Check for updates

REVIEW

Transition from monolithic to microservice-based applications. Challenges from the developer perspective

[version 1; peer review: 2 approved with reservations]

Antonios Makris[1], Konstantinos Tserpes[1,2], Theodora Varvarigou[2]

[1]Department of Informatics and Telematics, Harokopio University, Athens, Greece
[2]Electrical and Computer Engineering, National Technical University of Athens, Athens, Greece

v1 **First published:** 23 Feb 2022, **2**:24
https://doi.org/10.12688/openreseurope.14505.1
Latest published: 23 Feb 2022, **2**:24
https://doi.org/10.12688/openreseurope.14505.1

Abstract

Microservices have taken the world of software development by storm. Application developers are struggling to understand the new concepts and make the transition by the so-called ``monolithic'' application approach to microservices. This paper touches upon this delicate issue, providing a more concrete view of the developers' concerns together with recent responses to these concerns. The objective is to place the concept of microservices in the most up-to-date context and shed some light in the challenges that puzzle the developers the most while they attempt to make use of this development and design style.

Keywords

microservices, monolithic, workflows, challenges

H2020 This article is included in the Horizon 2020 gateway.

Open Peer Review

Approval Status ? ?

	1	2
version 1 23 Feb 2022	? view	? view

1. **Turgay Celik**, BITES Defense & Aerospace, Ankara, Turkey

2. **Ivan Lanese**, University of Bologna, Bologna, Italy
INRIA, Rocquencourt, France

Any reports and responses or comments on the article can be found at the end of the article.

Corresponding author: Antonios Makris (amakris@hua.gr)

Author roles: Makris A: Conceptualization, Formal Analysis, Investigation, Visualization, Writing – Original Draft Preparation, Writing – Review & Editing; **Tserpes K**: Conceptualization, Formal Analysis, Funding Acquisition, Project Administration, Supervision, Visualization, Writing – Original Draft Preparation, Writing – Review & Editing; **Varvarigou T**: Conceptualization, Project Administration, Supervision, Writing – Review & Editing

Competing interests: No competing interests were disclosed.

Grant information: This work was supported by the CHARITY and ACCORDION projects that have received funding from the European Union's Horizon 2020 research and innovation program under Grant Agreement Nos. 101016509 and 871793, respectively.
The funders had no role in study design, data collection and analysis, decision to publish, or preparation of the manuscript.

Copyright: © 2022 Makris A *et al*. This is an open access article distributed under the terms of the Creative Commons Attribution License , which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

How to cite this article: Makris A, Tserpes K and Varvarigou T. **Transition from monolithic to microservice-based applications. Challenges from the developer perspective [version 1; peer review: 2 approved with reservations]** Open Research Europe 2022, **2**:24 https://doi.org/10.12688/openreseurope.14505.1

First published: 23 Feb 2022, **2**:24 https://doi.org/10.12688/openreseurope.14505.1

Introduction

The term "microservices" refers to a design and development style for applications that favours the decomposition of the application into very small functionalities that can be deployed as standalone, completely independent components. Unlike web services, microservices are expected to be deployed on top of a single infrastructure management fabric that is typically based on container technologies.

The main benefit from working with microservices is that it allows development teams to break up into subgroups that work independently in the language and technologies that they prefer and are more proficient, provided that they can create and support a standalone service that delivers the promised functionality within a container. This allows the development team to stay truly agile.

Equally important is the fact that microservices are reusable in the sense that microservice instance chains can result in different application instances or application workflows. For the purposes of this document we will interchangeably refer to microservice chains as "chains" or "workflows".

These important benefits steer more and more developers towards the use of microservices, often leaving them wondering about how to make the transition from the established approaches of building applications, like SOA (In Table 1 are presented the abbreviations used in the manuscript). The literature commonly refers to the latter as "monolithic" applications in order to highlight the level of decomposition that happens using the microservices approach.

In this work we discuss the set of challenges or concerns that a developer transiting from monolithic to microservices will most likely face. We also provide some hints about how those challenges could be tackled based on the literature as well as the authors' experience. The challenges and hints are classified into nine distinct sections based on the system non-functional requirement that the developers will have to deal with. The addressed non-functional requirements are: communication patterns, performance, orchestration, granularity, discovery and recovery, hosting, scalability, load balancing and security.

Furthermore, we allocate one section at the end of the paper to provide some examples for applications built using the microservices style mainly to strengthen some points to which the authors feel strong about especially about the communication patterns. Finally, we conclude with some thoughts on the progress of this admittedly strong trend.

Table 1. List of Abbreviations.

Abbreviations	Description
SOA	Service Oriented Architecture
REST	Representational State Transfer
API	Application Programming Interface
VM	Virtual Machine
DSL	Domain Specific Language
QoE	Quality of Experience
SLA	Service Level Agreement
COLBA	Chain-Oriented Load Balancing Algorithm
MTLS	Mutual Transport Layer Security
PKI	Public Key Infrastructure
RBAC	Role Based Access Control
ABAC	Attribute Based Access Control
CA	Certificate Authority
RSA	Rivest–Shamir–Adleman
AES	Advanced Encryption Standard
MAC	Media Access Control
ARP	Address Resolution Protocol
GIS	Geographic Information System
GPS	Global Positioning System
ZSM	Zero touch network & Service Management
IoT	Internet Of Things

Communication patterns

A key problem that needs to be addressed when shifting from a monolithic way of developing an application to microservices relates to the communication of the microservices themselves as well as their communication with the external world. Given that they have to be part of the original application but at the same time remain independent components that meet the needs of multiple client services[1], the problem of interoperability and communication needs to be addressed.

