Microservices: The Evolution and Extinction of Web Services?

Luciano Baresi and Martin Garriga

Abstract In the early 2000s, service-oriented architectures (SOA) emerged as a paradigm for distributed computing, e-business processing, and enterprise integration. Rapidly, SOA and web services became the subject of hype, and virtually every organization tried to adopt them, no matter their actual suitability. Even worse, there were nearly as many definitions of SOA as people adopting it. This led to a big fail on many of those attempts, as they tried to change the problem to fit the solution. Nowadays, microservices are the new weapon of choice to achieve the same (and even more) goals posed to SOA years ago. Microservices ("SOA done right") describe a particular way of designing software applications as suites of independently deployable services, bringing dynamism, modularity, distributed development, and integration of heterogeneous systems. However, nothing comes for free: new (and old) challenges appeared, including service design and specification, data integrity, and consistency management. In this chapter, we identify such challenges through an evolutionary view from the early years of SOA to microservices, and beyond. Our findings are backed by a literature review, comprising both academic and gray literature. Afterwards, we analyze how such challenges are addressed in practice, and which challenges remain open, by inspecting microservice-related projects on GitHub, the largest open-source repository to date.

L. Baresi
Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano, Milan, Italy
e-mail: luciano.baresi@polimi.it

M. Garriga (✉)
Faculty of Informatics, National University of Comahue, Neuquán, Argentina

CONICET, National Scientific and Technical Research Council, Buenos Aires, Argentina
e-mail: martin.garriga@fi.uncoma.edu.ar

A. Bucchiarone et al. (eds.), *Microservices*,
https://doi.org/10.1007/978-3-030-31646-4_1

1 Introduction

Some 20 years ago, service-oriented architecture (SOA), web services, and service-oriented computing (SOC) were the buzzwords of the day for many in the business world [11]. Virtually every company adopted, or claimed to adopt, SOA and web services as key enablers for the success of their projects. However, there were nearly as many definitions of SOA as organizations adopting it. Furthermore, such panorama obscured the value added from adopting the SOA paradigm. The many proposed standards (e.g., WSDL and BPEL) were supposed to break the barriers among proprietary systems and serve as common languages and technologies to ease the integration of heterogeneous, distributed components, fostering the cooperation among independent parties. However, these approaches often failed when applied in practice, mainly because ever-changing business requirements, to which they were not able to (nor designed to) react timely [25]. In other words, many organizations applied SOA because of the hype and not given their actual needs.

Nowadays, we are witnessing the same hype for a new set of buzzwords: microservices and microservice architectures [26]. Microservices describe a particular way of designing software applications as suites of independently deployable services. One may also say that it is nothing but "SOA done right," as they preach for the same advantages, such as dynamism, modularity, distributed development, and integration of heterogeneous systems. However, now the focus is not on reuse and composition, as it is on independence, replaceability, and autonomy [28]. Services then become micro in terms of their contribution to the application, not because of their lines of code. They must be entities that can be conceived, implemented, and deployed independently. Different versions can even coexist and the actual topology of the system can be changed at runtime as needed. Each single component (microservice) must be changeable without impacting the operation and performance of the others.

However, as happened with SOA, microservices are not a silver bullet. With them, new challenges have appeared, as old ones regained attention. Just like any incarnation of SOA, microservice architectures are confronted with a number of nontrivial design challenges that are intrinsic to any distributed system—including data integrity and consistency management, service interface specification and version compatibility, and application and infrastructure security. Such design issues transcend both style and technology debates [49].

This chapter attempts to provide an evolutionary view of what services have been, are, and will be from the early times of SOA—with WSDL/SOAP-based services—through RESTful services, and finally to the advent of microservices and their possible evolution into functions-as-a-service (FaaS) [35]. By doing this, we shed some light on what is novel about microservices, and which concepts and principles of SOA still apply. Then, we complement this evolutionary view with a literature review (including both academic and gray literature) to identify the new (and the old) challenges still to be faced when adopting microservices. Finally, we analyze how practitioners are addressing such challenges by diving into the current microservices landscape in the biggest open-source repository to date: GitHub.[1] Our preliminary study on mining microservices on GitHub helps us understand the trending topics, challenges being addressed, as well as popular languages and tools.

To conclude, and summarize, the contributions of this chapter are threefold:

- An evolutionary view of SOA, from WSDL/SOAP to microservices and beyond
- A discussion regarding current challenges on microservices, based on a review of academic and gray literature
- A panorama of the current landscape of microservices on GitHub, and how those challenges are being addressed

The rest of this chapter is organized as follows. Section 2 presents the evolutionary view from first-generation SOA through REST to microservices and serverless. Section 3 revisits old and new challenges of SOA in the era of microservices. Section 4 discusses the microservices ecosystem on GitHub. Finally, Sect. 5 concludes the chapter.

2 Web Services Then and Now

This section provides an evolutionary view from the early days of WSDL/SOAP-based services (Sect. 2.1), to RESTful services (Sect. 2.2), then to microservices (Sect. 2.3), and the possible evolution into the novel functions-as-a-service (FaaS) [35] (Sect. 2.4).

2.1 SOA(P) Services

Service-oriented architectures (SOA) emerged as a paradigm for distributed computing, e-business processing and enterprise integration. A service, and particularly a web service, is a program with a well-defined interface (contract) and an id (URI), which can be located, published, and invoked through standard Web protocols [29]. The web service contract (mostly specified in WSDL) exposes public capabilities as operations without any ties to proprietary communication frameworks. Services decouple their interfaces (i.e., how other services access their functionality) from their implementation.

The benefits of SOA are multifaceted [10]. It provides dynamism, as new instances of the same service can be launched to split the load on the system. Modularity and reuse, as complex services are composed of simpler ones and the same services can be (re)used by different systems. Distributed development, since distinct teams can develop conversational services in parallel by agreeing on their

[1] https://github.com/.

interfaces. Finally, integration of heterogeneous and legacy systems, given that services merely have to implement standard protocols (typically SOAP—Simple Object Access Protocol [7]) to communicate over existing logic.

On top of that, specific workflow languages are then defined to orchestrate several services into complex compositions (e.g., WS-BPEL, BPEL4WS) [16]. As these languages share ideas with concurrency theory, this aspect fostered the development of formal models for better understanding and verifying service interactions (i.e., *compositions*), ranging from foundational process models of SOA to theories for the correct composition of services [10]. In the early years of SOAP-based service composition, according to different surveys [32, 40] the literature mainly focused on two aspects: Definition of clear/standard steps (modeling, binding, executing, and verifying) of Web Service composition, and classification of compositions into workflow-based industry solutions (extending existing languages, e.g., WS-BPEL and BPEL4WS) and semantics-based academic solutions, using planning/AI upon semantic languages such as OWL-S.

2.2 RESTful Services

Years after SOA irruption, stakeholders still disagreed about its materialization, and mostly failed to implement it [25]. First, the absence of widely accepted usage standards led organizations to develop and/or describe web services and compositions using divergent specification practices and concept models [16, 17]. Besides, daunting requirements regarding service discovery (e.g., UDDI registries [36]) or service contracts agreements (WSLA) hindered the adoption of early SOA models. Second, the claimed benefits and hype of SOA tempted organizations to adopt it even when their particular context said the contrary [25]. Pursuing flexibility too early, before creating stable and standardized business processes, plus the problems of interoperability and data/process integration (through too smart communication mechanisms such as the enterprise service bus), led traditional SOA to fail often.

In such a context, REST (REpresentational State Transfer) [13] appeared as a simpler, lightweight, and cost-effective alternative to SOAP-based services. Although the term was coined in 2000 by Roy Fielding, RESTful services gained traction around one decade after [30]. RESTful services use the basic built-in HTTP remote interaction methods (PUT, POST, GET, and DELETE) and apply their intended semantics to access any URI-referenceable resource. HTTP methods then became a standardized API for services, easier to publish and consume.

