# Towards Emergent Microservices for Client-Tailored Design

Roberto Rodrigues Filho
Lancaster University
Lancaster, UK
r.rodriguesfilho@lancaster.ac.uk

Marcio Pereira de Sá
Federal University of Goiás
Goiânia, GO, Brazil
marcio.pereira@inf.ufg.br

Barry Porter
Lancaster University
Lancaster, UK
b.f.porter@lancaster.ac.uk

Fábio M. Costa
Federal University of Goiás
Goiânia, GO, Brazil
fmc@inf.ufg.br

## Abstract

Contemporary systems are increasingly complex, with both large codebases and constantly changing environments which make them challenging to develop, deploy and manage. We consider two recent efforts to tackle this complexity: microservices and emergent software. Microservices have gained recent popularity in industry, in which monoliths of software are broken down into compositions of single-objective, end-to-end services running on HTTP which can be scaled out on cloud hosting systems. From the research community, the emergent systems concept demonstrates promise in using real-time learning to autonomously compose and optimise software systems from small building blocks, rapidly finding the best behavioural composition to match the current deployment conditions. We argue that emergent software and microservice architectures have strong potential for synergy in complex systems, offering mutually compatible lessons in dealing with complexity via scale-out design and real-time client-tailored behaviour. We explore self-designing microservices, built with emergent software, to demonstrate the complementary boundaries of both concepts – and how future intersections may offer novel architectures that lie at a compelling point between human- and machine-designed systems. We present the conceptual synergy and demonstrate a specific microservice architecture for a smart city example where scoped microservices are continually self-composed according to the demands of the applications and operating environment. For the purpose of reproducibility of the study, we make available all the code used in the evaluation of the proposed approach.

**ACM Reference format:**
Roberto Rodrigues Filho, Marcio Pereira de Sá, Barry Porter, and Fábio M. Costa. 2018. Towards Emergent Microservices for Client-Tailored Design. In *Proceedings of Middleware'18, Rennes, France, December 2018,* 6 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

## 1 Introduction

Distributed systems remain highly complex to design and maintain. This complexity comes from large code bases, a collection of added failure modes, and dynamicity in the deployment environment which makes it difficult to predict how best to optimise a system.

Reducing this complexity for systems developers continues to be a key focus of research in both industry and academia. We draw together two recent trends in this area, *microservices* and *emergent software*, to demonstrate how they offer mutually complementary lessons in addressing aspects of systems complexity.

Microservices encourage the development of single-objective services with strong encapsulation, loose coupling and independent deployment. They operate over HTTP, making them inherently compatible with most existing systems on the web and enabling them to be developed in a multitude of implementation languages. Their stateless design also supports scale-out mechanics naturally, so that automated processes can manage scaling to user demand. However, their outward-facing API specification is ad-hoc, and recent efforts have examined how to offer more explicit interface descriptions – to in turn enable automated composition.

Emergent software systems (ESS), recently demonstrated as a successful approach to reducing complexity in local system building [8], use a concept of continuous self-assembly over a library of small component building blocks. Combined with real-time learning, this automatically derives systems that are functionally correct and constantly improve their performance according to the current characteristics of the deployment environment, by finding alternative compositions of building blocks which maximise a reward function of interest. The building blocks used for emergent software are strongly encapsulated, with their dependencies resolved by an external managing process; because they are designed to build local systems, the building blocks are very fine grained and some of them will carry state that is transferred across the adaptations that occur while improved behavioural compositions are sought.

While microservices represent a clean approach to build distributed applications, their internal implementations are largely ad-hoc, despite recent efforts to adapt model-driven techniques used for traditional SOA [9]. ESS techniques can be used to effectively bridge this gap, providing a natural way to build microservices that are inherently adaptable at a fine granularity level. In turn, microservice architectures may be leveraged by providing a simplified yet effective approach for remote interaction and state handling and by reducing the search space of ESS techniques to the scope of individual microservices. Put simply, their combination enables the effective use of human design at the macro level (to design the microservice architecture of an application) and machine-driven design at the micro level (to create the internal implementation of each required microservice).

Based on these observations of complementarity, we make the following contributions in this paper:


*Middleware'18, Rennes, France*

DOI: 10.1145/nnnnnnn.nnnnnnn

1. We present an approach to building micro-services in an emergent way, to yield a runtime search space of behavioural variation which machine learning can navigate at runtime to tailor the system to the current environment and client behaviour.