The naive approach is to implement specific endpoints within the microservices themselves. This limits the microservice interoperability to the protocols and endpoints it supports and forces the communicating party to learn and implement the microservices' API. For instance, if a microservice implements a RESTful API that allows another component to push data objects to it, then this perhaps excludes streaming applications that could benefit from, *e.g.*, a websocket type of connection. However, the most important drawback, is the elimination of a key strength of microservices, scaling.

To simplistic approach to deal with this issue is to place a load balancer in front of a cluster of similar microservices. The API of the load balancer is known and accessible from anywhere and could be even be standardized so as to be general enough to meet the needs of several of the microservices of the application. The load balancer would distribute the requests across the available instances and perhaps issue scale out commands.

There are several drawbacks with this option. First of all, the client requests are analogous to the number of microservices (or microservices' clusters) which is inefficient and also some microservices might use some not web-friendly protocols (which leads to the main problem of the naive approach).

A much better approach is to use an API Gateway. The API Gateway is a server which is the single entry point into the system and encapsulates the internal architecture and interface of the system. In addition, it provides functions such as authentication, monitoring, load balancing, caching, request shaping and management, and static response handling. The API Gateway is responsible for request routing, composition, and protocol translation. All the client requests passing through the API Gateway which in turn routes these requests to the appropriate microservice. The main benefit of the API Gateway is that it encapsulates the internal structure of the application and provides each client with a specific API reducing network overhead and complexity. Nonetheless, it constitutes a highly available component that must be developed, deployed and managed.

A typical case of an API Gateway is a message broker system, such as RabbitMQ, MQTT or Apache Kafka. These systems allow the automatic creation of workflows through the publish-subscribe (pubsub) pattern and a standard API. In short, a component the produces data objects publishes (pushes) them in a labeled queue and the consumers of those data objects only need to subscribe to this queue so that the system can send them the new data objects. One can create microservice workflows by publishing and subscribing into the appropriate queues. The microservices only need to implement the message broker's API, thus, eliminating the cost of re-building them in cases of the implementation of new workflows since the only thing needed is a reconfiguration.

Further patterns are reviewed in 2 which are capable to solve issues related to communication failures, resource overloading and service relocation at runtime. In a microservice architecture a failure service might affect other services that rely on it. The circuit breaker pattern aims to solve this failure by monitoring the failure rates of a target service. If a large number of requests are failing or the target service becomes slow, the circuit breaker will trip so that further attempts fail immediately. Thus, the circuit breaker pattern contributes to the stability and resilience of both clients and services. Netflix Hystrix library[1] is an open source library that implements the circuit breaker pattern. The basic idea is that circuit breaker is used directly inside the client. Although, the authors propose that the circuit breaker pattern can be deployed also on the side of the services or in proxies that operate between clients and services. The three deployment strategies are: client-side where circuit breakers are placed directly within clients, service-side where circuit breakers can also be implemented on the side of services and proxy where circuit breakers are deployed in a proxy service that sits between clients and services. In a microservice architecture the service instances are dynamically relocated over time due to auto-scaling, failure and upgrade reasons. Therefore, a service discovery mechanism is necessary. There are two main service discovery patterns: client-side discovery and server-side discovery. The client-side discovery pattern handles server or cluster internal interactions. The client is responsible for determining the network locations of available service instances and load balancing requests across them. In server-side discovery the client makes a request to a service via a load balancer. Then, the load balancer queries the service registry and routes each request to an available service instance. In both patterns service instances are registered within the service registry.

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

Performance

It is rather unclear whether an application deployed using a microservice style will result in better or worse performance than the monolithic approach. The monolithic approach has the benefit of less communication and computation overhead given its simpler infrastructure stack. However, a monolithic application is likely to grow large and as a consequence the complexity will increase. The large size of the application results in slow development and deployment. Furthermore, scaling out can be proven a costly and demanding task when different modules have conflicting resource requirements.

On the other hand, microservices can scale dramatically faster than the monolithic application, enhancing the throughput of particular functions of the overall architecture. The loosely coupled components of the microservice pattern speed up the development as the responsibilities of the developer teams are more defined. The application stack is no longer dependent on specific programming languages and frameworks. Continuous delivery can be achieved more efficiently as the deployment may concern only some of the microservices. Nevertheless, the distributed nature of the microservice pattern generally leads to an increased complexity.

The monolithic pattern is proven suitable for simple, lightweight applications while the microservices can be adapted more efficient in complex, evolving applications.

Indeed, microservices are not a panacea. As stated in 3 the microservice approach does not always present better performance compared to the monolithic implementation. The result show 24% less rejected requests, 73.4% less SLO violations and 80% less cost for the monolithic implementation when compared with the microservice approach. Although, the latter processed 14.2% more requests. The increased network communications between microservices result in increased latency and as a consequence a performance degradation is observed. Similarly, another research[4] points out a significant decrease of performance of microservices compared to monolithic due to the network virtualization overhead.

In general there are quite a few factors that may affect the performance of a microservice-based application implementation. Some of these are presented in 5 where the authors are investigating elasticity, load balancing, provisioning variation, infrastructure retention and memory reservation size. The main conclusion is that the elasticity provided by a serverless

computing infrastructure can affect the overall performance. Furthermore, the individual computational requirements of microservices impact load balancing. The results show a well balanced distribution of requests across containers and VMs for both cold and warm service invocations. Concerning provisioning variations, a large number of containers in the same host adversely affect the service performance. For infrastructure retention a performance degradation is observed after a short period of time while a better performance can be achieved with increasing increasing memory reservation size.

