level agreements (SLAs) in a variety of domains [12]. An important aspect to maintaining and/or increasing the performance of microservice-based systems is elasticity. Besides the use of popular tools, such as Kubernetes[1], which provides horizontal and vertical autoscaling, many papers explore different approaches to extend and improve those tools. Coulson et al. [7] explore how to identify which microservice should scale out in order to increase overall system performance. Rossi et al. [27] explore a hierarchical approach to ensure that elasticity performed on specific microservices has a positive impact on the global systems performance. As an alternative to horizontal scaling, Al-Dhuraibi et al. [1] also look into vertical scaling, combined with container migration when the host machine cannot vertically scale as required. Brondolin et al. [5] use these supporting tools not only to increase performance but to also limit power consumption.

Microservices are also often used as building blocks to create self-adaptive systems. These adaptive systems are the result of the composition and recomposition of multiple microservices through the enactment of choreographies that detail the contract defining the interactions among

[1] Kubernetes is an open-source container-orchestration system (https://kubernetes.io/).

different microservices to accomplish tasks [8]. Similarly, microservice-based systems can be composed through the actions of an orchestrator, which actively coordinates the participating microservices of a macro-architecture that defines the system's functionality [21]. Sampaio et al. [29] also consider the placement of microservices across the infrastructure as a way to improve overall system performance and resource utilisation.

These two groups of works aim at exploiting mechanisms to adapt microservice-based systems either by changing their macro-architecture or by externally acting on individual instances of microservices. Either way, the microservice internal composition remains the same. Our approach differs as we propose to create microservices that are able to evolve their micro-architecture (i.e., their internal implementation). Enabling a microservice's internal architecture to autonomously evolve allows it to cope not only with changes in the workload volume, but also, and more significantly, with changes in the workload pattern. For instance, the pattern of requests to a given microservice may change to retrieve one kind of information instead of another (e.g., a large list instead of a single element), or to trigger a certain kind of data-processing more often than another (e.g., CPU-bound instead of I/O-bound).

Current approaches to adapt the internal implementation of microservices rely on the role of DevOps [13], which has gained popularity due to the importance of continuous delivery. Particularly, in [31, 16, 6], the authors advocate that microservice-based architectures assist DevOps practices and enable engineers to respond to changes. However, this process remains highly human-dependent and slow, especially considering the development part. In practice, developers handle changes by manually (and offline) creating new microservices and adjusting/maintaining existing ones, while operators automate the deployment process through the use of containers, container-orchestrators and scripts. Our approach goes beyond the current trend of automating DevOps practices [34, 33]. Instead of focusing on the infrastructure, we act at the application level, focusing on automating the evolution of the microservice's own internal architecture as they execute. Thus, EM promotes the evolution and change of microservice implementations at runtime, giving more flexibility to the system and freeing engineers from the need to handle low-level evolution.

Finally, we compare our work to other concepts: classic Autonomic Computing [17], Emergent Middleware [4], and Emergent Software Systems [25]. Although our work can be considered part of the autonomic computing research agenda, the EM approach to evolve a microservice's internal architecture is not based on predefined, manually-crafted rules and models that guide software adaptation. Rather, an emergent microservice learns at runtime, by applying reinforcement learning algorithms, which composition is more suitable for the current workload pattern, without relying on predefined domain-specific information. Similarly, EM radically differs from the Emergent Middleware concept as its goal is not to synthesise connectors to overcome interoperability problems on-the-fly. Instead, EM tackles the online performance optimisation problem by learning, from an existing set of components, which composition yields optimal performance. To conclude, the concept of Emergent Software System (ESS) has been used to enable autonomous evolution and adaptation of web server architectures serving static HTML-based files, a different application domain with different requirements. In this work, on the other hand, we apply ESS as the main method for autonomous software adaptation and evolution within the internal architecture of microservices, resulting in a novel concept named Emergent Microservice (EM). We also concentrate our efforts on investigating EM performance in elastic environments.

## 3. Emergent Microservices

This section describes the architecture, development and operation of emergent microservices, along with their online learning process, which are key aspects in realising the approach. Specifically, it describes the basic internal architecture of an emergent microservice, with components for business logic and general-purpose non-functional properties, which in turn create a search space over which the online learning process executes. We also present the development and operation tasks that are required to build and deploy emergent microservices on elastic operating environments. Finally, we conclude the section by providing details on the online learning process that guides the search for the most suitable microservice internal composition for each identified workload pattern.

### 3.1. Internal Architecture

The internal architecture of emergent microservices is depicted in Fig. 1 (a). It is divided into three main parts. The first part is the Web server, which is represented by components *ws.core* and *Dispatcher*. These components are required as part of any Web-based application. They implement the HTTP protocol and are able to handle incoming HTTP requests, forwarding them to be processed by the appropriate functions of the microservice.

The second part represents the microservice core, which comprises *business logic components* and *utility components*. The former implement the microservice functionality. Each incoming request handled by the Web server is forwarded to a function defined in one of these components. The latter, in turn, represent a library of generic utilities (such as parsers, sorting algorithms, data structures and database connectors) that assist in the creation of a microservice's business logic. Note that, while utility components are highly reusable, business logic components, are reused less often as they capture aspects that are specific of each microservice.

The above two parts are present in any microservice internal architecture. The third part, however, is specific to emergent microservices. It consists of *performance tuning interceptors*, which are designed to intercept function calls stemming from the Web server to the business logic components, adapting the non-functional concerns of the

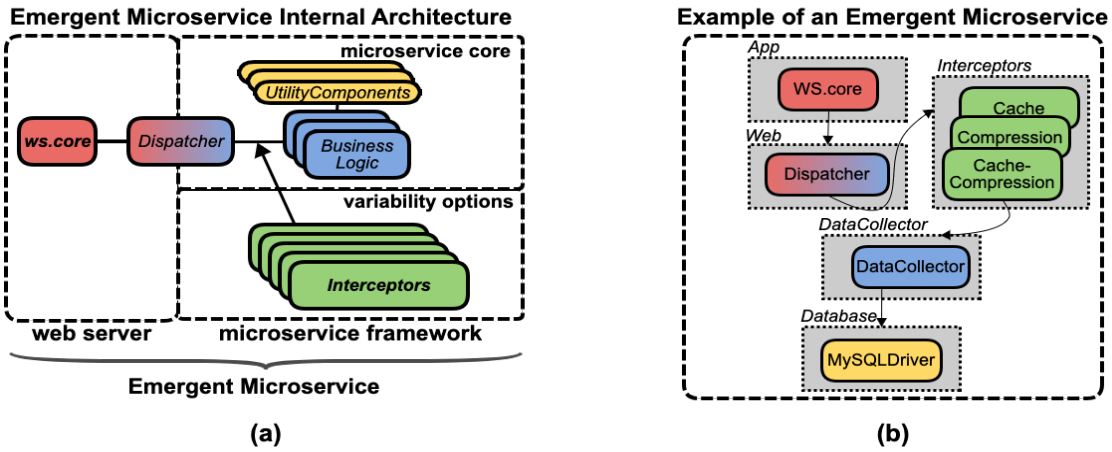


**Figure 1:** (a) Internal architecture of an emergent microservice, showing the main components and how they connect. (b) Generic example of an emergent microservice, with its functional components and examples of variability options.

microservice. An example of interceptor is *Cache*, which can be inserted between two components to cache content exchanged in function calls. This can enable multiple executions of time-consuming functions to be avoided once a return value is already cached, thus decreasing the system's overall response time. Interceptors are application-independent and are autonomously added/removed/replaced by the system to dynamically experiment with the non-functional concerns. Thus, they serve as the primary variability options to compose emergent microservices.