2. We discuss how we may best divide responsibilities between a human engineer and the machine's own learning and analysis capabilities, including the separation of business logic from behavioural variation points, and how future work may see the boundaries blurred at a distributed level between microservice composition and automated distribution of emergent systems.

3. We demonstrate a proof-of-concept implementation of emergent microservices using a smart city example, and experimentally show how real-time client-tailored design of individual microservices can benefit the performance of the overall system. We make available all of our source code for download[1] with instructions to reproduce our experiments.

In the remainder of this paper we discuss the relevant background on emergent software systems and microservices in Sec. 2, and related work in Sec. 3; in Sec. 4 we discuss how these two concepts can already be combined in compelling ways and how they may blur the boundaries of human- and machine-designed systems in the future; and in Sec. 5 we present early work on evaluating the combined approach, using a smart city platform and a related application scenario to explore the implementation of emergent microservices.

# 2 Background

## 2.1 Emergent Software Systems

Emergent Software Systems, as described in [4, 8], use continuous self-assembly over a pool of small software building blocks to derive systems that are autonomously composed as a factor of the operating environment and human-provided high-level goals. Starting from no initial knowledge, the ideal composition of behaviour for each range of observed deployment environment conditions is automonously learned using real-time reinforcement learning. The avoidance of pre-defined models or training means that the system basis its decision making purely on what is actually experienced in its deployment setting and how that affects its behaviour.

In order to realise this concept, emergent software systems rely on a framework, illustrated in Fig.1. The framework is composed of three main modules: Assembly, Perception and Learning. The Assembly module assumes a runtime-adaptive component model which explicit dependencies, and starts from a single 'main' component to derive all possible dependency resolutions (recursively) that result in a functionally correct system; this generates a set of valid compositions of behaviour. The Perception module is responsible for monitoring the system's performance status (according to some metrics of interest) and the operating environment. By adding special proxy components in strategic places, the Perception module can extract information from specific parts of the system (such as execution time of certain functions). Finally, the Learning module implements the reinforcement learning algorithm that conducts the learning process as the system executes; it epxeriments with each available composition (causing the system to adapt as necessary to reach that composition) and observes the resulting perception data

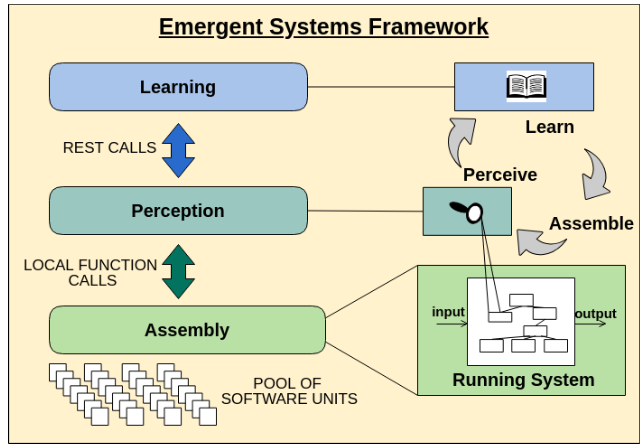


**Figure 1.** Emergent software system framework architecture.

to learn which software composition maximises the satisfaction of the defined metrics under each environment.

One of the key challenges in using emergent systems is that the search space for real-time machine learning can become very large as the scale of the software increases, requiring mitigation strategies such as dividing the system into smaller sub-systems each of which are emergent and contribute to the global picture.

## 2.2 Microservices

Microservices, as an architectural style, can be seen as an evolution in the way of realising Service-Oriented Architectures (SOA) [10]. Each microservice is a cohesive, independent service running in its own process, typically within a container, and communicating via messages (typically via RESTfull HTTP requests or a message queuing system) [3]. Microservices distinguish themselves from traditional SOA services as they are inherently distributed and follow the single responsibility pattern, resulting in focused, finer-grained and loosely coupled services. Furthermore, microservice architectures are typically used to build a single application or system, as opposed to targeting the integration of different applications as in more conventional SOA. A microservice architecture is thus a distributed composition of individual microservices that are coordinated to implement the functional and non-functional concerns of a particular distributed application.