Orchestration

A typical scheduling problem in all distributed applications, orchestration in the studied case is linked to the decisions made for the deployment of microservices in the hosting environment and its underlying infrastructure. Criteria such as the application requirements, the current hosting system load and its capacity to timely execute microservices need to be counted in, and the decisions need to be made at runtime. Mitigation plans are also devised and performed also considering their implementation cost. The dependency of microservices to the hosting system is what makes this problem unique, especially when the underlying infrastructure is heterogeneous (*e.g.* Kubernetes on top of multi-cloud providers[6] or hybrid cloud/edge infrastructures).

According to 7 current microservice orchestration solutions present limitations dealing with the microservices dynamic location since a previous registration of each microservice is required which in turn adversely affects scalability at runtime. At first, each new microservice is described and registered and then the orchestration platform is restarted with a view to include this microservice as part of the orchestration. The authors propose a lightweight platform for microservice composition, called Beethoven. The platform is composed of a reference event-driven architecture and an orchestration Domain Specific Language (DSL) used for microservice orchestration. Beethoven relies on the service discovery pattern for registering and describing microservices during orchestrations.

On the mitigation plans front, the orchestrator must bear some core management functionalities such as monitoring, auto-scaling and health management. These should be part of the orchestrator and independent on the managed application. Such a framework is presented in 8 where the authors propose a self-managing architecture for stateless, resilient and scalable management functionalities. One of its unique features is probably the use of a consensus algorithm. The proposed architecture can be employed for self-managing in the microservice architectural pattern where each microservice is deployed as an atomic service with two tokens for service discovery. A local token which is used for microservice cluster forming and leader election and a global token which forms another cluster with the other microservices leaders. The latter is used both for endpoint discovery across microservices and leader election in order to perform all the necessary actions for effective orchestration (e.g. monitoring, auto-scaling) at the service composition level.

In addition, the EC-funded R&I project CHARITY[2] aspires to leverage the benefits of intelligent, autonomous orchestration of cloud, edge, and network resources, to create a symbiotic relationship between low and high latency infrastructures that will facilitate the needs of emerging applications. Thus, for the CHARITY project, a 5-way approach is planned. More specifically, for service placement an artificial intelligence based resource aware orchestration framework is employed (AIRO) which leverages the ZSM (Zero touch network & Service Management) concept, a cloud-native approach, and Machine Learning techniques for efficiently managing network and computation resources. Due to the continuous heavy charge that the previous item will take, a Machine Learning system to make the AIRO completely adaptable is utilized, for which the project leverages the use of GPU-based mechanisms. In addition, a proactive and decentralized mechanism, control the number of application replicas (located in different geographic locations) of the same service in an edge computing platform, while meeting the requested Quality of Experience (QoE) promises. Furthermore, monitoring and prediction mechanisms provide the continuous visualization and perception of the status and the progress of applications and services. Finally, as orchestration and scheduling of emerging applications form the core of the CHARITY architecture, enclosing security for end-to-ends service delivery becomes an important aspect which is achieved in three parts: (i) Secured function execution, (ii) Microservice security and (iii) Learning with programmable switches[9].

Granularity

A critical question in the transformation of a monolithic application to a microservice-based is which functionality to transform to a standalone unit. How far down to the monolithic application functionality to go, so as to identify the microservices? Even though there is no single answer to this problem as it largely depends on the business functions and the various competencies of the development teams (thus, the application developer strategy), there are some observations that may contribute to resolving this issue.

In 10, the author claims that a reasonable technique would be to break down the application to small data transformations operations, *e.g.* a higher order function, tailor-made, low-complexity algorithms, and simple database operations. This approach is meaningful mainly when considering microservice reusability and ease of creating new workflows rather than business factors. Another benefit from breaking up the application in such small parts is the reduction of the size of the codebase into smaller parts.

Furthermore, in 11, evidence is presented about application subsystems which are not suitable to transformed into microservices. These subsystems include microservices that share the same database tables, microservices operations in the middle of other operations and business operations that involve more

[2]https://cordis.europa.eu/project/id/101016509

than one business subsystem on a transaction scope. To this end, a redesign of an event driven integration SOA platform is described in 12. The SOA transformation process involves some steps that include: microservices recognition, component decomposition, local microservice configuration creation etc. Platform redesign provides improved maintenance, production deployment, scalability and resource management.

Service discovery and recovery

One of the greatest challenges in achieving high scalability in a microservice system is the service discovery problem. The issue here is for the orchestrator to be aware of the location of microservices, especially in the context of "fluctuating" underlying infrastructures like cloud and cloud/edge resources.

This problem is not new and it is typically addressed by a service registry in SOA. The novelty, comes with message passing, the overarching communication protocol for most microservice-based applications. The key concept is that microservices are obliged to detect the message broker once they become available and publish messages or subscribe to the appropriate queue, thus becoming a member of the workflow without further knowledge of it (*i.e.* application agnostic).

Alternatives to this standard approach have been presented in the literature, like the work of Stubbs *et al.*[13]. The authors propose a fully decentralized open source solution called Serfnode which contains homogeneous nodes in a completely connected graph avoiding the concept of master node and a central registry. Serfnode advertises Docker containers to an existing dynamically formed cluster of Serfnode containers and provides service discovery, monitoring and a self-healing mechanism. The functionality that Serfnode provides to an arbitrary image is cluster membership, monitoring and event handling. The extensibility of the current solution is provided by describing a file synchronization problem on top of Serfnode and the results show that it performs quite satisfactory even with with a large number of read and write operation.

