Disambiguation and Comparison of SOA, Microservices and Self-Contained Systems

Tomas Cerny
Computer Science
Baylor University
Waco, TX, USA
tomas_cerny@baylor.edu

Michael J. Donahoo
Computer Science
Baylor University
Waco, TX, USA
jeff_donahoo@baylor.edu

Jiri Pechanec
Red Hat
Brno, Czech Republic
jpechane@redhat.com

ABSTRACT

There is an industrial shift from Service-Oriented Architectures (SOA) into Microservices; however, a quick review of online resources on these topics reveals a range of different understandings of these two architectures. Individuals often mix terms, grant false advantages or expect different quality attributes and properties. The purpose of this paper is to provide readers a solid understanding of the differences between these two architectures and their features. We provide both research and industry perspectives to point out strengths and weaknesses of both architectural directions, and we point out many shortcomings in both approaches that are not addressed by the architecture. Finally, based on this we propose challenges for future research.

Keywords

SOA; Microservices; Architectures; Self-contained Systems

1. INTRODUCTION

Over decades, industry demands push software design and architectures to various directions and corners. Ever-growing complexity of enterprise applications, their requirements, change and evolution management endorsed the rise of architectures such as Common Object Request Broker Architecture (CORBA), Java RMI or Enterprise Service Bus. Service-Oriented Architecture or just - SOA, became the answer to multiple industrial demands for large enterprises, removing its predecessors from the market, however, it seems that even SOA has its successor a Microservice Architecture (µService).

Both SOA and µServices suggest decomposing of systems into services available over a network and integratable across heterogeneous platforms. In both approaches, services cooperate to provide functionality for the overall system, and thus both share the same goal, however, the path to achieving the goal is different. General use of SOA goes the direction preferring the design of system decomposition info simple services emphasizing service integration with smart routing mechanisms for the entire company's IT. The smart routing mechanism provides a global governance or so-called centralized management and is capable of enforcing business processes on top of services, message processing, service monitoring or even service control. SOA services are uncoupled reacting on events without knowing the trigger for the events, a new service can be easily integrated by reacting to such event. For instance, one service can write an invoice and another can initiate delivery. A new logging system can just listen to the event not impacting other services.

The µServices, on the contrary, suggest decomposition preferring smart services while considering simple routing mechanisms [3], without the global governance notable in SOA. This naturally leads into higher service autonomy and decoupling, since services do not need to agree on contracts on the global level. However, services become responsible for business processes management as well as for interaction with other services.

There are however other perspectives to consider. SOA's difficulty comes with the complex stack of the web service protocols necessary for transactions, security, etc., spanning through all the interoperable services. Moreover, since SOA enables building business processes on top of the services on the integration level, it brings flexibility to change the processes, but at the same time binds all services to a single general context. As the consequence service contracts expressing the service operation expose its data types leading into dependencies regarding deployments [20], in an extreme case leading into one large monolith deploy.

In µServices, it is possible to involve light and heterogeneous protocols for service interaction. Each service only maintains its context and its own perspective over particular data, however, possibly leading into duplicities across services. If deployment dependencies exist among services, they are on a much lower scale since no general context, and no centralized governance exists. The primary goal of µService is to enable independent service deployments and evolution.

The above features lead into multiple consequences. For

instance, it is fairly easy to selectively deploy overloaded µService in order to scale it, however, it is not easy in SOA [21]. The SOA integration mechanism and centralized governance predetermine a bottleneck when the system needs to scale up. When a scaling issue arises in SOA feature it is hard to determine where the bottleneck is, whether it is the service itself, the integration or it is in a shared database. Self-contained µServices are more efficient when it comes to elasticity, scalability, automated and continuous deploy with a fast response on demands. The above characteristics make µServices more cloud-friendly [12].

There are however the reversed consequences requiring µServices to restate and redefine data definitions or even business rules across services, introducing replicas in databases, lacking a centralized view on the overall system processing, rules, and constraints, their correlation, etc.

The industry seems to be in the shift towards µServices, leaving SOA behind. However, µServices are not a superset of SOA and many its challenges do not exist in SOA. Various interpretations of these architectures [10] put part of the community on the side considering µServices to be a subset of SOA, although many others [18] see them as distinct architectures.

In this paper, we aim to describe differences of SOA and µServices so that reader gets a clear picture what to expect from on or the other. It points out strengths and weaknesses of the two. Moreover, we want to disambiguate the terms and give the reader a solid understanding of benefits of the particular approach. Since µServices seem to be the future direction for the industry, we emphasize our focus on challenges this architecture has to face. The next two sections introduce SOA and µServices in details. Their comparison is the subject of section 4. Open research challenges elaborate section 5. The last section concludes the paper.