As the years passed, REST and HTTP (and JSON as data exchange format) became ubiquitous in the industry, in detriment of WSDL/SOAP-based solutions [36]. This dominance fits well with the characteristic of microservices being built on top of lightweight communication mechanisms, as we will see in the next section.

Still, reuse and composition issues were under discussion in the RESTful era. Humans being considered as the principal consumer/composer of

RESTful services explains the lack of machine-readable descriptions, and the massification of user-driven composition approaches (mashups) [17]. We can keep the aforementioned distinction between workflow- and semantic-based solutions; process-oriented mashups and extended business composition languages (such as BPEL4REST) belong to the first group, while semantic annotations, planning-based solutions, and formalization efforts define the second class [17].

2.3 Microservices

Nowadays, most of the issues related to defining, classifying, and characterizing services and composition solutions mentioned in the previous sections are overcome. However, yet new challenges appeared, posed by the internet of services/things, pervasive computing, and mobile applications.

The environment in which services are developed and executed has become more open, dynamic, and ever changing. This raises several malleability issues, including the ability of self-configuring, self-optimizing, self-healing, and self-adapting services. This may involve devices with limited resources and computational capabilities [6], and calls for novel algorithms for dynamically managing such lightweight and simple services. Also, to manage services in current pervasive environments, one must address context awareness, heterogeneity, contingencies of devices, and personalization. A pervasive environment claims for appropriate semantic technologies, shared standards, and mediation to assure interoperability of heterogeneous entities, such as mobile devices, sensors, and networks. Finally, as users are now becoming "prosumers" [22] (i.e., both producers and consumers), it is still unclear how to combine the need for aggregating several services, maintaining their QoS, and keeping the coupling level as low as possible.

In this context, microservices came to the scene as the weapon-of-choice to address such challenges at the enterprise scale. Microservices are independently deployable, bounded-scoped components that support interoperability by communicating through lightweight messages (often a HTTP API) [27]. In turn, microservice architecture is a style for engineering highly automated, evolvable software systems made up of capability-aligned microservices [27]. Each service also provides a physical module boundary, even allowing for different services to be written in different programming languages and be managed by different teams [26].

However, most of this definition applies to traditional SOAP-based or RESTful services as well, which feeds the debate regarding microservices and SOA. Although microservices can be seen as an evolution of SOA, they are inherently different regarding sharing and reuse. SOA is built on the idea of fostering reuse: a *share-as-much-as-possible* architecture style, whereas microservice architectures seconds the idea of a *share-as-little-as-possible* architecture style [33]: the goal became how to build systems that are *replaceable* while being *maintainable* [26]. Given that service reuse has often been less than expected [47], microservices should be "micro" enough to allow for the rapid development of new versions that can coexist, evolve, or even replace the previous one according to the business needs [19]. This is also possible thanks to continuous deployment techniques [2], such as canary deployment—pushing the new versions to a small number of end users to test changes in a real-world environment; and version concurrency—incrementally deploying new versions of a service, while both old and new versions of the service contract are running simultaneously for different clients. Thus, microservices fit well to scenarios with loose data integration and highly dynamic processes, bringing the opportunity to innovate quickly [25].

Undoubtedly, also microservices will be replaced by the next technological choice to implement the SOA architectural style. Thus, before moving to the challenges being faced nowadays by microservices (Sect. 3), we discuss one of the possible evolution paths for this architecture: functions-as-a-service (FaaS), also known as serverless computing. One should note that FaaS conveys the same design principles and benefits of microservices (isolation, interchangeability), but presents substantial differences to support such design at the technical and technological level, as we will see below.

2.4 Upcoming Faasification

A serverless architecture is a refined cloud computing model that processes requested functionality without pre-allocating any computing capability. Provider-managed containers are used to execute functions-as-a-service, which are event-triggered and ephemeral (may only last for one invocation) [35]. This approach allows one to write and deploy code without considering the runtime environment, resource allocation, load balancing, and scalability; all these aspects are handled by the provider.

Serverless represents a further evolution of the pay-per-use computing model: we started allocating and managing virtual machines (e.g., Amazon EC2) by the hour, then moved to containers (e.g., CS Docker Engine), and now we only allocate the resources (a container shared by several functions) for the time needed to carry out the computation—typically a few seconds or milliseconds.

The serverless architecture has many benefits with respect to more traditional, server-based approaches. Functions share the runtime environment (typically a pool of containers), and the code specific to a particular application is small and stateless by design. Hence, the deployment of a pool of shared containers (workers) on a machine (or a cluster of machines) and the execution of some code onto any of them become inexpensive, efficient, and completely handled by the cloud provider.

Horizontal scaling is completely automatic, elastic, and quick, allowing one to increase the number of workers against sudden spikes of traffic. The serverless model is much more reactive than the typical solutions of scaling virtual machines or spinning up containers against bursts in the workload [20]. Finally, the pay-per-use cost model is fine-grained, down to a 100 ms execution granularity for all the major vendors, in contrast to the "usual" hour/minute-based billing of virtual machines and containers. This allows companies to drastically reduce the cost of their infrastructures with respect to a typical monolithic architecture or even a microservice architecture [46].

Several cloud providers have developed serverless solutions recently that share those principles. AWS Lambda is the first and perhaps most popular one, followed by Azure Functions, Google Firebase, and IBM/Apache Openwhisk (the only open-source solution among the major vendors). A couple other promising open source alternatives are OpenFaaS (multilanguage FaaS upon Docker or Kubernetes) and Quarkus (heavily optimized for Java and Kubernetes).

Back to microservices, one of their main concerns is the effort required to deploy and scale each microservice in the cloud [46]. Although one can use automation tools such as Docker, Chef, Puppet, or cloud vendor-provided solutions, their adoption consumes time and resources. To address this concern, FaaS appears as a straightforward solution. Once deployed, functions can be scaled automatically, hiding the deployment, operation, and monitoring of load balancers or web servers. The per-request model helps reduce infrastructure costs because each function can be executed in computing environments adjusted to its requirements, and the customer pays only for each function execution, thus avoiding infrastructure payment when there is nothing to execute [46].

Thus, the way to go for microservices could be to become even more fine-grained, slayed into functions. For instance, given a RESTful microservice that implements an API with basic CRUD operations (GET, POST, PUT, DELETE), one might have a single function to represent each of these API methods and perform one process [41]. Furthermore, when CRUD microservices are not desirable,[2] event-driven or message-based microservices could still be represented as functions, tailored to the same events that the microservices listen(ed) to. Besides, serverless computing is stateless and event-based, so serverless microservices should be developed as such.

However, these new solutions bring together new challenges and opportunities.[3] For example, we still need to determine the sweet spots where running code in a FaaS environment can deliver economic benefits, automatically profile existing code to offload computation to serverless functions [4], bring adequate isolation among functions, determine the right granularity to exploit data and code locality, provide methods to handle state (given that functions are stateless by definition) [39], and finally increase the number of out-of-the-box tools to test and deploy functions locally. Additionally, going serverless is *not* recommended when one wants to [41]:

- Control their own infrastructure (due to regulations or company-wide policies)
- Implement a long-running server application (transactional or synchronous calls are the rule)

[2]https://www.ben-morris.com/entity-services-when-microservices-are-worse-than-monoliths/.

[3]https://blog.zhaw.ch/icclab/research-directions-for-faas/.

- Avoid vendor lock-in (given that each provider has its own set of serverless APIs and SDKs)
- Implement a shared infrastructure (as multi-tenancy is managed by the provider).