Hosting environments

Microservices are tightly coupled to containerization technologies. A container enables the microservice to operate autonomously, as if it was in its own system. The container model virtualizes operating system calls under the same kernel instance that can isolate and control resources for a set of processes. This provides the illusion of a unique system that consumes only a small portion of the resources of the underlying stack. The transition from a monolithic application to a microservice-based needs also to consider the shift from bare metal or cloud resources to containers. The problem comes with leveraging on the abilities and coping with the particularities of the container management environments (microservice hosting systems) in the cloud such as Docker Swarm, Kubernetes and OpenStack Neutron.

14 raises some open issues related to the need for a scheduling platform for managing an ecosystem of microservices. The heterogeneous configuration, performance requirements and characterization of microservices create the need for autonomous decision-making techniques and application-agnostic frameworks. This concept is also addressed in the frame of the ACCORDION project[3]. The idea is to distribute the application microservices on top of opportunistically created edge resource pools, with the intention to deliver improved QoE to the end user. The expected heterogeneity of the infrastructure and the relaxed QoS guarantees from the edge resource providers create the need for continuous adaptation frameworks.

Furthermore, this framework needs to cater for the need of continuous delivery of microservices on top of these "volatile" infrastructures. The work of Singh and Peddoju[15] can be regarded as a starting point as it addresses continuous integration and continuous delivery in microservice environments. The authors propose that each newly created service registers itself and stores its configuration. Then the deployment server stores the configuration information to the configuration storage. Subsequently, each microservice is able to communicate with other microservices by simply access this configuration information. Also, this information is used from an API proxy for configuration changes at runtime. Due to the need of constantly updated microservices, the proposed model is able update the services with minimum downtime. The results show the superiority of the microservice-based model over the monolithic design as it presents low response times and high throughput.

Scalability

Horizontal scalability constitutes the dominant solution in production cloud computing environments in which a service instance is spawned to a cluster of replicas to increase the overall system capacity. This scaling model is employed in the microservice architecture pattern where each microservice can be scaled out by placing new instances separately according to the associated load. Most application platforms provide automate auto-scaling features that are capable of adjusting the performance of microservice instances depending on metrics such as processor load, memory utilization or network traffic.

The problem occurs when a single scaling decision is taken disregarding the context of the workflow. On one hand the application provider has to manually determine some empirically suitable configuration of initial scaling factors and thresholds and on the other hand most auto-scaling systems do not consider the dependencies or the specificities of each microservice.

The expected behaviour is that by scaling a microservice the performance would increase. However, the actual behaviour presents deviations as the performance at some point is decreasing or totally tips the balance[16]. We coin the term "scaling paradox" to describe this phenomenon. Therefore, scaling out an overloaded microservice is not always the solution to overcome performance problems. The application needs to know the best combination of microservices instances for the current workload. The decision to scale out or in cost-effectively

[3]https://cordis.europa.eu/project/id/871793

and cost-predictably, depends on the best combination of microservices.

The work in 17 proposes a solution for this scaling paradox problem. The proposal involves scaling within well-defined boundaries in which every scaling decision, contributes to the overall system performance. The method finds the optimal scale combination depending on specific requirements on performance and available resources.

A relevant concern is on message-based communication within microservice based applications. The state of a message queue can be a challenge if the number of incoming messages is bigger that the number of outgoing messages, leading to degradation concerning performance and reliability. Some challenges concerning scaling microservices consuming messages from message queues, related on how to recover or avoid congested queues and how to utilize information of the state of the queue to avoid over- or under- provisioning of consuming microservices. In 18 and 19 the authors discuss a similar approach where the message queue plays the governing role in scaling decisions. In both cases, the authors adopt a rule-based, auto-scaling setup with the exception that in 18 they make a distinction for two classes of microservices: I/O- and computer- intensive. The common objective is to prevent overloaded queues and to avoid SLA violations. The results show that scaling decisions based on message queue metrics are much more resilient to microservice characteristics variations.

Load balancing

Load balancing remains a crucial question to be answered by the developer when moving towards microservices. Load balance refers to the decision-making problem of distributing the application load towards multiple instances[20–22] of the same microservice. The problem starts once you break down the operations into multiple components. Then it becomes unclear which one may become a bottleneck and what will be the criteria and the means to distribute the load. This is especially true in the light of the use of different technologies to develop the various microservices. Even more, microservice workflows which may account for different applications, may share certain microservices, creating a context where the load must be distributed to each instance of the microservice based on different criteria. The existing load balancing solutions do not account for the requests heterogeneity and inter-chain competition as stated in 23.

Given a message-based communication architecture, the authors in 24 attempt to address the above issues. They propose a chain-oriented load balancing algorithm which based solely on message queues. The algorithm balances the load (requests) based on microservice requirements of chains. Specifically, the approach differentiates between intra-chain and inter-chain load and determines the number of instances of a microservice based on both contexts. The evaluation results show that an instance-oriented solution handles well inter-chain competition but presents communication overhead while the microservice-oriented solution exhibits the exact opposite behavior.

A rule of thumb is to break up the application operations into so small pieces so that they serve a simple function with simple input/output. This will result in simple performance criteria, thus simplifying the decisions to be made by the load balancer.

An aggregation of these ideas is presented in 23 where the authors present the disadvantages of having a load balancer per microservice cluster of instances and provide evidence of the superiority of a Chain-Oriented Load Balancing Algorithm (COLBA) using message queues. The idea is that for each chain the microservice instances are maintaining a different message topic in the message broker allowing the latter to balance the load based on request heterogeneity and competition across chains.