In addition to interceptors, our microservice framework admits two secondary sources of variation to form the search space for emergent microservice composition: alternative implementations of utility components, such as search or sorting algorithms that exhibit different performance for different inputs; and alternative implementations of business logic components that can, e.g., use different query strategies to interact with external data sources. In this paper, however, we only explore the primary kind of variability option (i.e., interceptors) as they are more directly related to handling the kind of workload changes that we target.

### 3.2. Development and Operation

The development and operation of emergent microservices involve actions from both DevOps (engineers) and the emergent systems framework (machine). Although there are minor differences, the overall process is largely similar to developing and operating regular microservices.

DevOps are responsible for: *i)* implementing the components that form the business logic of the microservice, as well as the utility components and the interceptors; *ii)* selecting and placing, in a specific folder, all components that might be used to compose the microservice (considering all available variations); *iii)* packaging that folder in a container; and *iv)* strategically annotating the components where performance tuning interceptors are to be inserted.

Developing the business logic components themselves is no different from the usual development task, including the reuse of utility components. The key is to connect the business logic components to the framework. The connection is done by implementing the component that provides the *ws.Web* interface, illustrated as the *Dispatcher* component in Fig.1. The *ws.Web* interface is used by the *ws.core* component to forward requests to the application running on the Web platform. By implementing the *ws.Web* interface, the *Dispatcher* forwards specific requests, based on their URI, to the appropriate components.

The required utility components are typically picked from an existing library. Interceptors, on the other hand, are often written specifically to operate with particular interfaces, such as *ws.Web*, as they take into account the semantics of those interfaces. Nevertheless, once implemented, interceptors are generic to all uses of the interface; for instance, interceptors for the *ws.Web* are reusable across different microservices.

Once the microservice's key components have been developed, the role of the operator is to select the components that will be used by the emergent systems framework (machine) to autonomously compose the microservice at runtime. This involves the use of widely adopted container technologies to package the selected components, as well as the use of container orchestration systems to automate microservice deployment. Note that a key part of an emergent microservice is the ESS framework itself, which is also packaged in the container, as shown in Fig. 2. The resulting microservice composition, such as depicted in Fig. 1 (b), is thus autonomously assembled and used as a basis for online experimentation to locate optimal compositions at runtime, as described next.

### 3.3. Internal Architecture Evolution

Evolution of the internal architecture of emergent microservices is led by an online learning process that takes place once the container with the microservice starts executing. This enables emergent microservices to learn, at runtime, the most suitable internal architectural composition for the current workload. The learning process is performed

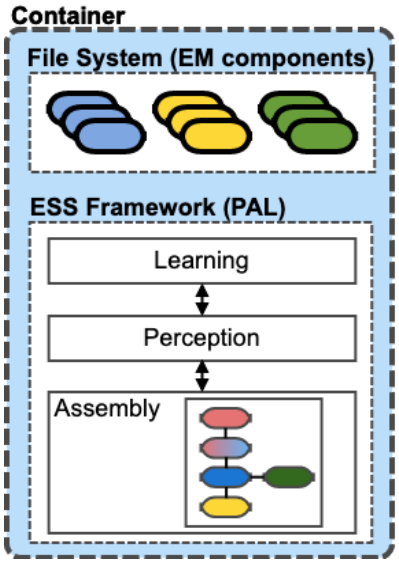


**Figure 2:** Container with an emergent microservice.

by an intelligent agent that is implemented at the top layer of the ESS framework, which is called Perception, Assembly and Learning (PAL) [23], as shown in Fig. 2.

PAL is a general-purpose framework to create emergent software systems that has been implemented in the Dana programming language[2] [24]. PAL is a feedback control loop different from the well-known MAPE-K [2]. PAL does not have a planning phase nor start with predefined knowledge of the systems application domain. Instead, PAL continually samples information from all available system compositions as it builds its understanding of the system and its underlying operating environment before converging to the composition that better satisfies its goals (e.g., improve or maintain system's performance). Its *Assembly* module is responsible for loading components from a file system into memory and connecting them to form functioning systems. This module, using features of the Dana runtime, is also able to perform runtime adaptation for both stateful and stateless components, covering all possibilities and ensuring seamless runtime adaptation and service continuity [24]. The *Perception* module is responsible for collecting information from the running system. Such information is used to characterise both the system's performance (e.g., response time) and the incoming workload (e.g., requests per second). Finally, the *Learning* module interacts with the *Assembly* and *Perception* modules, respectively, to change the microservice composition and to collect information about the execution of a particular microservice instance. It works as an agent and leads the composition and evolution of the microservice.

For that, a reinforcement learning algorithm [30] is used. The *Learner* starts its execution with no predefined domain-specific knowledge, and balances between exploration and exploitation phases to locate the optimal microservice composition for any identified workload pattern. In detail, the *Learner* starts by composing and executing an initial microservice internal composition. This process is performed by interacting with the *Assembly* module, which loads the Web server components, the business logic components and the utility components, and connects them together. Afterwards, a collection of possible microservice compositions is discovered (also by the *Assembly* module) by locating the interceptors. At this point, the *Learner* has a list of possible microservice compositions, each of which containing an interceptor inserted between the *Dispatcher* and *Business Logic* components. Once the microservice is executing and handling incoming requests the learning process begins. The learning process has two main goals: *i*) to classify workload patterns, and *ii*) to locate the best performing microservice composition for the identified workload. Note that changes to the microservice composition are driven by the *Learner*, which interacts with the *Assembly* module to remove or add interceptors. This adaptation of the internal composition happens at runtime with no downtime for the service.

There are many learning algorithms for realising emergent software systems. In this paper, however, we apply a baseline algorithm that was investigated in [25]. The algorithm consists of exploring all available compositions at least once while classifying the environment and determining which composition best suits the observed environment. In detail, the algorithm has only one fixed exploration phase, where the agent tests all available compositions. The agent tests the available compositions by changing the microservice internal architecture from one composition to another. After changing the microservice to a specific composition, the agent waits for it to run for a specific amount of time (the observation window), collecting performance metrics and measurements to characterise the current workload. Based on the collected information from all microservice compositions, the agent is then able to classify the workload patterns and locate the optimal microservice composition for each one. After this exploration phase, the agent selects the composition that yields the best performance metric. As the algorithm has a fixed exploration phase, it helps the integration with other autonomic systems without the need to apply a specialised coordinator to integrate them.

## 4. Single-case Mechanism Experiment

Our single-case mechanism experiment is based on a microservice-based platform to support smart city applications, called InterSCity [10]. The platform aims at providing a highly scalable service layer that abstracts the interaction with city infrastructure devices, helping applications to collect data from sensor devices, and send commands to actuators. It is an open source project and is accessible via the project website[3].

InterSCity has a total of six types of microservices with specific roles. *Resource Adaptor* is the microservice

[2] Dana is a multi-purpose programming language that implements a state-of-the-art component model and supports seamless runtime software adaptation as required for emergent software systems (https://projectdana.com).

[3] InterSCity Project website: http://interscity.org

responsible for receiving all incoming requests from devices, redirecting those requests to the appropriate microservice. *Resource Catalog* is responsible for storing information about all devices available in the city. *Resource Discovery* provides information about specific devices. *Data Collector* is responsible for providing applications with access to all data collected from the city devices, as well as to provide the devices with a gateway to push their collected data. *Actuator Controller* is the microservice that provides access to the city actuator devices. Finally, *Resource Viewer* is responsible to show visual representations of available city resources.

For this paper, we focus on a specific InterSCity microservice, namely *Data Collector* (DC), which is key to the platform's scalability and provides a suitable example of how autonomous microservice evolution can improve performance in the face of workload changes. Specifically, we wrote a new version of the DC microservice using the Dana programming language and applying the concept of EM.