3 Challenges

The evolutionary view from early SOA to the advent of microservices helped us understand what is novel about microservices, and which concepts and principles of SOA still apply. In this section, we complement this evolutionary view with a discussion of the challenges still to face when adopting microservices.

The challenges presented throughout this section are the result of a literature review, following the guidelines for systematic literature review (SLR) proposed in [24]. Although a complete SLR is outside the scope of this work, this helped us organize the process of finding and classifying relevant literature. We considered research published up to the first quarter of 2017. This led us to a collection of 46 relevant works,[4] both primary (28) and secondary studies (18). Interested readers can refer to [15] for details on these studies.

Given the novelty of the topic, we enriched our results by comparing them with those of a recent gray literature review [38], which includes materials and research produced by organizations outside of the traditional academic publishing and distribution channels—such as reports, white papers, and working documents from industry [14]. Interestingly, academic and gray literature share common findings regarding open challenges in the microservice era, as we will see throughout this section. For the sake of organization, we divide such challenges regarding the lifecycle stages: Design (Sect. 3.1), Development (Sect. 3.2), and Operations (Sect. 3.3) and conclude with a discussion (Sect. 3.4).

3.1 Design Challenges

Despite the hype and the business push towards microservitization, there is still a lack of academic efforts regarding the design practices and patterns [19]. Design for failure and design patterns could allow to address challenges early as to bring responsiveness (e.g., by adopting "let-it-crash" models), fault tolerance, self-healing, and variability characteristics. Resilience patterns such as circuit-breaker and bulkhead seem to be key enablers in this direction. It is also interesting to understand whether the design using a stateless model based on serverless functions [20] can affect elasticity and scalability as well [8].

[4]Due to the space limit, the full list can be found at: https://goo.gl/j5ec4A.

Another problem at design time is *dimensioning* microservices—i.e., finding the right granularity level [38]. This obviously implies a trade-off between size and number of microservices [19]. Intuitively, the more microservices, the higher the isolation among business features, but at the price of increased network communications and distribution complexity. Additionally, the boundaries among the business capabilities of an application are usually not sharp. Addressing this trade-off systematically is essential for identifying the extent to which "splitting" is beneficial regarding the potential value of microservitization [3].

Security by design is also an open challenge, given the proliferation of endpoints in microservice ecosystems, which are only the surface of a myriad of small, distributed and conversational components. The attack surface to be secured is hence much larger with respect to classical SOA, as all the microservices are exposing remotely accessible APIs [38]. In this direction, access control is crucial, as the design of microservice-based applications should allow each component to quickly and consistently ascertain the provenance and authenticity of a request, which is challenging due to the high distribution [38].

3.2 Development Challenges

Most of today's microservices exploit RESTful HTTP communication [36]. Message queues are promising but not adopted as expected, in concordance with the lack of proposals for asynchronous interaction models [15]. As such, communications are purely based on remote invocations, where the API becomes a sort of contract between a microservice and its consumers. This generates coupling and directly impacts APIs' versioning, as new versions must always be retro-compatible to avoid violating the contracts among microservices, hence allowing them to continue to intercommunicate [38].

This suggests not only that microservices are being used in-house, with contracts negotiated between different teams/people inside the company, but also that their reuse should support concurrent versions and incremental releases: new versions can be (re)developed entirely to fulfill new requirements, while keeping old versions for other clients. The recent efforts on standardizing RESTful APIs through OpenAPI specifications[5] seem interesting and also applicable to specify microservices [3].

Another challenge comes from data persistency issues. A database can be part of the implementation of a microservice, so it cannot be accessed directly by others [34]. In this scenario, data consistency becomes difficult to achieve. Eventual consistency (the distributed database does not exhibit consistency immediately after a write, but at some later point) is an option, even if not always acceptable for any domain, and not easy to implement too. At the same time, this heavy distribution complicates distributed transactions and query execution (also because

[5]https://www.openapis.org/.

of the heterogeneity of the data stores to be queried). In this scenario, testing is also complex, as the business logic is partitioned over independently evolving services. Approaches that use/propose frameworks for resilience testing [21] or reusable acceptance tests [31] are highly required.

3.3 Operation Challenges

The primary challenge during the operation of microservice-based applications is given by their resource consumption. More services (with respect to traditional SOA) imply more runtime environments to be distributed, and remote API invocations. This increases consumption of computing and network resources [38]. However, there seems to be a mistrust regarding built-in solutions of cloud providers, which sometimes become too rigid [5] or cumbersome to configure and adjust [20]. In the meantime, cloud providers are growing in variety and usability (e.g., AWS has offered around 1000 new features per year[6]), and we believe that they will become the standard to deploy and manage cloud microservices in the near future [15].

Operational complexity also comes along with the distributed and dynamic nature of microservices. They could be flexibly scaled in and out, or migrated from one host to another. Moreover, they could be switched from the cloud to the edge of the network [6]. This, along with the huge number of microservices forming an application, makes it challenging to locate and coordinate their concrete instances. At the same time, distributed logging calls for aggregation approaches that help track the reasons behind issues/errors [38].

3.4 Discussion

The challenges of microservice-based applications are mainly due to their novelty and intrinsic complexity and distribution. Their design, development, and operation is hampered by the fact that the business logic in such applications is heavily distributed over many independent and asynchronously evolving microservices [38]. As a summary, Table 1 highlights the relationship among the usual steps of the development process (design, development, operation), the principles behind microservices (defined in the seminal book by Newman [28]), example features related to each principle extracted from the (academic and grey) literature review, and finally example tools or practices applicable to such a stage/principle. In this way, we pave the ground to the analysis of the microservices ecosystem on GitHub, presented in the next section.

[6]https://techcrunch.com/2016/12/02/aws-shoots-for-total-cloud-domination/.

Table 1 Relationship among microservices lifecycle stages, principles, features, and tools

Stage	Principle	Example features	Tools/practices
Design	Modeled around business domain	Contract, business, domain, functional, interfaces, bounded context, domain-driven design, single responsibility	Domain-driven design (DDD), bounded context
Design	Hide implementation details	Bounded contexts, REST, RESTful, hide databases, data pumps, event data pumps, technology-agnostic	OpenAPI, Swagger, Kafka, RabbitMQ, Spring Cloud Data Flow
Dev	Culture of automation	Automated, automatic, continuous*(deployment, integration, delivery), environment definitions, custom images, immutable servers	Travis-CI, Chef, Ansible, CI/CD
Dev	Decentralize all	DevOps, Governance, self-service, choreography, smart endpoints, dumb pipes, database-per-service, service discovery	Zookeper, Netflix Conductor
Dev/ Ops	Isolate failure	Design for failure, failure patterns, circuit-breaker, bulkhead, timeouts, availability, consistency, antifragility,	Hystrix, Simian Army, Chaos Monkey
Ops	Deploy independently	versioning, one-service-per-host, containers	Docker, Kubernetes, canary|A/B|blue/ green testing
Ops	Highly observable	Monitoring, logging, analytics, statistics, aggregation	ELK, Elasticsearch, Logstash, Kibana

Recent implementations of microservices take the SOA idea to new limits, driven by the goals of rapid, interchangeable, easily adapted, and easily scaled components. As a cloud-native architecture, they play well on the basic functional features of cloud computing and its delivery capabilities [44]. The resulting factorization of workloads and incrementally scalable features of microservices provide a path by which SOA can be evolved from its previously rigid and overly formal implementation settings and be implemented in much less forbidding ways.

One consequence of this evolution is the development of new architectural patterns and the corresponding emergence and use of standards [37]. In that direction, we believe that open standard agreement is the basic prerequisite for achieving high interoperability and compatibility, being a key issue to be addressed [17]. The most clear example is a continued emphasis on the use and proper documentation of RESTful APIs, by means of Swagger/OpenAPI specifications [37]. A standardized service description and choreography approach can assure compatibility with any service, and achieve greater evolvability. Finally, standardized services in the surface can collaborate with partners for better data portability, collaborating to solve the challenges around distributed storage in microservices [38].