Security

New security challenges arise from the microservice architectural style, that were not present in the traditional monolithic applications. Although microservices constitutes a well-established technology pattern, attention to security, as commonly happens in these cases comes late. The fine granularity that characterise the microservice architecture leads to complex communications and the large number of messages exchanged between services can be leaked by a malicious software. A microservice architecture consist of a large number of microservices and hence of a large number of entry points. If an attacker exploits a vulnerability in a single entry point, then the overall system can be affected service by service.

Lightweight, scalable and easy to automate are just some features for effective microservice security. As microservice security is a multifaceted problem and relies on underlying technologies, it needs to be decomposed into some components[25]. decomposes microservice security into six layers: hardware, virtualization, cloud, communication, service and orchestration. Hardware and virtualization constitutes the bottom level and are the least accessible layers to an attacker. On the cloud level, the cloud provider can be itself a threat while a network attacker can be a major concern on the remaining layers. The straightforward security approach of microservice-based systems is the perimeter defense[26]. However due to the architectural microservice pattern, an attacker is able to manipulate all the nodes of a microservice network from a single compromised service through malicious requests. Thus, the concept of “trust no one” is adopted, where multiple security mechanisms are placed at different levels of the system. Some security practices used are the Mutual Transport Layer Security (MTLS) with a self-hosted Public Key Infrastructure (PKI) and tokens utilization with local authentication. As an example, MTLS is used by all the nodes in Docker Swarm for authentication and network traffic encryption and a PKI is deployed for node identification. Also, a PKI based on short-lived certificates for TLS with mutual authentication is utilized in Netflix. On the other hand, token-based authentication is a well known security mechanism that relies on cryptographic objects.

Furthermore, two promising approaches concerning fine-grained authorization are: security tokens for user authorization where control mechanisms such as Role Based Access Control (RBAC) and Attribute Based Access Control (ABAC) are used and inter-service authorization where a separated signing certificate created per microservice type. Due to the lack of a standard way to cope with the security concerns, the authors propose a microservice security framework called MiSSFire which provides a standard way to include security mechanisms into microservices. The establishment of trust between individual microservices is the main challenge that the framework has to address. The security mechanisms that used are based on MTLS and principal propagation. The two infrastructure services that bundled to the framework are: CA service which generates a self-signed root certificate that enables MTLS between microservices and Reverse STS that generates security tokens. The performance of the framework is evaluated based on a microservice-based bank system called MicroBank. The experiments show negligible latencies concerning network overheads for secure microservice communications.

A taxonomy of security issues in microservices is presented in 27. Specifically, it focuses on service communications and investigates security vulnerabilities related to four aspects: containers, data, permission and network. Containers are widely used in the microservice approach but present security issues. The same kernel utilization as well as the exposing of the same host on different container instances, could lead to unauthorized access and other potential issues related to DoS attacks, poisoned images, escape from containers etc. Isolation of containers is an important point for containers security. Several security issues can arise in the default container network, which is vulnerable to ARP spoofing and MAC flooding attacks since a filtering mechanism is not used on the network traffic passing through the bridge. For data confidentiality, integrity and availability, the microservices provider must be able to provide an encryption schema, access controls and a safe storage. Another important dimension of security that should be considered is the permission. It is necessary for every service in a microservice architecture to be verified. Finally, the network security constitutes an important aspect because it can ensures secure communications among the microservices. The authors suggest that considering the security complexity introduced in a microservice architecture, a multiple layer system that can prevent a single point of failure is necessary. They propose an ideal system that addresses the main four aspects described above. SELinux are adopted in the VMs for containers security while the Advanced Encryption Standard (AES) and the RSA encryption algorithm are utilized for data security. The Spring Cloud Security framework in combination with OAuth2 authentication system ensures the security of authentication and authorization. Finally, the SDN-based security is used for network protection.

Microservices applications

In what follows we present a number of applications that they have dealt with the abovementioned issues and promoted the use of a message-based communication pattern in order to tackle them.

A Hazmat transportation management system based on the microservice architecture is presented in 28. Every day hazardous substances are transported inside residential areas and potential accidents can lead to injuries, environmental pollution, evacuations etc. A system which is able to predict and minimize the risk of such incidents and their potential consequences is necessary. The objective is to monitor and control the transportation of hazardous materials. The system is built based on the microservices architecture over a cloud environment and provides the following services: pre-transportation which prepares and sends the shipping paper, routing for finding the least-risky routes by employing a GIS database with risk costs and the pgRouting extension, monitoring and tracking which visualize the vehicle moving based on GIS and GPS, data collection that collects sensor data from vehicle, substance and risk knowledge that permits to retrieve hazardous material description stored in a database, transport documentation for shipping dangerous materials by road, historical information thus improve reliability of the system and finally alert and query information which send warnings and alerts notification in case of dangerous situations. The collection of data and the monitoring service are the most basic requirements. For efficient data transfer, a lightweight bandwidth-efficient protocol is used which provides a simple, reliable stream of data. The Message Queue Telemetry Transport (MQTT) protocol is in essence a publish/subscribe mechanism. The data collection architecture can be classified into the IoT devices which represents the sensor layer and is responsible for obtaining various information of sensors, the MQTT broker (Mosquitto server) which receives and filters all messages, the MQTT subscribe server which consist of the MQTT client (Node.js server) configured to subscribe to the published messages and storing them in a database server (MongoDB) and the clients which represent the application layer and exploits the collected data in order to provide monitoring. The best routes for the transportation can be extracted by combining the GIS with the spatial database and the pgRouting bi-directional A* algorithm. The data used as input to the routing service are: spatial data which represents the transportation network as a collection of arcs with nodes loaded from OpenStreetMap through QGIS server and stored in the spatial database PostgreSQL/PostGIS and attribute data which represents the values of risk on an arc. Finally, for containers orchestration, scheduling and deployment, Docker Engine, Docker Machine, Docker Compose and Kubernetes are used.