This emergent version of the DC microservice offers the same functionalities of the original one, which entails:

1. Providing access to all historical data stored in the database;
2. Providing access to historical data of a specific resource;
3. Providing access to the latest data collected from all existing resources;
4. Providing access to the latest data collected from a specific resource; and
5. Receiving data from resources and storing them into a database.

In addition, the Emergent DC has the ability to change its internal composition to provide its functionalities using three different performance tuning interceptors.

As previously described, performance tuning interceptors are transparently inserted into an emergent microservice's original internal architecture to autonomously generate composition variants. For the DC single-case mechanism experiment, the interceptors are: **Cache**, **Compression** and **Cache- Compression**. Based on these components, the microservice internal architecture can be assembled into four architectural compositions: **Default**, **Cache**, **Compression**, and **Cache-Compression**.

The **Default** composition is the vanilla microservice, with no interceptors, so that the microservice runs as it was originally intended (with only its business logic and utility components). The **Cache** composition differs from the **Default** one by adding a cache interceptor module to the microservice internal architecture, so that, for each incoming request that is cacheable (i.e., whose HTTP header sets the permission to be cached), the microservice checks whether the request's response is already cached; if it is, then the microservice returns the response from the cache; otherwise, it processes the request as it normally would, sending the response to the client and then caching it for future requests.

The **Compression** composition adds a compression interceptor to the microservice, compressing responses before sending them to the client. Particularly, this interceptor works according to the compression information detailed in the HTTP header. In the request header, the client can choose whether or not to accept compressed data from the service, and also can define which compression algorithm the service should use. Thus, the compression interceptor does not compress the response in case the client does not accept compression, or when the required compression algorithm is not the one it provides. Finally, the **Cache-Compression** composition adds both the cache and compression interceptors to the microservice internal architecture, so that responses are both cached and compressed. This composition compresses the response and then stores the compressed response in a cache. Subsequent requests to the same resource will thus result in cache hits, and the system will transmit the compressed response directly from the cache to the client.

## 5. Evaluation

In this section, we show that Emergent Microservices optimise their internal composition to better perform when subjected to a wide range of different workload patterns. First, we characterise the workload patterns that we use in the experiments, showing that for different workload patterns different internal compositions have better performance. We also show that emergent microservices operate well within the kind of deployment environment in which classic microservices are typically deployed, and that they coexist with commonly used tools of the microservice software ecosystem without negative side effects. In particular, we demonstrate that emergent microservices can synergistically operate with autoscalers, often even reducing the amount of resources necessary to handle sudden increases in workload volume.

Our goal is to discuss and provide first answers to the following evaluation questions, used to design the set of experiments we present later on in this section:

EQ-1 Is there a clear benefit in adapting a microservice's internal architecture?

EQ-2 Can the baseline learning algorithm used in the current implementation of the EM approach learn the best composition for different workload patterns?

EQ-3 Can an off-the-shelf implementation of EM coexist with supporting tools of the microservice ecosystem?

All experiments were executed at least 5 rounds. The depicted results on all graphs is an average of all rounds. Moreover, all experiments were conducted on *Google Cloud*. We created a cluster located in the *us-central* region, with 8 nodes managed by Google Kubernetes Engine (*GKE*). Each node was running the *GKE* standard Ubuntu image, with 2vCPUs, 4GB of memory and 100GB of storage. The Ubuntu image comes with *NFS*, *GlusterFS*, *XFS*, *Sysdig*,

*Debian packages*, and *Docker* installed. The source code that we used to generate the results is publicly available as open-source in the paper's companion repository[4], along with a guide to reproduce the experiments and the results obtained from running the emergent version of the Data Collector microservice. Finally, due to the use of a public cloud platform for conducting experiments, we have noticed small fluctuations in the measured response time when executing experiments on different days and at different times. However, these changes were not significant, and they do not compromise the results depicted in this paper in any way. This is because although the measured response time collected fluctuates slightly, the graph trends and final results remain unchanged.

### 5.1. Workload Patterns and Ground Truth

As described in Sec. 4, we are using the Data Collector microservice to experiment with the proposed concept of Emergent Microservices. In this section, we look into the different characteristics of typical Data Collector workloads. We then describe the workloads that we used in the experiments.

Data Collector is the microservice responsible for providing access to data collected from city devices to InterSCity applications. All request patterns that make up any workload for this microservice have four common dimensions: content type, cache-control, response size, and request entropy. Three of these dimensions may have their values changed in the requests by clients, forming a different workload pattern. Next, we describe each of these dimensions and the values we used to create 18 unique workload patterns for the experiments.

*Content type:* The content type dimension of the workload defines the type of data that the microservice returns to clients. In the case of DC, the content type is *text* formatted in JSON. This means that, for every request it handles, DC returns an HTTP response with all its contents in JSON-formatted text. This dimension is the only one that remains unchanged in all incoming requests. It was predefined by the InterSCity project when defining the API of the microservice.

*Cache control:* refers to the cache-control parameter set by the client in the HTTP request to define whether or not the response should be cached and, if cacheable, how long it should be kept until the item becomes stale. All requests to DC considers the cache-control value set by the client. For the workloads that we created to test the emergent DC, the values for cache control were set to range from 'no-cache' (when items are not cacheable) to 'max-age=1' (items are cached for 1s) and 'max-age=5' (items are cached for 5s).

*Response size:* refers to the size of the returned data upon a request to DC. Depending on whether or not the request is for historical data and the length of the period requested, the size of the returned data can be considerable. For the purpose of our experiments, this dimension assumes the following values to form different patterns: ≈168 bytes (small),≈14KB (medium) and ≈116KB (large).

*Request entropy:* refers to the degree of variability in a sequence of requests sent to the microservice, and assumes one of two values: *high* and *low*. Low request entropy means that the sequence of requests is for the same small set of data, whereas high request entropy means that distinct requests are for distinct data. Therefore, this dimension directly affects the data locality principle, with a significant effect on caching.

All synthetic workloads were created to have unique combinations of the values for these dimensions. Thus, from the possible values of the three dimensions that vary, a total of 18 distinct workload patterns were created ($3x3x2$). Although these patterns are synthetically made with predefined client programs that implement specific user behaviour, some of these patterns match scenarios that we expect to find in real-world workloads for DC in the InterSCity platform.

For instance, if we consider, as a case example, an application that allows users to track a specific bus in real-time, the generated workload will have a well-defined pattern, characterised by having: content type as text (i.e., bus location coordinates are returned to the client in JSON format); cache control set to either 'no-cache' or '1s' (since the user wants the latest collected data, with little tolerance to staleness); small response size (i.e., only the latest bus location); and high request entropy (i.e., every request is for newly collected data).

Another example of a real-world workload pattern is for applications that aim at analysing historical data from a set of bus lines. In this "bus management" scenario, although content type remains text-based (as bus line information is sent as text, e.g., bus locations during a period, or the number of buses running the line), the values for the other dimensions are remarkably different: cache control is set to '5s' (since the desired data is historical and, thus, less likely to be changed); response size is either medium or large, depending on the length of the period the client is interested in (e.g., a month of data, or only last week); and request entropy is low, considering the focus on a specific small set of bus lines.