Finally, a few words about the organizational aspects that surround microservices. It is important to link more explicitly microservices with the DevOps (development plus operations as a single team) movement. DevOps seems to be a key factor in the success of this architectural style [1], by providing the necessary organizational shift to minimize coordination among the teams responsible for each component, and removing the barriers for an effective, reciprocal relationship between teams. DevOps implies an organizational rewiring (equivalent to, e.g., the adoption of agile methodologies) and certain key practices (e.g., continuous delivery, integration, management). As this organizational shift is not simple, the literature reports different sociotechnical patterns [43] to enable the transition towards microservices. For example, sociotechnical-risks engineering, where critical architecture elements remain tightly controlled by an organization and loosely coupled with respect to outsiders, or shift left, where organizational and operational concerns (e.g., DevOps team mixing) are addressed earlier ("left") in the lifecycle toward architecting and development, rather than implementation and runtime.

4 Microservices on GitHub

Given the open challenges discussed in the previous section, we are interested in how practitioners are addressing them in practice. To answer this, we delve into the current microservices landscape in the biggest source of software artifacts on the Web to date: GitHub.[7] Our main goal is to identify the actual incidence of microservices and related tooling in practice. We stated the following research questions (RQs):

- **RQ1**: What is the activity and relevance of microservices in open source public repositories?
- **RQ2**: What are the characteristics of microservices-related repositories?
- **RQ3**: How are these projects addressing the aforementioned open challenges?

4.1 Dataset Creation

We followed the guidelines for mining GitHub defined in the literature [23, 48], and considered the following information:

- *Releases* of a repository: Each release is a specially tagged push event, composed of several commits (at least one) to a stable repository branch.

[7]https://GitHub.com/.

- *Push events* to the master branch of a repository, as an indicator of repository *activity* (each push event is composed of one or more commits, and is triggered when a repository branch is pushed to).
- *Stars*, as an indicator of repository *relevance* for the GitHub community (starring a repository allows one to keep track of interesting projects).
- *Topics*, as an indicator of the repository *topics* (this allows one to describe, find, and relate repositories).

Then, we used GitHub Archive[8] as our datasource of GitHub events. GitHub Archive provides a daily dump of GitHub activity (around 2.5 Gb of events per day). Given its size, it is only accessible through Google Big Query,[9] a web service that allows one to perform SQL-like interactive analysis of massive datasets (billions of rows).

We started by looking for *active* repositories—those with a Push event to their *master* branch during the last month. The total amount of *active* projects during 2018 exceeds 1 million. Thus, we additionally filtered repositories corresponding to our research—i.e., those using the *topic* "microservice" or "microservices" or mentioning these terms in the repository description.

The total number of repositories related to microservices is around 36,000. However, when analyzing sample repositories at random, we noticed that some of them are personal or class projects that, although being active, are not relevant for the community. These repositories have only one contributor, zero forks, and low popularity. With this dataset as a starting point, we narrowed our scope to *active* repositories related to microservices. Then, we defined an additional criteria for *relevant* repositories as those with 10+ *stars* (equivalent to followers or level of popularity). This information is accessed through the GraphQL-based GitHubAPI.[10] All in all, the number of 2018's *relevant* and *active* microservices-related repositories on GitHub is 651,[11] roughly 2% of the total 36,000 repositories, excluding forks and duplicated.

4.2 Quantitative Analysis

From the dataset of 651 repositories extracted in the previous step, we performed an initial quantitative analysis with the goal of answering the research questions. We started by identifying their topics, languages used, and other metadata such as commits, stars, and forks. Table 2 presents a summary of the initial analysis and tries to answer RQ1: (*What is the activity and relevance of microservices in open source*

[8]https://www.GitHubarchive.org/.

[9]https://bigquery.cloud.google.com/.

[10]https://developer.GitHub.com/v4/.

[11]The full list can be found at: http://cor.to/Gvyp.

Table 2 Summary of Microservice Projects on GitHub (RQ1)

Metric	Total	Average	Median
Total microservices projects	∼36,000	–	–
Active and relevant projects (+10 stars, PR in the last month)	651	–	–
Pull requests per project	–	128.7	17
Stars per project (average)	–	730.6	77
Watchers per project (average)	–	62.7	15

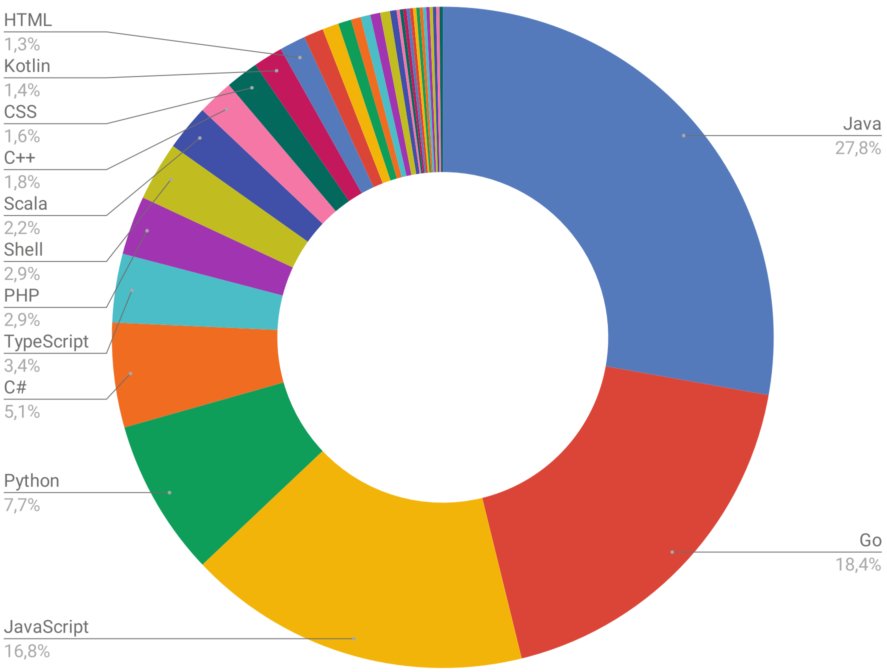

Fig. 1 Languages distribution of microservices projects (RQ2)

public repositories?). For brevity, raw data and the scripts and queries used to gather and analyze repositories metadata are accessible within our replication package.[12]

Moving to RQ2 (*What are the characteristics of the microservices-related repositories?*), Fig. 1 decomposes the use of programming languages in our dataset. Interestingly, Java is the most common language (27.8%), followed by Go (18.4%) and JavaScript (16.8%). Other languages show a narrower adoption, including Python, PHP, Typescrypt, and C#, among others. Given the polyglot nature of microservices, some of the repositories adopt various languages—we report the main language for each. Our results suggest that Java is still widely used [36] and,

[12]http://www.GitHub.com/webocs/mining-GitHub-microservices.

although it is perceived as an "old-fashioned" language, it can make microservices easier to adopt by Java developers, and also easier to integrate with legacy Java systems. Besides, microservices are commonly associated with lightweight, scripting languages such as JavaScript, which is reflected in practice—although we expected JavaScript to be the most popular language. Finally, Go is mostly a server-side language (developed by Google), mainly adopted for projects that provide support for microservices in the form of frameworks or middleware infrastructures.