The fact that microservices have bounded context, as defined by the single responsibility pattern, makes them ideal to handle the problem of search space explosion discussed above. Moreover, microservice architectures in which individual microservices are built in an emergent way can be exploited as a straightforward means to extend the benefits of ESS to distributed systems. In turn, ESS techniques represent an effective approach to build environment-tailored and self-adaptive microservices.

# 3 Related Work

The problem of designing and implementing microservices has been extensively targeted both by the industry and academia. The focus has mainly been on the definition of service boundaries (such as by using domain-driven design and data isolation patterns), on the issues of dealing with failure in distributed systems, and on infrastructure issues (e.g., deployment and scalability automation) [7].

[1] All experiments and code from this paper are available at http://research.projectdana.com/arm2018rodrigues

In this section we limit the review to existing proposals that aim to facilitate the flexible design of adaptive microservice architectures.

Microservice architectures are inherently adaptive. They strongly encourage encapsulation, loose coupling, substitutability, and independent deployment. As a result, microservice-based applications can be adapted by dynamically replacing the implementation of individual microservices and by creating/deleting instances of existing ones. Additionally, the use of flexible choreography definition languages, such as Jolie [6], enables adaptation of the distributed coordination protocol, allowing the architecture itself to change as new microservices are added or deleted. Microservice adaptation at this macro level has also been proposed in [5], which provides an approach to rewrite deployment descriptors and adapt the current deployment of a microservice architecture according to new requirements and resource availability.

In [2], models@runtime and component-based software engineering are proposed as part of a vision to apply the principles of continuous software engineering to microservices. Individual microservices are designed and implemented as configurations of components and a runtime model provides a reflective interface for dynamic inspection and adaptation of such configurations. In common with our approach, model-integrating microservices enable both micro and macro adaptations, leveraging a dynamic component model for the former and the properties of microservices (notably encapsulation and independent deployment) for the latter. However, model-integrating microservices rely on the existence of suitable DSLs and modeling tools for expressing and manipulating the component models, in addition to the necessary involvement of humans in the design of the initial component model of each microservice (and, potentially, for its adaptation as well). In contrast, in our approach the component configuration emerges from a set of fine-grained components and system goals, without human involvement and without the need for additional modeling tools.

Finally, microservices are often used as building blocks for constructing large-scale systems. For the purpose of system optimisation, microservices are either replaced, added or removed from the systems composition. In this paper, we concentrate on a fourth option, where the micro-architecture of microservices are themselves adaptive (in a model-free manner, as opposed to [2]), showing a different dimension to be explored in the optimisation process of large distributed systems. For emergent systems themselves, microservices offer a way to extend emergent systems naturally intro large-scale distributed systems by defining boundaries of responsibility as the encapsulation offered by a microservice.

# 4 Approach

This section describes the methodology for creating new emergent microservices, depicting the roles of both humans and participating machines in the process. We describe the basic architecture for a microservice, including our web framework that includes roles for business logic as well as generalised non-functional variational points to create a search space over which an emergent systems process learns. We conclude the section with a discussion of the potential implications of the described approach in future extensions.

## 4.1 Emergent Microservices Anatomy

The anatomy of an emergent microservice is depicted in Fig. 2. The architecture is divided into four main groups of components that have specific roles. The first group is represented in red. These are components that are part of the web service framework and are required to form any web-based application. The second group, coloured in blue, are components that implement the microservice's business logic; these are reused less often and are specific to each individual microservice. The yellow components represent the third group. These are generic and highly reusable components from a standard library of utilities (such as parsers or sorting algorithms), and are reused across many different microservices and other emergent applications. Finally, the fourth group, coloured in green, represent *non-functional requirements proxies* that are designed to operate specifically with the web service framework. These are components that are autonomously added by the system to experiment with non-functional concerns which are generic to the business logic components. This framework offers two main sources of variation to form a search space for an emergent system: variations in utility component, such as alternative search or sort algorithms which have different performance characteristics under different inputs, and variations in non-functional requirements proxies which can be injected into or removed from the system to insert concepts such as caching in a generalised way.

Once these components are placed in a folder known to the emergent systems framework, the machine takes over the design process for the emergent microservice. The resulting architecture, depicted in Fig. 2, is autonomously assembled and used as a base for online experimentation to locate optimal compositions at runtime. Further details on the component groups and on the machine's role in the microservice design are provided in the following sections.