Fig. 3 depicts 12 out of the 18 workloads we experimented with, showing how the four DC compositions perform when subjected to these distinct workload patterns. The figure is divided into four parts: the graphs depicted in the upper part of the figure (3.a and 3.b) were produced with the response size set to medium (≈14KB), whereas in the graphs at the bottom (3.c and 3.d) the response size was set to large (≈116KB). In turn, the graphs on the left hand side (3.a and 3.c) depict workloads with low request entropy, whereas the graphs on the right (3.b and 3.d) depict workloads with high entropy. The cache-control dimension of the workloads is depicted on the $x$-axis on each graph, showing the different

[4]The companion is available at: https://github.com/robertovrf/em

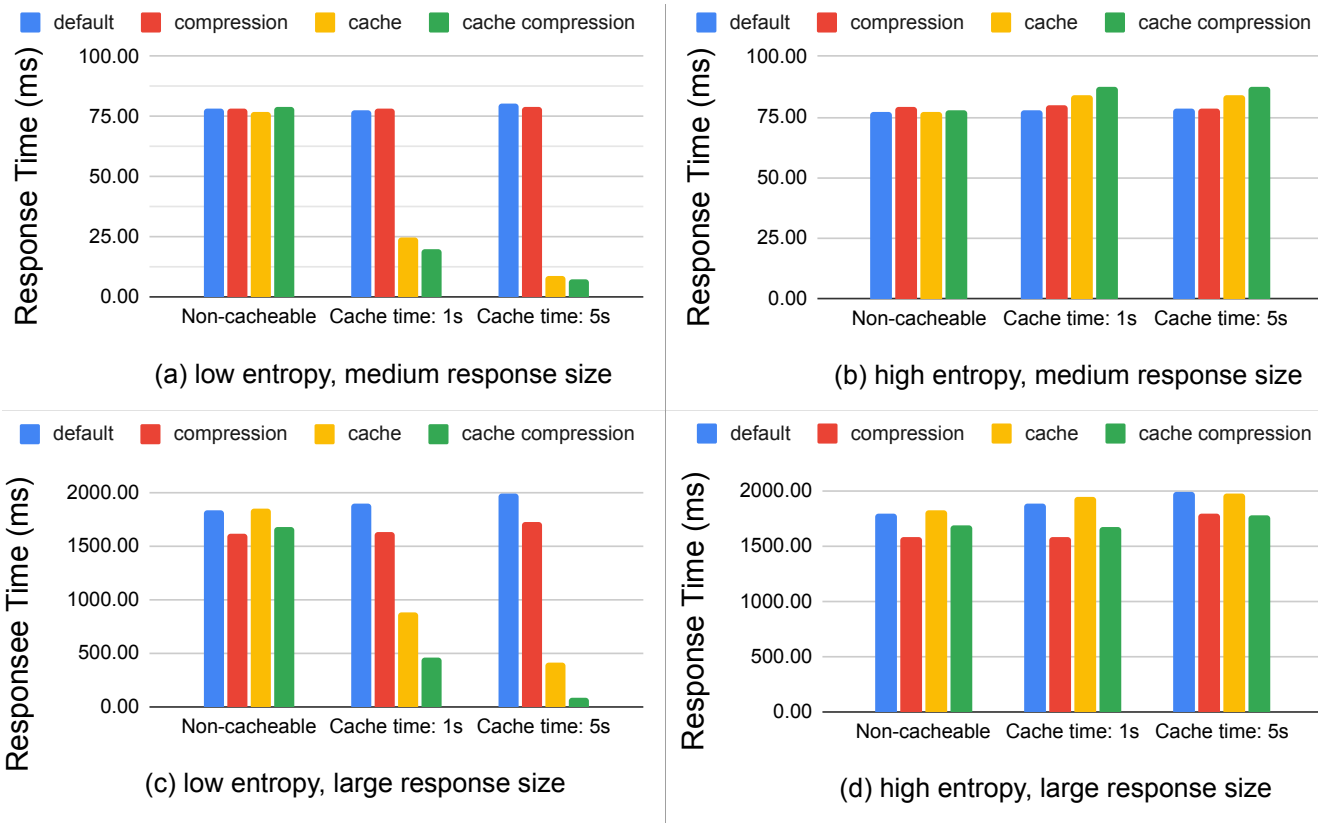


**Figure 3:** Average response time for each available variant of the microservice when exposed to different request patterns. Request patterns are characterized in terms of entropy level, response size, and response cacheability.

DC compositions exposed to cache-control set to *no-cache*, 1s, and 5s. Note that the six remaining workloads, which are characterised by a small response size (≈160 bytes), were not included here as they introduce no significant differences compared to the workloads with medium response size. For the sake of completeness, they have been included in the companion.

To generate the graphs depicted in Fig.3, we exposed the four static compositions of the DC microservice (default, compression, cache, and cache-compression) to all the 12 selected workload patterns. The results show that for different workload patterns different microservice internal compositions yield different performance. More specifically, for the medium response size and no caching (first set of bars in Fig. 3.a and 3.b), all DC compositions have similar performance with variances 1.6 for default, 0.9 compression, 0.7 cache and 3.0 cache-compression. For low request entropy and cache-control set to either 1s or 5s, the cache compression composition has the best overall performance (Fig. 3.a and 3.c), and the default composition often has the worst performance. In turn, for medium response size, high request entropy and cache-control set to 1s and 5s, the default composition yielded the best overall performance (Fig. 3.b), whilst cache-compression had the worst. Finally, for high entropy and large response size (Fig. 3.d), compression is slightly better, though not really standing out from the others. This result shows that, although cache-compression has the best performance on the majority of workload patterns, there are some workload patterns where other compositions have better performance. Therefore, we conclude there is no overall best composition for all workload patterns. We revisit similar workload patterns in the experiments conducted in Sec. 5.3.

The charts in Fig.3 will serve as ground truth to verify whether or not our emergent DC implementation can determine, at runtime, the best performing internal composition with no human interference nor predefined domain-specific knowledge. In our experiments we found that cache-compression is the composition with wider applicability with respect to better performance under different workload patterns. However, we also demonstrate that, for some workload patterns, the DC microservice demands a different composition (other than cache-compression) to perform better. It is essential to note, though, that the EM approach does not use any such previous information on how the DC microservice performs under distinct workload patterns.

The results presented in this subsection provide an answer to EQ-1. Fig.3 clearly demonstrates that, for different workload patterns, there is a different best-performing internal architectural composition for the microservice. Some of these compositions are not obvious choices to perform well when subjected to a given workload pattern. For example, we do not expect cache to perform well under workload patterns with high entropy. However, in scenarios where the