In 29 the microservice architecture is adopted to built a Smart City IoT platform. The implementation of such platform, involves building a large scale IoT system and on the top of it a service platform which would provide access to the IoT data and services. The DIMMER platform aims to increase the energy efficiency of a city at the district level. It includes heterogeneous Sensor Technologies (monitoring systems) and District Information Models (GIS, BIM, SIM) integrated in the Service Platform that is used by Smart City Applications (Web, Desktop, Mobile, Semantic Web clients). The Service Platform includes Middleware services, as well as a number of platform services for Smart City applications (Smart City Services). Middleware Services are responsible for gathering and

processing sensor data and integrate heterogeneous IoT devices and ICT systems in the platform. In the DIMMER platform a decentralized data management approach of microservices is employed to manage the IoT devices meta-data. The Resource Catalog service provides information about the devices configuration, deployment and supported communication protocols while the Semantic Datastore service provides additional attributes and relations to other entities for the same devices. Furthermore, the middleware includes the Historical Datastore service which used for storing sensor data, a Message Broker as a publish/subscribe communication [mechanism and Service Catalog for service discovery. The Smart City Services can be decomposed into services that expose district information model, an energy efficiency engine and energy data simulator and a set of services that provide context awareness features to the applications.

Another study[30] investigates different patterns and aspects in the microservices approach and examines how these practices can be integrated in the IoT. In the microservice architecture, individual distributed interconnected services are designed to work together and structure an application. The interoperability of IoT services and the creation of value added applications could benefit by employing the same architectural design. The aspects compared, related to self-containment, monitoring and fault handling. Self-containment property focuses on separation of the functionality and enforces isolation via independently deployable units. By adopting this property in IoT, several benefits arise such as independent evolution of services, easier deployment and better decoupling between services. Monitoring is a process of reporting, gathering and storing information. Each service should provide an interface about its health status in order to prevent other services to call a broken one. Microservices and IoT employ the concept of circuit breaker in conjunction with the load balancer pattern. The circuit breaker prevents messages delivered to broken services and enables the load balancer to distributes the workload only on "healthy" services. In conclusion, this research work supports that architectural goals of microservices and IoT are quite similar and IoT could benefit from aspects used in the microservices approach.

Following this direction, 31 presents a vision of applying microservice architecture in an IoT system. Several challenges concerning IoT systems have been already addressed and the Internet consist the backbone for the IoT. However, existing IoT systems facing several well-known problems including interoperability, security flaws, heterogeneity of technologies and protocols used, power limitations etc. In this research, "things" are not treated as atomic elements of the system but rather follow the SOA approach where IoT is a network of services. An IoT node is a smart object that provides services over the network. Thus, the focus is shifted to the level of data and services rather on devices and communication. However, the SOA approach is heavyweight and consist of centralized service models. As a solution the microservice pattern is applied to IoT systems where each component is independently developed and deployed. Because IoT systems have important differences with cloud-or web-centric patterns, the microservice pattern is combined with complementary patterns which are able to solve several issues concerning the internet of things. These patterns include API Gateway, distribution, service discovery, containers and access control. Two case studies are employed and the results show that in order to successfully apply microservices in IoT systems many trade-off should be considered and open question to be addressed.

Conclusions

The motivation behind this work was to highlight the design and development concerns that come with the decision of adopting a microservice-based design and development approach when implementing an application. Of particular interest was to identify certain solutions that will successfully navigate the transition from the monolithic type of application development to microservices.

A major conclusion of this work is that there is a large body in the literature that views microservices as a variation of SOA. This is usually implicit: *e.g.* through claims that microservices need to cater for access from external parties, microservices support single workflows, etc. However, further friction with microservice development and investigation has revealed that this style of design and development is more suitable for reusable, generic operations that are shared across multiple workflows. This fact shift the emphasis of the developers to the support of automated workflows, in the sense that they have to be adaptive when it comes to resilience, performance, security and other suchlike non-functional requirements.

Towards this direction, the application developers have found a powerful alliance in the communities supporting container technologies and their hosting environments. The advances brought in Docker, Linux containers and recently in Unikernels[32] and chiefly in Kubernetes, Swarm and relevant overlays such as OpenShift, have greatly contributed towards alleviating the burden of the developers to implement their applications in microservices.

As a result of this work, the authors would like to propose the following rules of thumb to the developers wishing to adopt microservices: breaking up the application into very basic autonomous operations facilitates the delivery of a single performance from the microservices disregarding the application. This is beneficial in cases where the same microservice (but not necessarily the same instance) is shared across different application workflows as it implies standard performance expectations that simplify the scalability and load distribution problems and limits the communication possibilities range, thus limiting the security vulnerabilities. The same applies for the adoption of an application-independent design for the microservices. Thus, instead of making tailored, fine grained APIs for each microservice, the developer must rely on generic message brokers and pubsub.

As a final remark, we would like to note that we have intentionally avoided to use the term "microservice architecture" which is used commonly in the literature due to the fact that they consider it overloaded. That is, the granularity of the

architecture may differ when viewed from different angles, often resulting in a confusing nested concept (e.g. message-passing, container-based, microservice, distributed architecture). It seems much more accurate to refer to microservices as a design and development style or even a tangible computing unit.

Data availability

No data are associated with this article

Ethics and consent

Ethical approval and consent were not required.