## 4.2 Designing Emergent Microservices

The development cycle of emergent microservices involves actions from both developers and the emergent systems framework (machine). Developers are responsible for i) implementing the components that form the business logic of the microservice's architecture, ii) selecting and placing components in a specific folder (selective deployment task), and iii) strategically annotating components where non-functional requirement proxies are to be inserted.

Developers are entirely in charge of the development process of the business logic components, and the development of these components is no different from the usual development task. The key is then to connect the business logic components to the framework architecture. The connection process is done by implementing the component that provides the *ws.Web* interface, illustrated as the Dispatcher component in Fig.2. The *ws.Web* interface is used by the *ws.core* component (the component that represents the requests to the applications running on top of the web platform). By implementing the *ws.Web* interface, the Dispatcher component forwards specific requests, based on the URI in the request, to the appropriate components.

While implementing business logic components, developers may reuse existing interfaces (and their implementing components) found in the repository. Highly reusable components such as database drivers and data structures are part of this group. These components often have implementation variants that are used by the emergent framework to optimise the system, by finding the right implementation variant for the execution environment – such as alternative sorting algorithms or data parsers.

The last group of components that we introduce are special proxies or interceptors which implement a non-functional concern and

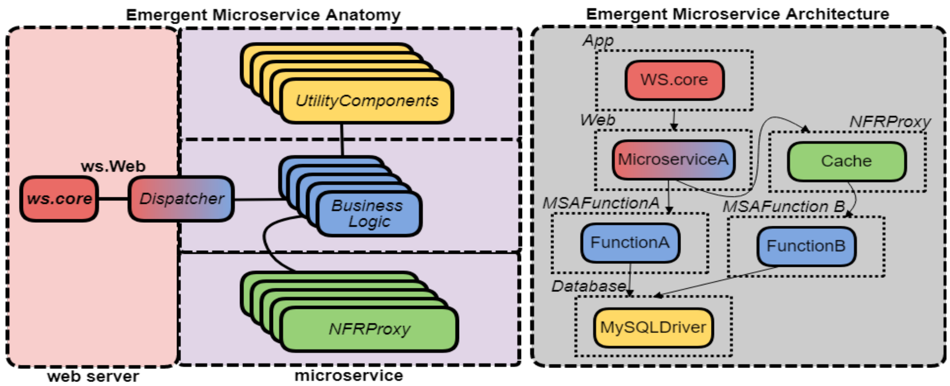


**Figure 2.** Emergent microservice anatomy depicting the reusbale components, business logic, NFRP and the required components, on the left. An example of an architectural composition on the right.

can be injected (or removed) at runtime in between any two components. These proxies are often written specifically to operate with particular interfaces in mind, such as *ws.Web*, as they take account of the semantics of those interfaces. We refer to these components as non-functional requirement proxies (NFRP). An example of an NFRP is *NFRPCache*, where a cache proxy is autonomously 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. Although NFRPs must be implemented with specific attention to a given interface, once implemented they are then generic to all uses of that interface; in the example of a microservice we implement NFRPs against *ws.Web* which are then reusable across different microservices.

Finally, the machine role in the system's design is entirely performed at runtime. Once the components are selected and placed in the microservice's folder, and the path to the folder is given to the Assembly module, the machine takes the available components and assembles them into every possible architectural composition available. Each composition represents one action for real-time learning, where the selection causes the Assembly module to calculate a delta between the current composition and the requested one before performing a sequence of adaptations to move between the two compositions.

As the system handles incoming requests from users, the framework experiments with different architectural compositions to learn which composition best satisfies the system's high level goals. Once the optimal composition is found for the current operating environment, the system exploits the benefits of the optimal composition, and as soon as the optimal composition performance starts decaying, or a new operating environment is detected, the system triggers the exploration phase again and starts experimenting different compositions in the new operating environment. Whenever previously seen environments are encountered, the system is able to remember the associated optimal composition and immediately change its architecture composition to that known optimal.

### 4.3 Discussion

Our approach to designing emergent microservices has the potential to reduce the complexity of building performant microservices which become tailored in their design to the current client workload, while still supporting the scale-out properties and composition of microservices at the macro level.