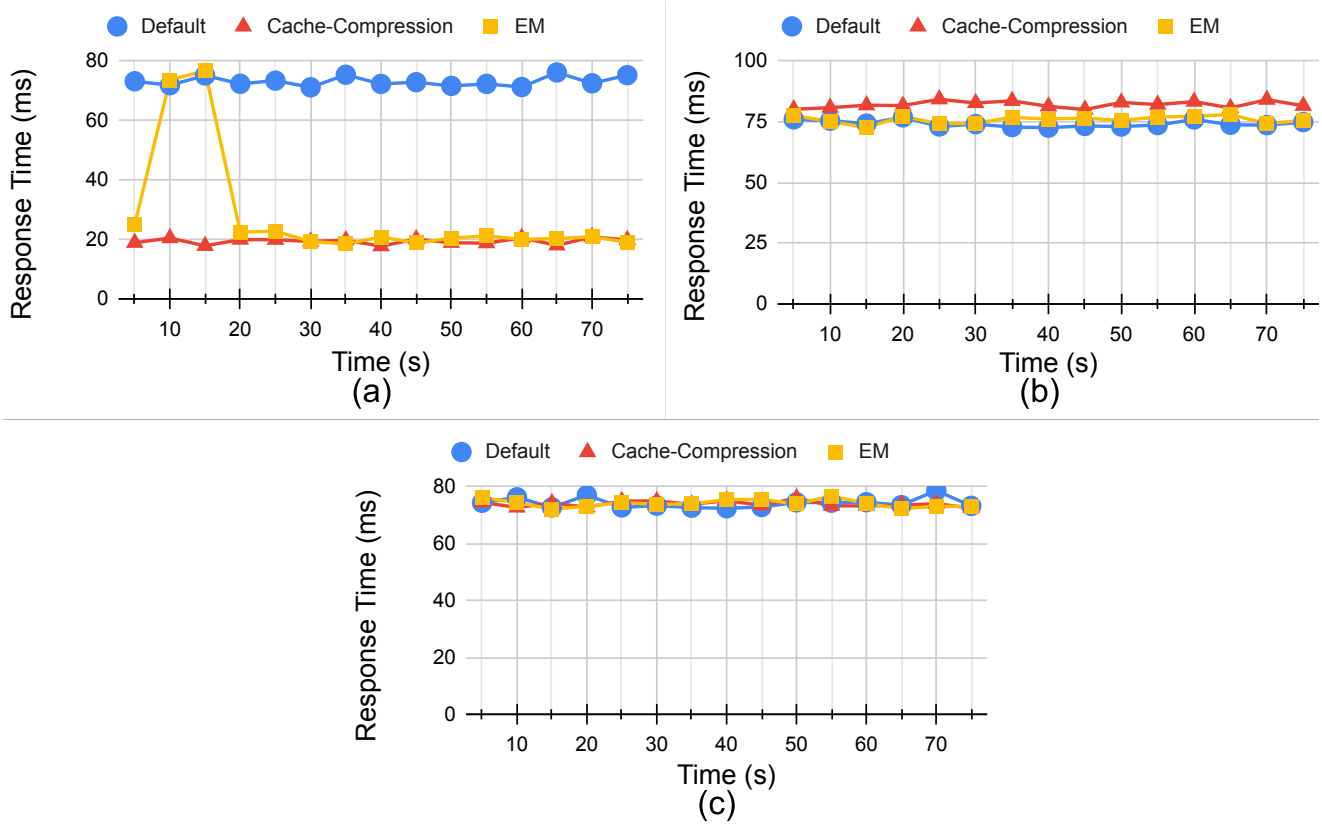


**Figure 4:** EM convergence towards the best composition (in terms of response time) under three distinct workloads. (a) EM (yellow line) converging towards cache-compression (red line). (b) EM converging towards the the default composition (blue line). (c) All available compositions have the same performance and the EM may converge towards any of them.

response size is very large, the negative effect of caching is overcome by the positive effect of compression, making cache-compression (or rather just compression) a viable composition for handling workloads with large response sizes (Fig. 3 (d)). Therefore, considering the upfront effort of carefully analysing every workload pattern for every microservice created, the proposed EM solution, capable of finding the best composition at runtime, is justified. Note that, as self-evidenced by the graphs in Fig. 3, the DC allows us to observe the effectiveness of changing the internal composition with workload patterns that demand only two distinct best-performing microservice compositions (default and cache-compression). In future work, we aim to further characterise the ground truth and further elaborate on the answer to EQ-1 by using microservices that feature a larger number of variations (e.g., by using CPU-bound, as opposed to I/O-bound, microservices).

## 5.2. EM Learning

This section describes two important results: *i*) a demonstration that an emergent microservice converges towards the optimal microservice composition without using any predefined knowledge; and *ii*) a demonstration that emergent microservices learn the internal architectural compositions that are optimal for the distinct workload patterns, without the need to explore the available compositions again.

The first set of experiments aims at (*i*) and evaluates the ability of an emergent microservice to experiment with different compositions at runtime and compare them to find the best performing one. The second set of experiments aims at (*ii*), showing the ability of emergent microservices to recognise workload patterns and immediately adjust their internal composition to a known architectural option. Both results provide an answer to EQ-2.

Based on the 18 unique workload patterns that we have explored, the experiments in Sec. 5.1 enabled the identification of three subsets of workload patterns that exhibit similar behaviour: a subset for which the cache-compression composition stands out; a subset for which the default composition is better; and one workload pattern for which all compositions show similar performance, meaning that any of them can be chosen. Thus, we use the two best-performing compositions as a reference to evaluate the convergence of the emergent microservice. Fig. 4 shows the results of three experiments to compare the response time of the emergent DC microservice (yellow line), with that of two static versions of DC (cache-compression, shown in red, and default, in blue). Each graph shows the average response times, after running the experiment 5 times for each microservice across a period of 75s. In each of the three experiments we use a workload pattern that is representative of one of the above-mentioned subsets. The experiment aims at showing that the

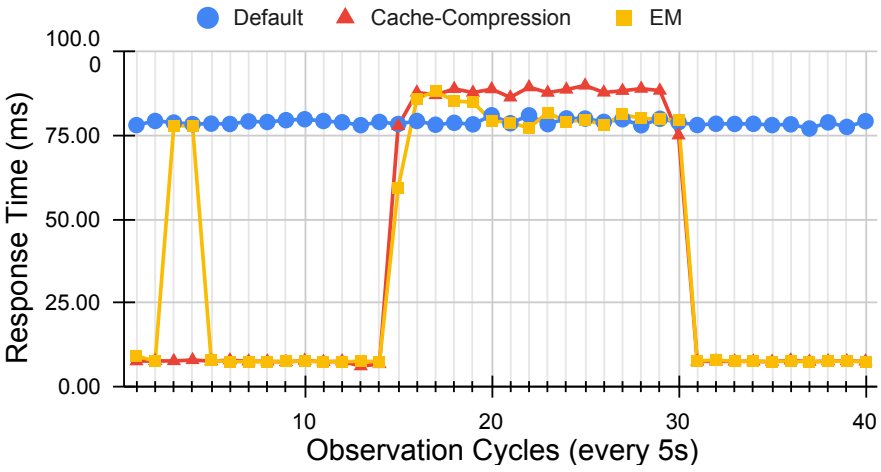


**Figure 5:** Average response time for three versions of the same microservice when subjected to two distinct workloads (A and B). EM learns and converges to the best performing composition for both Workload A (cache-compression) and Workload B (default). At the 6th cycle, EM identifies a previously seen workload pattern and is able to immediately adapt its composition.

emergent microservice is able to adapt its internal architectural composition to learn, at runtime, the most suitable composition for the current operating environment.

For all workload patterns used in the experiment, the EM managed to converge to the best performing composition. The learning algorithm exposes all available compositions during execution to learn which composition yields the best performance (i.e., the lowest response time). This is the exploration phase of the learning algorithm. The exploration phase has a fixed time span determined by the number of available compositions (in this case four) multiplied by the size of the observation window (5s) resulting in 20 seconds total. In workload patterns for which there is a large difference in response times among the different architectural compositions, the result of the exploration phase is clear, as noted by the sudden spike in the yellow line, followed by a drop, during the first observation cycles, shown in Fig. 4(a). This means that, after exploring the two compositions, the emergent microservice converges to cache-compression. Although the experiments shown in Fig. 4(b) and (c) also go through the exploration phase, in Fig. 4(c) the difference in response times is insignificant, meaning that the emergent microservice may rightfully converge to any of the compositions. In Fig. 4(b), in turn, although the spikes have been smoothed down by averaging over multiple executions, the emergent microservice can still converge to the slightly better composition (the default one in this case).

The result of the second experiment is shown in Fig. 5. As in the previous set of experiments, the graph shows the response time of three microservices: fixed with the default composition (blue); fixed with the cache-compression composition (red); and the emergent version (yellow).

The graph shows two distinct workloads (Workload A and Workload B). Each microservice instance was first exposed to Workload A for 15 observation cycles ($\approx$ 75 seconds), then there was a complete change in the workload pattern as the microservices were exposed to Workload B for another 15 cycles. Finally, all microservices were suddenly exposed again to Workload A for 10 cycles. Fig. 5 shows the emergent microservice converging to cache-compression when exposed to Workload A. Then, as the workload changes to Workload B, the emergent microservice detects it and starts learning the new workload pattern. After four cycles (i.e. 20 seconds), it learns the best composition for the new workload pattern. At the end, when the workload pattern changes to the previously seen Workload A, the emergent microservice immediately (i.e., without having to go through the exploration phase again) reassembles its composition to match the most suitable one for the previously seen pattern.