References

1. Zhao JT, Jing SY, Jiang LZ: **Management of api gateway based on micro-service architecture.** *J Phys Conf Ser.* IOP Publishing, 2018; **1087**(3): 032032.
Publisher Full Text
2. Montesi F, Weber J: **Circuit breakers, discovery, and API gateways in microservices.** *CoRR.* abs/1609.05830, 2016.
Reference Source
3. Abdullah M, Iqbal W, Erradi A: **Unsupervised learning approach for web application auto-decomposition into microservices.** *J Syst Softw.* 2019; **151**: 243–257.
Publisher Full Text
4. Ueda T, Nakaike T, Ohara M: **Workload characterization for microservices.** In: *2016 IEEE international symposium on workload characterization (IISWC).* IEEE, 2016; 1–10.
Publisher Full Text
5. Lloyd W, Ramesh S, Chinthalapati S, *et al.*: **Serverless computing: An investigation of factors influencing microservice performance.** In: *2018 IEEE International Conference on Cloud Engineering (IC2E).* IEEE, 2018; 159–169.
Publisher Full Text
6. Anastasi GF, Carlini E, Dazzi P: **Smart cloud federation simulations with cloudsim.** In: *Proceedings of the first ACM workshop on Optimization techniques for resources management in clouds.* 2013; 9–16.
Publisher Full Text
7. Monteiro D, Gadelha R, Maia PHM, *et al.*: **Beethoven: an event-driven lightweight platform for microservice orchestration.** In: *European Conference on Software Architecture.* Springer, 2018; **11048**: 191–199.
Publisher Full Text
8. Toffetti G, Brunner S, Blöchlinger M, *et al.*: **An architecture for self-managing microservices.** In: *Proceedings of the 1st International Workshop on Automated Incident Management in Cloud.* 2015; 19–24.
Publisher Full Text
9. Makris A, Boudi A, Coppola M, *et al.*: **Cloud for holography and augmented reality.** In: *2021 IEEE 10th International Conference on Cloud Networking (CloudNet).* IEEE, 2021; 118–126.
Publisher Full Text
10. Tserpes K: ***stream-msa*: A microservices' methodology for the creation of short, fast-paced, stream processing pipelines.** *ICT Express.* 2019; **5**(2): 146–149.
Publisher Full Text
11. Levcovitz A, Terra R, Valente MT: **Towards a technique for extracting microservices from monolithic enterprise systems.** arXiv preprint arXiv: 1605.03175, 2016.
Reference Source
12. Djogic E, Ribic S, Donko D: **Monolithic to microservices redesign of event driven integration platform.** In: *2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO).* IEEE, 2018; 1411–1414.
Publisher Full Text
13. Stubbs J, Moreira W, Dooley R: **Distributed systems of microservices using docker and serfnode.** In: *2015 7th International Workshop on Science Gateways.* IEEE, 2015; 34–39.
Publisher Full Text
14. Fazio M, Celesti A, Ranjan R, *et al.*: **Open issues in scheduling microservices in the cloud.** *IEEE Cloud Computing.* 2016; **3**(5): 81–88.
Publisher Full Text
15. Singh V, Peddoju SK: **Container-based microservice architecture for cloud applications.** In: *2017 International Conference on Computing, Communication and Automation (ICCCA).* IEEE, 2017; 847–852.
Publisher Full Text
16. Makris A, Tserpes K, Andronikou V, *et al.*: **A classification of nosql data stores based on key design characteristics.** In: *Procedia Comput Sci.* 2016; **97**: 94–103.
Publisher Full Text
17. Ramírez López M, Spillner J: **Towards quantifiable boundaries for elastic horizontal scaling of microservices.** In: *Companion Proceedings of the10th International Conference on Utility and Cloud Computing.* 2017; 35–40.
Publisher Full Text
18. Gotin M, Lösch F, Heinrich R, *et al.*: **Investigating performance metrics for scaling microservices in cloudiot-environments.** In: *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering.* 2018; 157–167.
Publisher Full Text
19. Tserpes K, Pateraki M, Varlamis I: **Strand: scalable trilateration with node.js.** *J Cloud Comp.* 2019; **8**(1): 1–16.
Publisher Full Text
20. Makris A, Tserpes K, Anagnostopoulos D: **Load balancing in in-memory key-value stores for response time minimization.** In: *International Conference on the Economics of Grids, Clouds, Systems, and Services.* Springer, 2016; **10382**: 62–73.
Publisher Full Text
21. Makris A, Tserpes K, Anagnostopoulos D: **A novel object placement protocol for minimizing the average response time of get operations in distributed key-value stores.** In: *2017 IEEE International Conference on Big Data (Big Data).* IEEE. 2017; 3196–3205.
Publisher Full Text
22. Makris A, Tserpes K, Anagnostopoulos D, *et al.*: **Load balancing for minimizing the average response time of get operations in distributed key-value stores.** In: *2017 IEEE 14th International Conference on Networking, Sensing and Control (ICNSC),* IEEE, 2017; 263–269.
Publisher Full Text
23. Niu Y, Liu F, Li Z: **Load balancing across microservices.** In: *IEEE INFOCOM 2018-IEEE Conference on Computer Communications.* IEEE. 2018; 198–206.
Publisher Full Text
24. Dragoni N, Lanese I, Larsen ST, *et al.*: **Microservices: How to make your application scale.** In: Alexander K. Petrenko and Andrei Voronkov, editors, *Perspectives of System Informatics*. Springer International Publishing. ISBN 978-3-319-74313-4. 2018; 95–104.
Reference Source
25. Yarygina T, Bagge AH: **Overcoming security challenges in microservice architectures.** In: *2018 IEEE Symposium on Service-Oriented System Engineering (SOSE),* IEEE, 2018; 11–20.
Publisher Full Text
26. Jander K, Braubach L, Pokahr A: **Practical defense-in-depth solution for microservice systems.** *Journal of Ubiquitous Systems & Pervasive Networks.* 2019; **11**(1): 17–25.
Publisher Full Text
27. Yu D, Jin Y, Zhang Y, *et al.*: **A survey on security issues in services communication of microservices-enabled fog applications.** *Concurrency and Computation: Practice and Experience.* 2019; **31**(22): e4436.
Publisher Full Text
28. Cherradi G, Bouziri AEL, Boulmakoul A, *et al.*: **Real-time microservices based environmental sensors system for hazmat transportation networks monitoring.** *Transp Res Proc.* 2017; **27**: 873–880.
Publisher Full Text
29. Krylovskiy A, Jahn M, Patti E: **Designing a smart city internet of things platform with microservice architecture.** In: *2015 3rd International Conference on Future Internet of Things and Cloud.* IEEE, 2015; 25–30.
Publisher Full Text
30. Butzin B, Golatowski F, Timmermann D: **Microservices approach for the internet of things.** In: *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA).* IEEE, 2016; 1–6.
Publisher Full Text
31. Lu D, Huang D, Walenstein A, *et al.*: **A secure microservice framework for iot.** In: *2017 IEEE Symposium on Service-Oriented System Engineering (SOSE).* IEEE, 2017; 9–18.
Publisher Full Text
32. Kuenzer S, Santhanam S, Volchkov Y, *et al.*: **Unleashing the power of unikernels with unikraft.** In: *Proceedings of the 12th ACM International Conference on Systems and Storage.* 2019; 195.
Publisher Full Text