The use of microservices to define the boundaries of an emergent system, in terms of the scale over which real-time learning needs to operate, offers a useful way to control the granularity of an autonomously-designed subsystem. We introduce two main classes of components which deliver variation to form a search space for machine learning: utility components from a standard library with various different implementations, and injected proxies to deliver generalised non-functional properties between specific components. This allows the programmer to write the business logic of the microservice while our framework injects behavioural variation around this logic to maximise performance.

A third dimension of injected variation, which we consider for future work, is in the distribution of individual components that form a microservice. Here a selected piece of business logic, or a utility component, could be relocated to a remote host – or replicated across a set of remote hosts – to control the distributed design of the microservice. With this capability, the lines between microservices would become far more blurred as an individual microservice could scale out pieces of itself as appropriate; indeed, we could design a single microservice representing a large-scale service which becomes distributed across a network in the most efficient way to meet current demand.

## 5 Evaluation

In order to demonstrate and evaluate the overall approach proposed in Section 4, we leverage existing research on microservice architectures; specifically, we employed the Perception, Assembly and Learning framework to the implementation of the InterSCity smart city platform [1], which was designed from scratch as a microservice architecture.

InterSCity provides microservice-based APIs to support the development of smart city applications and services. Its microservices provide a variety of basic city-related functionalities as follows.

The Resource Adaptor microservice is responsible for integrating city resources (e.g., public transport buses, traffic lights, and lamp posts) with the platform, resolving issues related to heterogeneity and concurrent access. In turn, Resource Catalog, Data Collector and Actuator Controller are responsible, respectively, for the management of existing resources and the data collected from them, as well as for managing actuation capabilities. Finally, applications can discover and visualise city resources via the APIs of the Resource Discovery and Resource Viewer microservices, respectively.

In order to demonstrate our approach, we built a new version of the InterSCity platform from scratch, following the guidelines described in Section 4 for the implementation of each microservice. We used this version of InterSCity for a first evaluation of the approach presented in the paper. We specifically examined the effectiveness of the PAL framework for autonomously handling different variations of the Data Collector microservice and directing its adaptation in response to different client workload characteristics. The component repository folder used to discover compositions of the Data Collector was populated with NFRPs that implement two different non-functional concerns, namely data compression and caching. This enables four variations of the microservice's internal architecture: with cache only (referred to as NFRPCache); with both cache and compression (NFRPCacheCompression); with compression only (NFRPCompression); and with neither cache nor compression (NFRProxy, i.e., the pure business logic composition). The intended role of the cache is to enhance the efficiency of the microservice's access to the underlying database as the same data items may be requested multiple times by clients. Compression, in turn, aims to reduce the size of the messages exchanged between the microservice and its clients. Thus, when using the microservice with compression, the header of the HTTP messages exchanged with clients is changed to indicate the type of compression used, so that the client can correctly handle the payload.

It is expected that the different compositions of the Data Collector described above will perform differently under different workloads. To verify this, we implemented two hypothetical applications from the public transportation domain, a common application area of the InterSCity platform. The first application is used by public transport users to query the current location and estimated time of arrival of buses, while the second one is used by transport engineers to capture long-term data about the mobility of buses in a bus route. Thus, the first application characterises a scenario with a low volume of data and a high frequency of updates (as the bus location is continuously changing). The second application, by comparison, represents a scenario with a (relatively) high volume of data and a low frequency of updates. In the real smart city deployment, the particular mixture of these application usage types would vary over the course of a day depending on what most users are doing.

For the purpose of machine learning, we set the non-functional goal of the PAL framework to be response time to client requests, where a lower average value is considered to be better. We measure response time at the server-side with an injected measurement proxy which records the length of time taken for the request handling routine to complete; in practice this equates to the amount of time taken for all HTTP response data to be sent to the client via a TCP send function. We checked experimentally that the observations taken by the server in this way were well correlated with the experience observed at the client side (which we cannot usually instrument) and confirmed that the two points of view are highly correlated under all conditions. Specifically, the client-side measurements showed higher overall response times, accounting for the extra latency of client data reception, but these response times changed across different workloads with the same ratios observed in the server-side response times.

We ran experiments to demonstrate the individual performance of each of the four Data Collector compositions in the two scenarios. The results, in terms of the response times observed at the server side, are shown in Figures 3 and 4, respectively, and are discussed next (NB: in both graphs, the orange line represents the resulting behaviour when using the PAL framework, which will be discussed at the end of this section). Both graphs use a logarithmic scale to more clearly show the lower response times where most of the data sits. All data shown here is taken from response time readings seen at the server side, which are used to inform learning decisions.