As for RQ3, (*How are these projects addressing the aforementioned open challenges?*), we performed a topic ranking to grasp the underlying types of solutions and technologies used. Topics are labels that create subject-based connections between GitHub repositories and let one explore projects by, e.g., type, technology, or keyword.[13] Our ranking is summarized in Table 3. Note that microservices/microservice do not appear as a topic on all repositories, but it can be part of the description. Apart from those, the most popular languages: java, nodejs (javascript) and golang (go) appear among the top topics. The others are several tools for deploying and operating microservices such as docker (containers), kubernetes (a container orchestration tool) and spring-boot (spring "containers"), and cloud-related topics (cloud, spring-cloud). There are a few topics regarding specific solutions for microservices communication (rpc, grpc, REST, rabbitmq) and/or API design (API, REST, rest-api). Other challenges are underrepresented in the list of topics with quite small percentages.

Table 3 Main topics in microservices projects

Topic	Times	%	Topic	Times	%	Topic	Times	%
Microservices	270	41.47	API	31	4.76	Python	20	3.07
Microservice	222	34.10	Framework	30	4.61	Cloud	17	2.61
Java	88	13.52	Microservices-arch	28	4.30	Service-mesh	17	2.61
Docker	68	10.45	Distributed-systems	28	4.30	CQRS	16	2.46
Kubernetes	63	9.68	grpc	25	3.84	DevOps	16	2.46
Spring-boot	57	8.76	Rest	24	3.69	redis	16	2.46
Golang	52	7.99	API-gateway	23	3.53	http	16	2.46
Nodejs	51	7.83	Rest-API	21	3.23	Spring-cloud-core	16	2.46
Cloud-native	46	7.07	Containers	21	3.23	mongodb	15	2.30
Go	41	6.30	Serverless	20	3.07	Kafka	15	2.30
Spring-cloud	41	6.30	Service-discovery	20	3.07	DDD	15	2.30
Spring	41	6.30	Javascript	20	3.07	Proxy	14	2.15
rpc	33	5.07	Rabbitmq	20	3.07	Consul	13	2.00

[13]https://blog.GitHub.com/2017-01-31-introducing-topics/.

4.3 Qualitative Analysis

Continuing with RQ3, (*How are these projects addressing the aforementioned challenges*?), we performed a Qualitative analysis divided into two parts. First, we took a small sample (64 projects, 10% of the total) at random and analyzed it, which led to some preliminary insights. Support for the microservices lifecycle is the most common practice (44%), ranging from templates and boilerplate code for development to operational support in the form of containerization solutions, continuous integration, load-balancing, etc.—broadly speaking, DevOps practices—followed by specific solutions for communication among microservices (33%), implementing orchestration mechanisms, asynchronous communication, and API gateways. The rest of the projects actually implement microservices (23%), ranging from sample applications fully developed with several microservices to single or few microservices—e.g., for authentication or cryptography.

Interestingly, the focus is on supporting microservice architectures rather than on the development of microservices themselves. We believe that this is mainly due to two factors:

- A complexity shift: The claimed simplicity of microservices moves the complexity to their supporting systems [19].
- A different reuse perspective [33]: Instead of reusing existing microservices for new tasks or use cases, they should be small and independent enough to allow for rapidly developing a new one that can coexist, evolve, or replace the previous one according to the business needs [19].

Afterwards, we performed qualitative analysis on the whole dataset, organized around each stage of microservice lifecycle (recall Table 1). For doing this, we assessed the documentation of the repositories, in particular the *Readme* documents at the master branch. We crafted a dataset of 590 readme documents (90% of the total), given that some of the projects did not present their documentation in such a standard way.

We used the dataset (along with the metadata of repositories) to populate Apache Solr,[14] a search engine that features advanced matching capabilities including phrases, wildcards, and clustering. Solr makes use of Apache Lucene,[15] a well-known search library based on the TF.IDF model. This scoring model involves a number of scoring factors such as term frequency (the frequency with which a term appears in a document), inverse document frequency (the rarer a term is across all documents in the index, the higher its contribution to the score is) and other tuning factors such as coordination factor (the more query terms are found in a document, the higher its score is) and field length (the more words a field contains, the lower its score is). The exact scoring formula that brings these factors together is outside the

[14]http://lucene.apache.org/solr/.

[15]https://lucene.apache.org/.

scope of the present chapter, but a detailed explanation can be found in the Lucene documentation.[16]

Then, we built the queries accounting the different features and tools/practices presented in Table 1. The following discussion of obtained results is also organized around lifecycle steps, namely: design, development, and operation.

Table 4 shows the three queries related to the design stage, together with the total number of repositories found for each query, and the top 5 (most relevant) results. Clearly, the least addressed topic in the design of microservices is their modeling around the business domain, or domain-driven design, with only 29 repositories listed. As modeling is somewhat subjective, one may argue that tool support or examples may be scarce or not extrapolable to different domains. However, designing microservices and determining their granularity is still an open challenge [38], thus this may be a call for further efforts. Note that the most relevant repositories are sample applications and not supporting tools—with the exception of #4 (boilerplate architecture) and #5 (an opinionated framework for .NET).

Conversely, RESTful interfaces (specified through Swagger/OpenAPI) seem to be a widespread practice (208 repositories) in line with the perceived state-of-practice in industry [36]—#1 bringing together the most popular communication style (REST) with the most popular language (Java). Gaining traction, 147 repositories deal with asynchronous communication through messaging protocols such as RabbitMQ or Spring Cloud Stream for Java (#3).

Entering development stage (Table 5), the most addressed topic is automation (275 repositories) through continuous integration and deployment techniques (CI/CD), Travis being the weapon of choice, perhaps because of its easy integration with GitHub [48]. One may highlight #1 as it provides an interesting approach for managing multiple apps in a single (mono) repository.

Orchestration/choreography (41 repositories) are not so popular, as microservice integration so far has been usually ad hoc and in-house. The salient example is Netflix Conductor (#1), nowadays the most popular orchestrator for microservices. Somewhat service discovery (101 repositories) is more popular, although from a different perspective w.r.t. discovery in traditional SOA [18]. Nowadays, the problem is to find the "alive" endpoints among the multiple copies of a microservice at runtime [42], through tools such as Apache Zookeeper, etcd, Consul or VertX (#2). However, for large-scale microservice architectures, orchestration and choreography solutions are a must-have, although there is not a large number of these cases on GitHub. Technologies such as NGINX, AWS API gateways, or Kubernetes control plane are the alternatives for enterprise microservices management.

Finally, Table 6 summarizes results for the operation stage. Failure isolation patterns are not so common (54 repositories). Circuit-breaker is the only pattern that provides significant results, while others such as bulkheads or timeouts were not mentioned throughout our dataset. The first four repositories in the results are sample microservices applications that implement circuit-breakers: #5 is an

[16]http://lucene.apache.org/core/3_5_0/api/core/org/apache/lucene/search/Similarity.html.