Two distinct spikes on the yellow line, which represents the emergent microservice response time, are visible. The spikes happen at the beginning of Workload A and at the beginning of Workload B. These spikes show the EM learning algorithm executing. During this time, the learning algorithm tests each of the available compositions under the current workload to discover which composition has the best performance. After exploring the compositions, the learning algorithm converges towards the best performing composition for the identified workload pattern, storing this information in a table in case it needs to remember it in the future.

The results depicted in Fig. 4 and Fig. 5 provide an answer to EQ-2. In both figures, we show that for different workload patterns EM is able to locate which architectural composition is the most suitable. We also show that it learns and remembers which composition works best for an identified workload pattern. Despite having limitations [25], the baseline learning algorithm, with its static exploration and exploitation phases, works well for the DC microservice compositions and for the considered workload patterns. It also works well whilst classifying workload patterns.

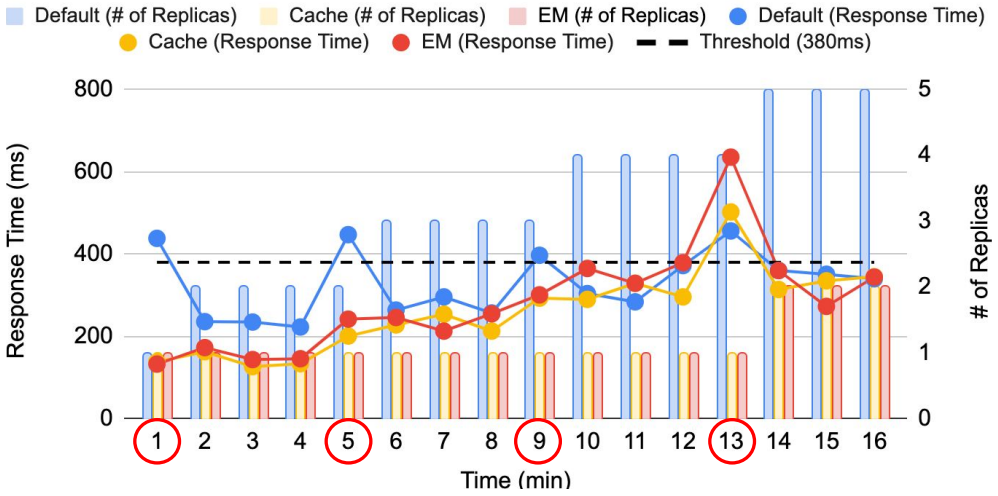


**Figure 6:** Microservices executing in an elastic deployment environment. Default microservice (blue line) starts with poor performance (above the threshold) and keeps triggering replication as the workload volume increases (red circle on the x-axis). EM converges towards the best performing composition (cache – yellow line) maintaining its response time below the threshold and creating fewer replicas.

### 5.3. EM Interaction with Horizontal Autoscaler

This section describes an experiment that shows the potential benefit of deploying emergent microservices in an elastic infrastructure. The goal is to compare the behaviour of static against emergent microservices when deployed in an elastic operating environment, and to demonstrate that emergent microservices can coexist with horizontal autoscalers that are part of the microservice ecosystem, with no explicit coordination between the emergent microservice and the autoscaler tools.

To realise this experiment, we set up a horizontal autoscaler to act upon individual instances of microservices. The autoscaler was also configured to replicate the microservice when its response time surpasses a certain threshold. We subjected the microservice to two different previously described workloads: Workload A (which benefits configurations with cache) and Workload B (which benefits configuration with no cache). We start the client scripts to generate a workload with pattern A (shown in Fig. 6) and another one with pattern B (shown in Fig. 7) at a low volume. As the time passes, we continually increase the volume.

In the experiments we report in this section, all compositions were executed in a container with the same available resources. All containers, regardless of the microservice's internal architectural composition, were created using the same configuration file, assigning 900 millicores of CPU to each container. Also, we used an emulator of the Horizontal Pod Autoscaler (HPA) tool that implements HPA's exact same replication algorithm[5] to trigger container replication in this experiment. The emulator tool was used to facilitate the integration of custom metrics other than CPU and memory usage, commonly used by HPA, and to facilitate the exploration of alternative policies for container replication in future work.

The first experiment was conducted by executing each microservice in the elastic environment and subjecting it to workload A. The result is depicted in Fig. 6. The graph shows the response time (y-axis), the observation cycles (x-axis), which represent the exact time when the microservices response time was collected, and three lines (blue, yellow, and red) that represent the microservice's response time. The blue line represents the execution of a static microservice in the default composition. The yellow line represents the execution of a static microservice in the cache composition. Finally, the red line represents the execution of the emergent DC. The graph also indicates a horizontal dotted black line at 350 ms response time, which represents the autoscaler threshold. It also shows, red circles indicating when the workload volume increases. Finally, the graph shows three-bar charts that represent the number of instances of each executing microservice throughout the experiment.

The default microservice composition (represented by the blue line) is the microservice that has the highest response time when subjected to the workload. Its response time gets above the threshold multiple times throughout the experiment (every time we increase the workload volume – red circles on the x-axis), which triggers the autoscaler to create more replicas of the service each time. The number of replicas is indicated on the bar charts on the right hand side y-axis, and they ultimately show that, every time we increase the workload volume, we create an extra replica that allows the microservice in the default configuration to work below the predefined response time threshold (380ms). The cache-compression composition has the overall best performance and executes below the threshold from the start until we last increase the volume (up to 25 reqs/s on the 13 min. – x-axis). Finally, as we can see following the red line, the

[5]HPA's replication algorithm: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/

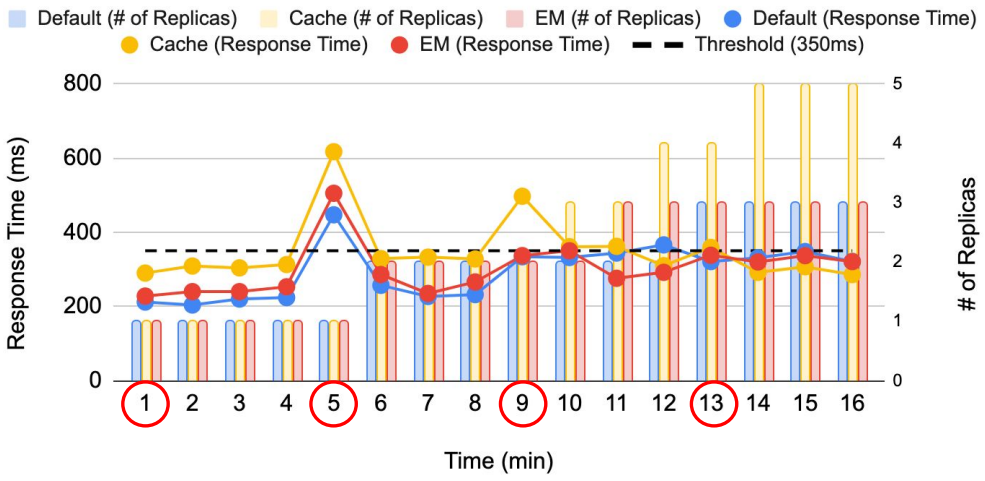


**Figure 7:** Microservices executing in an elastic deployment environment. Cache microservice (yellow line) starts with poor performance and as the workload increases (red circle in the x-axis) it triggers replication. EM converges towards the best performing composition (default – blue line) maintaining its response time below the threshold and creating fewer replicas.

emergent DC converges towards the best executing composition, which in this case is cache-compression (yellow line), and keeps its response time below the threshold, also only triggering replication when we increase the volume to 25 reqs/s at minute 13 on the x-axis.