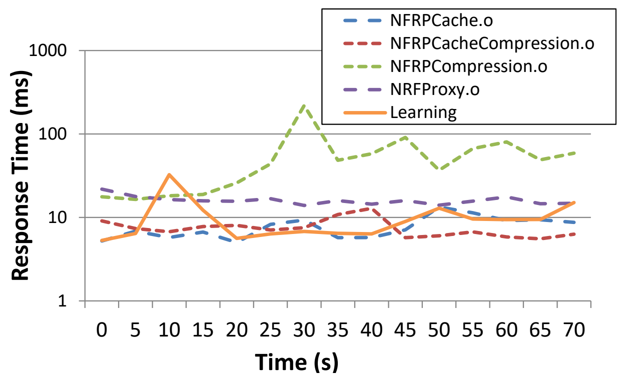


**Figure 3.** Performance of the emergent microservice compared with four fixed microservice compositions, exposed to the high frequency of update and low volume of data. The spike in the orange line represents the learning phase.

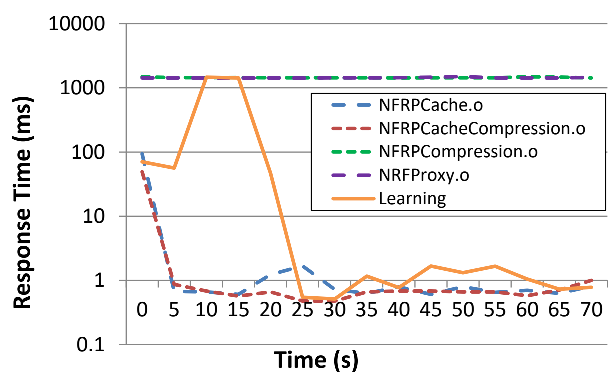


**Figure 4.** Performance of the emergent microservice compared with four fixed microservice compositions, exposed to the low frequency of update and high volume of data. The spike in the orange line represents the learning phase.

In the scenario considered in Figure 3, we can see that the two compositions employing cache (represented by the blue and red lines) have better performance than the other two (yellow and green lines). This is because a single data item (representing the current location of a bus) may be requested by different clients at the same time. Compression in turn has the poorest performance as the incurred overhead is not compensated by the diminishing gains of compressing low volumes of data.

In the scenario analysed in Figure 4, again the compositions that use caching perform visibly better, with the one using both compression and caching being expected to perform slightly better, due to the effects of compression being more evident with high data volumes. The dominant effect is again played by caching.

The PAL framework was then used to experiment with the autonomic composition of the Data Collector microservice using its four possible compositions under the two workload scenarios. The results are shown by the orange line in Figures 3 and 4. As can be seen, in both scenarios, after the learning phase, which spans approximately the first 20 seconds, the Learning module is able to select the best performing composition according to response time. For the scenario with high update frequency and low volume of data (Fig. 3), the Learning module selects the NFRPCache composition corresponding to the blue line, and then follows this line quite closely. Towards the end of the experiment we see that this composition degrades slightly in performance compared to the NFRPCacheCompression; the PAL framework does not change its choice here because performance is still within a threshold of the original learned value. If the performance moved out of this threshold it would trigger a new round of learning to verify the best decision. For the scenario with low update frequency and high volume of data (Fig. 4), the PAL framework selects the NFRPCacheCompression composition as the best option; again we see that this reflects the best-performance available and closely matches this performance over the course of the experiment.

Overall, the results clearly depict significant levels of improvement in the microservice performance. In cases where there is a substantial difference between the values from one architecture to another, we successfully demonstrate that the emergent microservice converges towards the optimal (or near optimal) composition. It is important to highlight that microservice developers should focus their efforts only on the development of the bare business logic of microservices, which might not, in most scenarios, represent the optimal composition. Based on the results of the above experiment, we argue that the added non-functional requirement proxies assist in improving the microservice's performance.

Furthermore, we argue that generalised non-functional requirement proxies can be used to transparently optimise any of the remaining microservices that are part of the InterSCity macro-architecture. We aim to refine this idea in future work, including by introducing further generalised proxies which offer other non-functional properties. We will also build on our work to demonstrate that the approach applies to microservice architectures in general, beyond the case study that we have presented here.