Table 4 Repository analysis by design principles and query (total repos and top 5 per query)

#repos	Principle/repoId	Query/description
29	**Modeled around business domain**	*"bounded context" OR ddd OR "domain driven design"*
1st	EdwinVW/pitstop	This repo contains a sample application based on a garage management system for PitStop—a fictitious garage. The primary goal of this sample is to demonstrate several Web-scale architecture concerns…
2nd	banq/jdonframework	Domain Events Pub/Sub framework for DDD
3rd	idugalic/digital-restaurant	DDD. Event sourcing. CQRS. REST. Modular. Microservices. Kotlin. Spring. Axon platform. Apache Kafka. RabbitMQ
4th	ivanpaulovich/clean-architecture-manga	Clean architecture service template for your microservice with DDD, TDD and SOLID using .NET Core 2.0. The components are independent and testable, the architecture is evolutionary in multiple dimensions…
5th	volak/Aggregates.NET	.NET event sourced domain driven design model via NServiceBus and GetEventStore
208	**Hide implementation details**	*rest OR restful OR swagger OR openapi OR "api blueprint"*
1st	noboomu/proteus	High-Performance RESTful Java web and microservice framework
2nd	mfornos/awesome-microservices	A curated list of microservice architecture-related principles and technologies
3rd	banzaicloud/pipeline	Pipeline enables developers to go from commit to scale in minutes by turning Kubernetes into a feature-rich application platform integrating CI/CD, centralized logging, monitoring, enterprise-grade…
4th	benc-uk/smilr	Microservices reference app showcasing a range of technologies, platforms and methodologies
5th	rootsongjc/awesome-cloud-native	A curated list for awesome cloud native tools, software, and tutorials
147	**Hide implementation details**	*asynchronous OR "data pump" OR "event pump" OR messaging OR RabbitMQ OR Kafka*
1st	mfornos/awesome-microservices	A curated list of microservice architecture-related principles and technologies
2nd	binhnguyennus/awesome-scalability	Scalable, available, stable, performant, and intelligent system design patterns
3rd	hipster-labs/generator-jhipster-spring-cloud-stream	JHipster module for messaging microservices with Spring Cloud Stream
4th	SaifRehman/ICP-Airways	Cloud Native application based on microservice architecture, IBM Middlewares and following 12 factor practices
5th	idugalic/digital-restaurant	DDD. Event sourcing. CQRS. REST. Modular. Microservices. Kotlin. Spring. Axon platform. Apache Kafka. RabbitMQ

Table 5 Repository analysis by development principles and query (total repos and top 5 per query)

#repos	Principle/repoId	Query/description
275	**Culture of automation**	*travis OR ci OR cd*
1st	MozillaSecurity/orion	CI/CD pipeline for building and publishing multiple containers as microservices within a mono-repository
2nd	vietnam-devs/coolstore-microservices	A containerized polyglot service mesh based on .NET Core, Nodejs, Vuejs and more running on Istio
3rd	rootsongjc/awesome-cloud-native	A curated list for awesome cloud native tools, software, and tutorials.
4th	scalecube/scalecube-services	ScaleCube services is a broker-less reactive-microservices-mesh that features: API-gateways, service-discovery, service-load-balancing, the architecture supports plug-and-play service communication…
5th	banzaicloud/pipeline	Pipeline enables developers to go from commit to scale in minutes by turning Kubernetes into a feature-rich application platform integrating CI/CD, centralized logging, monitoring, enterprise-grade…
41	**Decentralize all**	*orchestration OR choreography OR "netflix conductor"*
1st	Netflix/conductor	Conductor is a microservices orchestration engine
2nd	rootsongjc/awesome-cloud-native	A curated list for awesome cloud native tools, software, and tutorials
3rd	taimos/dvalin	Taimos microservices framework.
4th	InVisionApp/go-health	Library for enabling asynchronous health checks in your service
5th	Sharding-sphere/sharding-sphere	Distributed database middleware
101	**Decentralize all**	*"service discovery" OR zookeeper OR consul*
1st	smallnest/rpcx	Faster multilanguage bidirectional RPC framework in Go, like alibaba Dubbo and weibo Motan in Java, but with more features, scale easily
2nd	vert-x3/vertx-service-discovery	Some tools one can use for doing microservices with Vert.x
3rd	rootsongjc/awesome-cloud-native	A curated list for awesome cloud native tools, software, and tutorials
4th	senecajs/seneca-mesh	Mesh your Seneca.js microservices together—no more service discovery
5th	containous/traefik	The cloud native edge router

interesting framework that actually supports circuit-breakers out of the box (through the Netflix Hystrix library).

Following the "deploy independently" principle, the most popular practice overall, along with CI/CD (Table 5), is containerization, achieved mainly through docker (274 repositories). An important number of sample microservices or sidecar libraries is containerized. Interestingly, #5 combines microservices, mobile devices, and blockchain.

Table 6 Repository analysis by operation principles and query (total repos and top 5 per query)

#repos	Principle/repoId	Query/description
54	**Isolate failure**	*"circuit breaker" OR hystrix*
1st	sqshq/PiggyMetrics	Microservice architecture with Spring Boot, Spring Cloud and Docker
2nd	ERS-HCL/nxplorerjs-microservice-starter	Node JS, Typescript, Express based reactive microservice starter project for REST and GraphQL APIs
3rd	raycad/go-microservices	Golang Microservices Example
4th	spring-petclinic/spring-petclinic-microservices	Distributed version of Spring Petclinic built with Spring Cloud
5th	wso2/msf4j	WSO2 Microservices Framework for Java (MSF4J)
274	**Deploy independently**	*docker OR containers OR kubernetes*
1st	rootsongjc/awesome-cloud-native	A curated list for awesome cloud native tools, software, and tutorials
2nd	benc-uk/smilr	Microservices reference app showcasing a range of technologies, platforms, and methodologies
3rd	dotnet-architecture/eShopOnContainers	Easy to get started sample reference microservice- and container-based application. Cross-platform on Linux and Windows Docker Containers, powered by .NET Core 2.1, Docker engine and optionally Azure...
4th	IF1007/if1007	Desenvolvimento de Aplicaes com Arquitetura Baseada em Microservices
5th	IBM/android-kubernetes-blockchain	Build a blockchain-enabled health and fitness app with Android and Kubernetes
81	**Highly observable**	*monitoring OR logging OR elk*
1st	slanatech/swagger-stats	API telemetry and APM
2nd	hootsuite/health-checks-api	Standardize the way services and applications expose their status in a distributed application
3rd	banzaicloud/pipeline	Pipeline enables developers to go from commit to scale in minutes by turning Kubernetes into a feature-rich application platform integrating CI/CD, centralized logging, monitoring, enterprise-grade...
4th	wso2/msf4j	WSO2 Microservices Framework for Java (MSF4J)
5th	mfornos/awesome-microservices	A curated list of microservice architecture related principles and technologies

The last principle (highly observable) is mainly represented by monitoring and logging techniques (81 repositories), while other practices and technologies (e.g., correlation IDs, analytics, and specific libraries) are not relevant. #1 is a monitoring tool for RESTful APIs (i.e., for most microservices), while #2 and #3 are comprehensive frameworks that include monitoring facilities, among others.

To conclude, let us recap on RQ3, (*How are these projects addressing the aforementioned challenges*?). From our analysis, it can be highlighted that both containerization and CI/CD are the most widespread practices in the microservice ecosystem, followed closely by RESTful specifications. Those correspond to three principles: deploy independently, culture of automation, and hide implementation details, respectively. Mild attention is put on asynchronous communication, service discovery, and monitoring. Finally, the least discussed issues in the GitHub microservice landscape are failure isolation patterns (mostly synonyms with circuit-breakers), orchestration/choreography, and an alarming lack of modeling (DDD, bounded context, etc.) support.

As *threats to validity* of our qualitative assessment, one may note that: (1) queries are not representative of all the keywords and their combinations in Table 2 and (2) queries are constructed using terms related to each other (e.g., REST and OpenAPI/Swagger). This was done to increase the accuracy of results according to the Solr underlying matching mechanisms. We first excluded terms that are not relevant for the queries—i.e., they return (almost) all of the documents as a result, or the inverse (none). Then, we grouped only similar terms (according to their topics) in the same query. This prevents retrieving only general-purpose repositories (e.g., awesome lists[17]) and not the specific, relevant ones for the query at hand—a bias introduced by the coordination factor of TF.IDF weighting [12]. We acknowledge the importance of such lists, but in our case they introduce noise by biasing towards listed tools/frameworks/libraries. Additionally, we are not taking into account historical data of repositories, which may help us track certain behaviors—e.g., the periodicity of releases before/after implementing CI/CD tools, or the impact of containerization in the popularity of a given repository. Besides, this is an ongoing work and performing more comprehensive analysis through, e.g., clustering techniques or topic modeling is the subject of future work.