The second experiment with the autoscaler is illustrated in Fig. 7. Again, the graphs show three versions of the microservice: the microservice with the default configuration, the cache configuration and the emergent DC. We also start the client scripts to generate a different workload pattern, pattern B, which now benefits default configuration, as opposed to the workload pattern in Fig. 6.

Fig. 7 shows the emergent DC now converging towards a different new best composition and, as a result, creating fewer instances of the microservice to maintain the response time below the predefined threshold. In this case, the best composition is the default one. As we increase the workload volume, new instances of the microservice are created from minute 9 onwards, and the number of instances to maintain the cache composition below the threshold increases, reaching the final number of 5 instances at the peak of the workload volume. Meanwhile, for the optimal configuration (default – blue line), at the peak of the workload volume (after minute 13 on the x-axis) the default configuration needed only three instances to handle the workload. As the emergent DC converges towards the optimal configuration, the resulting number of instances is the same to cope with the highest volume the microservice is subjected to. This demonstrates that no matter the workload pattern, the emergent microservice always converges towards the optimal composition.

This experiment addresses EQ-3. We demonstrate that EM successfully addresses workload pattern changes, while autoscalers only address workload volume. We also show that whenever a gap exists between the best performing composition and the others, even if this only happens when the system is subjected to specific workload patterns, the use of EM in tandem with autoscalers can certainly reduce resource utilisation while maintaining systems performance. The results also suggest that the larger the gap between the optimal microservice's composition and others, the larger the amount of resources that can be saved. Also, it is important to mention that the best performing composition is not always obvious. In this particular case, due to previous experiments, we already knew that cache was the best composition for Workload A and default was the best for Workload B, but for a real deployment, this may not be the case. Furthermore, the workload pattern can change mid-execution, which justifies the need for online learning.

## 6. Discussion

In this section, we discuss the limitations of our approach as opportunities for future work. Although the results demonstrate that the EM works well in locating and learning the best available internal composition, the approach, as well as emergent software systems in general, still have limitations [25]. Specifically, we discuss the challenges related to classifying workload patterns, the impact of the variability options for the microservice, further integrating EMs with autoscalers, and online learning.

Workload classification still remains an open issue. Firstly, the features that are used to classify patterns are manually defined in a case-by-case manner. In addition, the current methodology to select the features that best characterise the incoming pattern is based on trial-and-error. This has a direct effect on the quality of learning. Secondly, classifying workload patterns as the learning algorithm runs (to learn which software composition has the best performance) is tricky. The workload pattern is perceived through an executing composition, which may distort the value of some of its features. For instance, if the average

response size over a set of incoming requests is a workload feature, depending on the current microservice composition (e.g., if it employs compression), its value may change even though the actual pattern remains unchanged.

Another aspect of learning that also needs to be carefully considered is related to the actual algorithm. In this work, we used the baseline algorithm described in [25], whose exploration phase happens only once and has a fixed duration. This allows coordination between EM and autoscalers by simply defining different observation windows for the two systems. For the experiments shown in this paper, the autoscaler observation window was set to 30s, whereas the observation window for analysing each microservice composition was set to 5s. Considering that our microservice has a total of four unique compositions, it takes the learning algorithm a total of 20 seconds to learn, thus fitting inside the autoscaler window. This approach has the advantage of simplicity, but may not be appropriate if a more sophisticate exploration phase is needed.

Moreover, the baseline algorithm may lead to negative side effects. Considering that the algorithm makes decisions after experimenting with each composition only once, in operating environments where the reward for each composition fluctuates, this approach has a high probability of converging towards the wrong composition. To handle this problem, algorithms that account for fluctuations on the reward, e.g., as determined by a probability distribution, may be more suitable. On the other hand, such algorithms have a dynamic balance between exploration and exploitation phases, which makes it difficult to coordinate with autoscalers and other elements of the microservice ecosystem. Thus, a future research direction is to investigate stochastic learning algorithms and their interaction with autoscalers.

Another important detail of the proposed approach is the impact a variability option may have on the microservice execution. In our experiments, we used cache components. Caching may positively affect system performance but can also present undesirable effects on data timeliness. If the microservice requires high data timelessness, caching would not be a viable variability option. This has to be carefully considered by the microservice's developers beforehand since our approach has no autonomic mechanism to detect and avoid the selection of compositions that may have undesired side-effects. Moreover, further investigation on the impact of interceptors on the system's performance is an interesting avenue for future work. Our experiments have not suggested any serious side effects of the use of interceptors on the performance of the microservice. However, the application of this approach to a diverse set of microservice implementations and the exploration of higher workload volumes may provide a deeper perspective on how interceptors may affect the microservice performance and/or generate potential negative side effects.

Finally, our online learning approach has its convergence time directly connected to the size of the search space. Considering that we explore all available microservice compositions, by adding new component variants, the search space grows exponentially and so does the convergence time. This problem is out of scope in this paper, but there are some approaches that aim to solve it. For example, Ontanón [22] explores reinforcement learning algorithms with millions of actions to be learned at runtime. Similarly, Donckt et al. [32] use neural networks to reduce the search space at runtime. Both approaches are complementary and relevant to ours.

## 7. Conclusion

We demonstrated that the Emergent Microservice (EM) approach is able to automate the evolution of the internal composition of microservices to quickly and accurately respond to changes in workload patterns. We also experimented with EM on elastic environments, showing that EM can save infrastructure resources when the microservice is autonomously optimised for the current workload.

We applied the approach in a single-case mechanism experiment, conducting a series of experiments in an industrial deployment platform to demonstrate that EM is able to learn the most suitable internal architectural composition at runtime when given a goal such as response time to optimise – a result achieved by combining programmer-supplied business logic with generalised performance tuning interceptors. We also demonstrate that our EM implementation coexists well with other elements of the microservice deployment platform, such as horizontal autoscalers, with no need for special adjustments.

In future work, we will explore the macro level of microservice composition in two major ways. First, we aim to investigate how multiple emergent systems (each modeled as its own microservice) can reach good decisions while learning and adapting at the same time as part of the same global system, so that globally-efficient compositions can be achieved. Second, we plan to investigate how EM coexists with other supporting elements that are often present in microservice deployment platforms. We aim to demonstrate that, as the workload volume increases and its pattern changes, our preliminary result, which shows the emergent microservice interacting with a horizontal autoscaler, can be generalised, thus showing that EM enables near-optimal performance with less overall resource consumption in more complex scenarios. We also aim to explore the interaction between EM and vertical autoscalers, external cache systems, and API gateways, which is important to demonstrate the effectiveness of EM on cloud computing environments.

## Acknowledgments

This research is also part of the INCT of the Future Internet for Smart Cities funded by the National Council for Scientific and Technological Development (CNPq) proc. 465446/2014-0, the Coordination for the Improvement of Higher Education Personnel (CAPES) proc. 88887.136422 /2017-00, and the São Paulo Research Foundation (FAPESP) procs. 14/50937-1, 15/24485-9, and 23/00811-0. Roberto Rodrigues Filho thanks FAPESP for funding his postdoctoral work under the process 2020/07193-2.

## References


[1] Al-Dhuraibi, Y., Paraiso, F., Djarallah, N., Merle, P., 2017. Autonomic vertical elasticity of docker containers with ELASTICDOCKER, in: 2017 IEEE 10th International Conference on Cloud Computing (CLOUD), pp. 472–479. doi:10.1109/CLOUD.2017.67.