## 6 Conclusion

We have presented a methodology for constructing emergent microservices, combining two recent trends to tackle complexity in modern software systems. Microservices offer a simple, strongly encapsulated way to deliver distributed systems with good scaling properties, while emergent systems offer a way to offload the responsibility for the design of a system to real-time learning.

In combining these concepts, we gain an intuitive way to scope the responsibility of the machine learning processes involved in emergent systems, which aids in reducing the search space size of possible behaviour compositions that is navigated at runtime; and we gain a new dimension of optimisation in microservice architectures which enables continuous tuning to the client workload.

We have applied the approach to a real case study of a smart city platform, and demonstrated that a microservice in this platform is able to quickly learn the most suitable behaviour at runtime when given a goal of response time to optimise – a result achieved by combining programmer-supplied business logic with generalised non-functional proxies which can inject caching or compression behaviour into the system.

In future work we will explore the macro level of microservice composition in two major ways. First, how multiple emergent systems (each modeled as its own microservice) can reach good decisions when they are learning at the same time as part of the same global system, so that a globally-good composition is located. And second, how the ability to autonomously distribute individual components of a microservice (such as an XML parser) may enable a more fluid scale-out architecture where sub-elements can be replicated as demand on their utility increases or decreases over time. This direction may, in turn, lead to redefining the boundary between human and machine design in traditional microservices – for example where a single emergent microservice can fragment and scale itself out across multiple hosts as demand on it increases.

## Acknowledgements

The research reported in this paper was partly funded by the Royal Society – Newton Mobility Grant NMG-R2-170105 – and by the Leverhulme Trust Research Project Grant *The Emergent Data Centre*, RPG-2017-166. This research is also part of the INCT of the Future Internet for Smart Cities funded by CNPq proc. 465446/2014-0, CAPES proc. 88887.136422/2017-00, and FAPESP procs. 14/50937-1 and 15/24485-9.

## References

[1] Arthur M Del Esposte, Fabio Kon, Fabio M Costa, and Nelson Lago. 2017. InterSCity: A Scalable Microservice-based Open Source Platform for Smart Cities.. In *SMARTGREENS*. 35–46.

[2] Mahdi Derakhshanmanesh and Marvin Grieger. Model-Integrating Microservices: A Vision Paper.. In *Software Engineering Workshops*, Vol. 1559. CEUR Workshop Proceedings.

[3] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. 2017. Microservices: yesterday, today, and tomorrow. In *Present and Ulterior Software Engineering*. Springer, 195–216.

[4] Roberto Rodrigues Filho and Barry Porter. 2017. Defining Emergent Software Using Continuous Self-Assembly, Perception, and Learning. *ACM Transactions Autonomic Adaptive Systems* 12, 3, Article 16 (Sept. 2017), 25 pages. DOI: https://doi.org/10.1145/3092691

[5] Maurizio Gabbrielli, Saverio Giallorenzo, Claudio Guidi, Jacopo Mauro, and Fabrizio Montesi. 2016. Self-reconfiguring microservices. In *Theory and Practice of Formal Methods*. Springer, 194–210.

[6] Claudio Guidi, Ivan Lanese, Manuel Mazzara, and Fabrizio Montesi. 2017. Microservices: a language-based approach. In *Present and Ulterior Software Engineering*. Springer, 217–225.

[7] P. Jamshidi, C. Pahl, N. C. MendonÃğa, J. Lewis, and S. Tilkov. 2018. Microservices: The Journey So Far and Challenges Ahead. *IEEE Software* 35, 3 (May 2018), 24–35. DOI: https://doi.org/10.1109/MS.2018.2141039

[8] Barry Porter, Matthiew Grieves, Roberto Rodrigues Filho, and David Leslie. 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, 14.

[9] F. Rademacher, S. Sachweh, and A. ZÃijndorf. 2017. Differences between Model-Driven Development of Service-Oriented and Microservice Architecture. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*. 38–45. DOI: https://doi.org/10.1109/ICSAW.2017.32

[10] Olaf Zimmermann. 2017. Microservices tenets. *Computer Science - Research and Development* 32, 3 (01 Jul 2017), 301–310. DOI: https://doi.org/10.1007/s00450-016-0337-0