4.3.1 The Serverless Panorama

Finally, we discuss the current tendencies of serverless microservices (Table 7). We found 35 repositories mentioning serverless (5% of the microservices-related repositories), showing that this technology is still in the early stages of adoption. Through a detailed analysis, one can find example apps using recent serverless platforms—in this case IBM/Apache Openwhisk, but there are others for Google Cloud and Azure functions. Two frameworks to handle the serverless functions' lifecycle, with special focus on deployment, are the serverless framework and UP. Finally, a representative of the event-oriented nature of functions: flogo, and the usual awesome list of serverless solutions.

The most popular languages (when applicable) are JavaScript (mostly for examples, tutorials, and boilerplate applications) and Go (for deployment frameworks such as UP). Popular topics are straightforward: *serverless* and its variety of vendor flavors (Google functions, Azure functions, IBM Openwhisk, AWS Lambda). Apart from that, other popular topics are: *deployment*, since functions involve yet more moving parts than microservices, making deployment even more complex; *Apis*

[17]A common type of curated lists of entries within a given topic—https://GitHub.com/sindresorhus/awesome.

Table 7 Serverless repositories analysis, total and top 5

#repos	Principle/repoId	Query/description
35	**Serverless**	*serverless OR faas*
1st	serverless/serverless	Serverless framework—build web, mobile, and IoT applications with serverless architectures using AWS Lambda, Azure functions, Google CloudFunctions
2nd	anaibol/awesome-serverless	A curated list of awesome services, solutions, and resources for serverless/no-backend applications
3rd	apex/up	Deploy infinitely scalable serverless apps, apis, and sites in seconds to AWS
4th	TIBCOSoftware/flogo	An open source ecosystem of opinionated event-driven capabilities to simplify building efficient and modern serverless functions, microservices, and edge apps
5th	IBM/spring-boot-microservices-on-kubernetes	In this code we demonstrate how a simple Spring Boot application can be deployed on top of Kubernetes

and *integration*, since functions are typically used to generate "entry points" for systems and architectures, probably relying on traditional servers for more complex processing, and finally *events* and *messages* platforms such as Kafka or Mqtt, as functions are typically event driven.

From this analysis, we derive the following challenges and opportunities. First, support for *FaaSification* [39] (i.e., splitting into functions) of legacy or microservices code. Then, tool support for creating and managing complex functions. Usually, functions are used for simple tasks, although they can encapsulate complex microservices such as image recognition [4, 6] or model checking [45], as demonstrated in our previous work. However, this involves trial and error and significant effort, which implies an opportunity to develop supporting techniques, frameworks, and tools. For example, to embed custom languages (e.g., OCAML), improve long-running algorithms, or exploit opportunistic container reuse of the underlying platform as a cache of sorts.

Additionally, some aspects that may not arise from the state of the art should be mentioned here. Serverless is being pushed forward by major vendors, beyond the traditional use cases of short computation as lambda functions.[18] For example, through AWS Fargate for long running functions, or AWS step functions to define complex serverless workflows. Furthermore, solutions such as OpenFaaS and Kubernetes as managed service are blurring the frontier between lightweight containers and serverless functions. The industry tendency is that of starting with containerized microservices (from scratch or from a monolith) and then migrate key features to FaaS to exploit its unique capabilities.

[18] https://aws.amazon.com/en/serverless/serverlessrepo/.

5 Conclusions

This chapter presented an evolutionary perspective that captures the fundamental understanding of microservice architectures, encompassing their whole lifecycle. This is necessary to enable effective exploration, understanding, assessing, comparison, and selection of microservice-based models, languages, techniques, platforms, and tools.

Microservice architectures are fairly new, but their hype and success is undeniable, as big IT companies have chosen them to deliver their business, with Amazon, Netflix, Spotify, and Twitter among those. Due to this traction, the industrial state of practice on microservices has surpassed academic research efforts, which are still at an earlier stage [38]. This resulted in a sort of gap between academic state of the art and industrial state of practice, confirmed by our literature review, which also provides a panorama of available solutions and current and future challenges. Among them are the early use of resilience patterns to design fault-tolerant microservice solutions, the standardization of their interfaces [37], and the development of asynchronous microservices. Special attention should be given to the latent use of the serverless model (FaaS) to design, deploy, and manage microservices. FaaS has the potential to become the next evolution of microservices [9] as event-driven, asynchronous functions, because the underlying constraints have changed, costs have reduced, and radical improvements in time to value are possible.

Finally, we present an analysis of microservices-related repositories on GitHub, which confirmed our findings regarding open challenges, in particular those related to microservices design and modeling (granularity, DDD, and bounded context). This is an on-going work with the main goal of understanding how, and to which degree, software developers are embracing microservice architectures in practice, and which tools and practices are available to overcome their challenges. Our current work encompasses automating the repository analysis through a mining tool, capturing and processing additional metadata, mainly regarding the history of releases, issues, etc. Then, applying natural language processing techniques to infer information (features, topics) from repositories' documentation. Finally we would like to combine this analysis with developers' feedback to understand their vision regarding microservice architectures.

Acknowledgements This work has been partially supported by the GAUSS national research project, which has been funded by the MIUR under the PRIN 2015 program (Contract 2015-KWREMX); and by the grant ANPCyT PICT-2017-1725.

References

1. A. Balalaie, A. Heydarnoori, P. Jamshidi, Microservices architecture enables DevOps: migration to a cloud-native architecture. IEEE Softw. **33**(3), 42–52 (2016)
2. A. Balalaie, A. Heydarnoori, P. Jamshidi, D.A. Tamburri, T. Lynn, Microservices migration patterns. Softw. Pract. Experience **48**(11), 2019–2042 (2018)
3. L. Baresi, M. Garriga, A. De Renzis, Microservices identification through interface analysis, in *European Conference on Service-Oriented and Cloud Computing (ESOCC)* (Springer, Berlin, 2017)
4. L. Baresi, D.F. Mendonça, M. Garriga, Empowering low-latency applications through a serverless edge computing architecture, in *European Conference on Service-Oriented and Cloud Computing* (Springer, Berlin, 2017), pp. 196–210
5. L. Baresi, S. Guinea, A. Leva, G. Quattrocchi, A discrete-time feedback controller for containerized cloud applications, in *ACM Sigsoft International Symposium on the Foundations of Software Engineering (FSE)* (ACM, New York, 2016)
6. L. Baresi, D.F. Mendonça, M. Garriga, S. Guinea, G. Quattrocchi, A unified model for the mobile-edge-cloud continuum. ACM Trans. Internet Technol. **19**(2), 29:1–29:21 (2019). https://doi.org/10.1145/3226644
7. D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H.F. Nielsen, S. Thatte, D. Winer, Simple Object Access Protocol (SOAP) 1.1 (2000). W3C Recommendation
8. G. Casale, C. Chesta, P. Deussen, E. Di Nitto, P. Gouvas, S. Koussouris, V. Stankovski, A. Symeonidis, V. Vlassiou, A. Zafeiropoulos, et al., Current and future challenges of software engineering for services and applications. Proc. Comput. Sci. **97**, 34–42 (2016)
9. A. Cockroft, Evolution of business logic from monoliths through microservices, to functions (2017). https://goo.gl/H6zKMn
10. N. Dragoni, S. Giallorenzo, A.L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, L. Safina, Microservices: yesterday, today, and tomorrow, in *Present and Ulterior Software Engineering* (Springer, Cham 2017), pp. 195–216
11. J. Erickson, K. Siau, Web service, service-oriented computing, and service-oriented architecture: separating hype from reality. J. BD Manage. **19**(3), 42–54 (2008)
12. C. Fautsch, J. Savoy, Adapting the TF IDF vector-space model to domain specific information retrieval, in *Proceedings of the 2010 ACM Symposium on Applied Computing* (ACM, New York, 2010), pp. 1708–1712. https://doi.org/10.1145/1774088.1774454
13. R.T. Fielding, R.N. Taylor, Architectural styles and the design of network-based software architectures, vol. 7. (University of California, Irvine, 2000)
14. V. Garousi, M. Felderer, M.V. Mäntylä, Guidelines for including grey literature and conducting multivocal literature reviews in software engineering. Inf. Softw. Technol. **106**, 101–121 (2019)
15. M. Garriga, Towards a taxonomy of microservices architectures, in *International Conference on Software Engineering and Formal Methods* (Springer, Berlin, 2017), pp. 203–218
16. M. Garriga, A. Flores, A. Cechich, A. Zunino, Web services composition mechanisms: a review. IETE Tech. Rev. **32**(5), 376–383 (2015)
17. M. Garriga, C. Mateos, A. Flores, A. Cechich, A. Zunino, Restful service composition at a glance: a survey. J. Netw. Comput. Appl. **60**, 32–53 (2016)
18. M. Garriga, A.D. Renzis, I. Lizarralde, A. Flores, C. Mateos, A. Cechich, A. Zunino, A structural-semantic web service selection approach to improve retrievability of web services. Inf. Syst. Front. **20**(6), 1319–1344 (2018). https://doi.org/10.1007/s10796-016-9731-1
19. S. Hassan, R. Bahsoon, Microservices and their design trade-offs: a self-adaptive roadmap, in *IEEE International Conference on Services Computing (SCC)* (IEEE, Piscataway, 2016), pp. 813–818
20. S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A.C. Arpaci-Dusseau, R.H. Arpaci-Dusseau, Serverless computation with openlambda. Elastic **60**, 80 (2016)