[2] Arcaini, P., Riccobene, E., Scandurra, P., 2015. Modeling and analyzing mape-k feedback loops for self-adaptation, in: 2015 IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, pp. 13–23. doi:10.1109/SEAMS.2015.10.

[3] Bauer, A., Herbst, N., Spinner, S., Ali-Eldin, A., Kounev, S., 2019. Chameleon: A hybrid, proactive auto-scaling mechanism on a level-playing field. IEEE Transactions on Parallel and Distributed Systems 30, 800–813. doi:10.1109/TPDS.2018.2870389.

[4] Blair, G., Grace, P., 2012. Emergent middleware: Tackling the interoperability problem. IEEE Internet Computing 16, 78–82.

[5] Brondolin, R., Santambrogio, M.D., 2020. Presto: a latency-aware power-capping orchestrator for cloud-native microservices, in: 2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS), IEEE. pp. 11–20.

[6] Chen, L., 2018. Microservices: Architecting for continuous delivery and devops, in: 2018 IEEE International Conference on Software Architecture (ICSA), pp. 39–397. doi:10.1109/ICSA.2018.00013.

[7] Coulson, N.C., Sotiriadis, S., Bessis, N., 2020. Adaptive microservice scaling for elastic applications, IEEE. pp. 4195–4202.

[8] Dai, F., Mo, Q., Qiang, Z., Huang, B., Kou, W., Yang, H., 2020. A choreography analysis approach for microservice composition in cyber-physical-social systems, IEEE. pp. 53215–53222.

[9] De Lemos, R., Giese, H., Müller, H.A., Shaw, M., Andersson, J., Litoiu, M., Schmerl, B., Tamura, G., Villegas, N.M., Vogel, T., et al., 2013. Software engineering for self-adaptive systems: A second research roadmap, in: Software Engineering for Self-Adaptive Systems II. Springer, pp. 1–32.

[10] Del Esposte, A.M., Kon, F., Costa, F.M., Lago, N., 2017. Interscity: A scalable microservice-based open source platform for smart cities., in: SMARTGREENS, pp. 35–46.

[11] Dragoni, N., Giallorenzo, S., Lafuente, A.L., Mazzara, M., Montesi, F., Mustafin, R., Safina, L., 2017a. Microservices: yesterday, today, and tomorrow, in: Present and Ulterior Software Engineering. Springer, pp. 195–216.

[12] Dragoni, N., Lanese, I., Larsen, S.T., Mazzara, M., Mustafin, R., Safina, L., 2017b. Microservices: How to make your application scale, in: International Andrei Ershov Memorial Conference on Perspectives of System Informatics, Springer. pp. 95–104.

[13] Ebert, C., Gallardo, G., Hernantes, J., Serrano, N., 2016. Devops. IEEE Software 33, 94–100. doi:10.1109/MS.2016.68.

[14] Esfahani, N., Malek, S., 2013. Uncertainty in self-adaptive software systems, in: Software Engineering for Self-Adaptive Systems II. Springer, pp. 214–238.

[15] Jamshidi, P., Pahl, C., Mendonca, N.C., Lewis, J., Tilkov, S., 2018. Microservices: The journey so far and challenges ahead. IEEE Software 35, 24–35. doi:10.1109/MS.2018.2141039.

[16] Kang, H., Le, M., Tao, S., 2016. Container and microservice driven design for cloud infrastructure devops, in: 2016 IEEE International Conference on Cloud Engineering (IC2E), pp. 202–211. doi:10.1109/IC2E.2016.26.

[17] Kephart, J.O., Chess, D.M., 2003. The vision of autonomic computing. Computer 36, 41–50.

[18] Larrucea, X., Santamaria, I., Colomo-Palacios, R., Ebert, C., 2018. Microservices. IEEE Software 35, 96–100. doi:10.1109/MS.2018.2141030.

[19] Leite, L.A., Oliva, G.A., Nogueira, G.M., Gerosa, M.A., Kon, F., Milojicic, D.S., 2013. A systematic literature review of service choreography adaptation. Service Oriented Computing and Applications 7, 199–216.

[20] Mendonça, N.C., Garlan, D., Schmerl, B., Cámara, J., 2018. Generality vs. reusability in architecture-based self-adaptation: the case for self-adaptive microservices, in: Proceedings of the 12th European Conference on Software Architecture: Companion Proceedings, pp. 1–6.

[21] Monteiro, D., Gadelha, R., Maia, P.H.M., Rocha, L.S., Mendonça, N.C., 2018. Beethoven: An event-driven lightweight platform for microservice orchestration, in: European Conference on Software Architecture, Springer. pp. 191–199.

[22] Ontanón, S., 2017. Combinatorial multi-armed bandits for real-time strategy games. Journal of Artificial Intelligence Research 58, 665–702.

[23] Porter, B., Grieves, M., Rodrigues-Filho, R., Leslie, D., 2016. RE$^{X}$: A development platform and online learning approach for runtime emergent software systems, in: Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation, USENIX.

[24] Porter, B., Rodrigues-Filho, R., 2021. A programming language for sound self-adaptive systems, in: 2021 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS), IEEE Computer Society, Los Alamitos, CA, USA. pp. 145–150. URL: https://doi.ieeecomputersociety.org/10.1109/ACSOS52086.2021.00036, doi:10.1109/ACSOS52086.2021.00036.

[25] Rodrigues Filho, R., Porter, B., 2017. Defining emergent software using continuous self-assembly, perception, and learning. ACM Transactions Autonomic Adaptive Systems 12, 16:1–16:25. URL: http://doi.acm.org/10.1145/3092691, doi:10.1145/3092691.

[26] Rodrigues Filho, R., de Sá, M.P., Porter, B., Costa, F.M., 2018. Towards emergent microservices for client-tailored design, in: Proceedings of the 19th Workshop on Adaptive and Reflexive Middleware, Association for Computing Machinery, New York, NY, USA.

[27] Rossi, F., Cardellini, V., Presti, F.L., 2020a. Hierarchical scaling of microservices in kubernetes, in: 2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS), IEEE. pp. 28–37.

[28] Rossi, F., Cardellini, V., Presti, F.L., 2020b. Self-adaptive threshold-based policy for microservices elasticity, in: 2020 28th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS), pp. 1–8. doi:10.1109/MASCOTS50786.2020.9285951.

[29] Sampaio, A.R., Rubin, J., Beschastnikh, I., Rosa, N.S., 2019. Improving microservice-based applications with runtime placement adaptation, SpringerOpen. pp. 1–30.

[30] Sutton, R.S., Barto, A.G., 1999. Reinforcement learning. Journal of Cognitive Neuroscience 11, 126–134.

[31] Trihinas, D., Tryfonos, A., Dikaiakos, M.D., Pallis, G., 2018. Devops as a service: Pushing the boundaries of microservice adoption. IEEE Internet Computing 22, 65–71. doi:10.1109/MIC.2018.032501519.

[32] Van Der Donckt, J., Weyns, D., Quin, F., Van Der Donckt, J., Michiels, S., 2020. Applying deep learning to reduce large adaptation spaces of self-adaptive systems with multiple types of goals, in: Proceedings of the IEEE/ACM 15th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, pp. 20–30.

[33] Vuppalapati, C., Ilapakurti, A., Chillara, K., Kedari, S., Mamidi, V., 2020. Automating tiny ml intelligent sensors devops using microsoft azure, in: 2020 IEEE International Conference on Big Data (Big Data), pp. 2375–2384. doi:10.1109/BigData50022.2020.9377755.

[34] Wettinger, J., Breitenbücher, U., Leymann, F., 2014. Standards-based devops automation and integration using tosca, in: 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing, pp. 59–68. doi:10.1109/UCC.2014.14.