Open Peer Review

Current Peer Review Status: ? ?

Version 1

Reviewer Report 12 October 2022

https://doi.org/10.21956/openreseurope.15658.r30240

© **2022 Lanese I.** This is an open access peer review report distributed under the terms of the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

?

Ivan Lanese
[1] University of Bologna, Bologna, Italy
[2] INRIA, Rocquencourt, France

Summary
The paper highlights challenges that are faced when moving from a monolithic approach to a microservice one. For each challenge, some solutions from the literature are discussed. The paper also includes a description of some applications developed using the microservice approach.

Good points
- The topic is of interest.
- The paper is easy to read.

Bad points
- I found the section on Orchestration confusing. In my understanding orchestration is how the different microservices are invoked and how they communicate, typically with a centralized control, as opposed to choreographies. Here instead the emphasis seems more on deployment and resource management. This is a relevant point, but if this is the aim I would at least change the title and clarify the scope of the section.
- I found the section on granularity not satisfactory. After a good statement of the problem, the authors immediately move to present 3 specific approaches from the literature. I would have liked to see first a discussion of the baseline approach, e.g. following the notion of bounded context.
- The section on service discovery says (correctly) that message passing is dominant in microservices, deducing (incorrectly in my opinion) that the key task in such cases is to find the message broker. I disagree since in many cases message passing is not mediated by a message broker (e.g., approaches based on choreographies or workflows).

- In the section about hosting environment before presenting specific approaches, I would describe the baseline approach, as provided e.g. by Kubernetes (named but not described).

- The authors state that "A major conclusion of this work is that there is a large body in the literature that views microservices as a variation of SOA." While I agree with the statement, in the paper this comes as a surprise since this point was never raised before as far as I can see.

Detailed comments

- p. 3, c. 2, l. -4: are analogous to the number of microservices: I do not understand this sentence.

- p. 8, c. 1, 8: please provide a citation for MiSSFire; also RBAC and ABAC may deserve a citation.

- there are a few typos, e.g.:
  To simplistic approach
  could be even be
  a component the produces
  a failure service
  SLO

Is the topic of the review discussed comprehensively in the context of the current literature?

Yes

Are all factual statements correct and adequately supported by citations?

Yes

Is the review written in accessible language?

Yes

Are the conclusions drawn appropriate in the context of the current research literature?

Partly

Competing Interests: No competing interests were disclosed.

Reviewer Expertise: I have been working on microservices. I come from the area of formal methods and programming languages

I confirm that I have read this submission and believe that I have an appropriate level of expertise to confirm that it is of an acceptable scientific standard, however I have significant reservations, as outlined above.

Reviewer Report 21 March 2022

https://doi.org/10.21956/openreseurope.15658.r28696

© 2022 Celik T. This is an open access peer review report distributed under the terms of the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

?

Turgay Celik
BITES Defense & Aerospace, Ankara, Turkey

This paper tries to summarize the state-of-the-art about microservices, especially in the context of migration from monolith architectures. Each aspect (comm patterns, scalability etc) is handled in a short summary. Pros-cons comparison tables would be useful to provide more detail. The current version of the paper is like a very high-level overview.

The language of the paper shall be reviewed. For example, migration from monoliths to microservices is called with different terms such as "transiting" and "shifting".

The paper needs a proof-read and revision e.g.:

- "In a microservice architecture a failure service might affect other services that rely on it. "
- "... can be achieved with increasing increasing memory reservation size."
- "... enables the load balancer to distributes the workload only on "healthy" services"

Is the topic of the review discussed comprehensively in the context of the current literature?
Yes

Are all factual statements correct and adequately supported by citations?
Yes

Is the review written in accessible language?
Partly

Are the conclusions drawn appropriate in the context of the current research literature?
Partly

Competing Interests: No competing interests were disclosed.

Reviewer Expertise: distributed systems, microservices, cloud

I confirm that I have read this submission and believe that I have an appropriate level of expertise to confirm that it is of an acceptable scientific standard, however I have significant reservations, as outlined above.