21. V. Heorhiadi, S. Rajagopalan, H. Jamjoom, M.K. Reiter, V. Sekar, Gremlin: systematic resilience testing of microservices, in *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)* (IEEE, Piscataway, 2016), pp. 57–66
22. V. Issarny, N. Georgantas, S. Hachem, A. Zarras, P. Vassiliadist, M. Autili, M.A. Gerosa, A.B. Hamida, Service-oriented middleware for the future internet: state of the art and research directions. J. Internet Services Appl. **2**(1), 23–45 (2011)
23. E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D.M. German, D. Damian, The promises and perils of mining github, in *Proceedings of the 11th Working Conference on Mining Software Repositories* (ACM, New York, 2014), pp. 92–101
24. B. Kitchenham, Guidelines for performing systematic literature reviews in software engineering. Technical report, ver. 2.3 EBSE Technical Report. EBSE. sn (2007)
25. P. Lemberger, M. Morel, *Why Has SOA Failed So Often?* (Wiley, London, 2013), pp. 207–218. https://doi.org/10.1002/9781118562017.app3
26. J. Lewis, M. Fowler, Microservices (2014). http://martinfowler.com/articles/microservices.html
27. I. Nadareishvili, R. Mitra, M. McLarty, M. Amundsen, *Microservice Architecture: Aligning Principles, Practices, and Culture* (O'Reilly Media, Sebastopol, 2016)
28. S. Newman, *Building Microservices* (O'Reilly Media, Sebastopol, 2015)
29. M.P. Papazoglou, P. Traverso, S. Dustdar, F. Leymann, Service-oriented computing: a research roadmap. Int. J. Coop. Inf. Syst. **17**(02), 223–255 (2008)
30. C. Pautasso, O. Zimmermann, F. Leymann, Restful web services vs. "big" web services: making the right architectural decision, in *17th International Conference on World Wide Web* (ACM Press, New York, 2008), pp. 805–814
31. M. Rahman, J. Gao, A reusable automated acceptance testing architecture for microservices in behavior-driven development, in *2015 IEEE Symposium on Service-Oriented System Engineering (SOSE)* (IEEE, Piscataway, 2015), pp. 321–325
32. J. Rao, X. Su, A survey of automated web service composition methods, in *International Workshop on Semantic Web Services and Web Process Composition* (Springer, Berlin, 2004), pp. 43–54
33. M. Richards, *Microservices vs. Service-Oriented Architecture* (O'Reilly Media, Sebastopol, 2015)
34. C. Richardson, Microservices architecture (2014). http://microservices.io/
35. M. Roberts, Serverless architectures (2016). http://martinfowler.com/articles/serverless.html
36. G. Schermann, J. Cito, P. Leitner, All the services large and micro: revisiting industrial practice in services computing, in *International Conference on Service-Oriented Computing (ICSOC)* (Springer, Berlin, 2015), pp. 36–47
37. A. Sill, The design and architecture of microservices. IEEE Cloud Comput. **3**(5), 76–80 (2016)
38. J. Soldani, D. Tamburri, W.J. Van Den Heuvel, The pains and gains of microservices: a systematic grey literature review. J. Syst. Softw. **146**, 215–232 (2018). https://doi.org/10.1016/j.jss.2018.09.082
39. J. Spillner, C. Mateos, D.A. Monge, Faaster, better, cheaper: the prospect of serverless scientific computing and HPC, in *Latin American High Performance Computing Conference* (Springer, Berlin, 2017), pp. 154–168
40. B. Srivastava, J. Koehler, Web service composition-current solutions and open problems, in *ICAPS 2003 Workshop on Planning for Web Services*, vol. 35 (2003), pp. 28–35
41. M. Stigler, Understanding serverless computing, in *Beginning Serverless Computing* (Springer, Berlin, 2018), pp. 1–14
42. J. Stubbs, W. Moreira, R. Dooley, Distributed systems of microservices using docker and serfnode, in *International Workshop on Science Gateways (IWSG)* (IEEE, Piscataway, 2015), pp. 34–39
43. D.A. Tamburri, R. Kazman, H. Fahimi, The architect's role in community shepherding. IEEE Softw. **33**(6), 70–79 (2016). https://doi.org/10.1109/MS.2016.144

44. G. Toffetti, S. Brunner, S., M. Blöchlinger, J. Spillner, T.M. Bohnert, Self-managing cloud-native applications: design, implementation, and experience. Futur. Gener. Comput. Syst. **72**, 165–179 (2017). https://doi.org/10.1016/j.future.2016.09.002.
45. C. Tsigkanos, M. Garriga, L. Baresi, C. Ghezzi, Cloud deployment tradeoffs for the analysis of spatially-distributed systems of internet-of-things. Technical Report, Politecnico di Milano (2019)
46. M. Villamizar, O. Garcés, L. Ochoa, H. Castro, L. Salamanca, M. Verano, R. Casallas, S. Gil, C. Valencia, A. Zambrano, et al., Infrastructure cost comparison of running web applications in the cloud using AWS Lambda and monolithic and microservice architectures, in *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)* (IEEE, Piscataway, 2016), pp. 179–182
47. N. Wilde, B. Gonen, E. El-Sheik, A. Zimmermann, *Emerging Trends in the Evolution of Service-Oriented and Enterprise Architectures, chap. Approaches to the Evolution of SOA Systems*. Intelligent Systems Reference Library (Springer, Berlin, 2016)
48. F. Zampetti, S. Scalabrino, R. Oliveto, G. Canfora, M. Di Penta, How open source projects use static code analysis tools in continuous integration pipelines, in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)* (IEEE, Piscataway, 2017), pp. 334–344
49. O. Zimmermann, Do microservices pass the same old architecture test? Or: SOA is not dead–long live (micro-) services, in *Microservices Workshop at SATURN Conference* (Software Engineering Institute SEI, Carnegie Mellon University, 2015)