2. BACKGROUND

SOA and µServices are two major architectures that are being used for the system integration decomposing systems into services. The question is how to coordinate services to achieve particular use cases. In general, there are two well-accepted approaches: centrally orchestrated and independent or distributed. Centrally orchestrated approaches seem to be a common pattern for SOA and the decentralized seems to be the pattern for µServices. For the purpose of this paper, we think of SOA as the centralized coordination and µServices the decentralized version.

In this section, we provide definitions of terms that help us to contrast these two architectures and recognize their differences.

Service is a reusable software functionality usable by various clients for different purposes enforcing control rules. It implements a particular element of the domain, defines its interface, it can be used independently over the network. While involving well-known interfaces and communication protocols it brings platform independence. In SOA services are registered in a directory or a registry to easily locate them. In order to reduce coupling, services are composed to produce an outcome.

The interaction patterns for centralization and decentralization are called *orchestration* and *choreography*. These indicate how services collaborate, how the sequence of activities look like, or how is the business process built. Service orchestration expects a centralized business process coordinating activities over different services combining the outcomes. Fig. 1 depicts a service orchestration.

The choreography does not have a centralized element for service composition. Service choreography describes message exchange, rules of interactions as well as agreements among interacting services. The control logic is not in a single location. Fig. 2. depicts service choreography.

When involving orchestration through a mediation layer we often introduce a *canonical data model*. With such as model various system parties agree or standardize their data models of the business objects that exchange. However, often [20] the entire system ends up with having just one kind of business object. For instance, there is a single Person, Order, Entry, Invoice, etc., with matching attributes and associations, since everyone agrees on them. It is easy to introduce such model with orchestration however clearly later changes to the model are very difficult since all parties have to agree on them and the individual system has limits on evolution.

Alternative approach arising from Domain-Driven development [19] is called *Bounded context*. The idea behind it is that each service aims to operate with particular business objects in a specific context, and thus it may make sense for some service to consider certain attributes, while not for another. For instance, we may consider a User's address to process shipments, however, for price calculations, it is sufficient to only have User level. Thus a large model is divided onto small contexts allowing to model business objects differently based on particular needs. Not all services have the same needs and thus should have the independence to design their needs.

3. SERVICE-ORIENTED ARCHITECTURE

The main reason for a software architect to use SOA or µServices is to modularize a system into services. SOA, however, requires big upfront commitments, since the entire company IT must distribute into separate services. It is fairly easy to introduce new SOA service, one can take a legacy application and define a new network accessible interface for it. A more advanced design divides an application onto multiple services opening for broader service reuse and service orchestration. The challenging task when introducing SOA is the setup of centralized governance, the component responsible integration of services and their communication. Usually, the solution for the integration is Enterprise Service Bus (ESB) that forms the backbone for SOA. As mentioned earlier it enables orchestration, moreover, services can interact through messages or events where the trigger is unknown

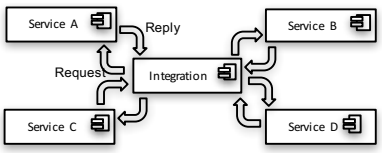

Figure 1: Service orchestration.

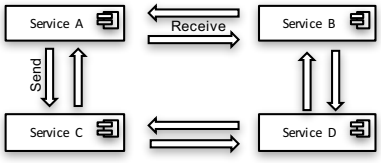

Figure 2: Service choreography.

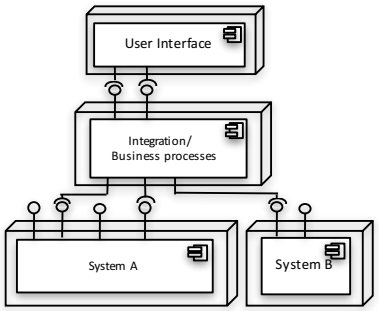

Figure 3: Sample SOA deployment of Systems A&B.

while multiple services react on the particular event. In addition, business processes may be defined at the integration level allowing flexible reconfiguration. However, this solution inclines toward the introduction of and canonical data model. The key point here is that the integration platform is smart, however, also complex. SOA is sometimes referred as "simple services and smart pipes" [12] for the above reason. On top of the system is usually a separate component of the user interface, such as a portal or a dedicated web system. Fig. 3 shows a sample SOA deployment with two systems A and B exposing services for the integration platform above them. The user interfaces part then communicates with the services through the integration component.

The main advantages of SOA comes when enough services are available. The business processes implement service orchestration with control over the company processes. It becomes easy to compose services, introduce alternative ways to deal with processes or build new functionality on the top. The services can be even open to third-parties. However, the layout is usually that various system parts (such as A and B in Fig. 3) are maintained by different teams. Separate teams usually exist for the integration component or user interface. When changing a particular service that introduces an interface modification, the update most likely promotes to the integration level, as well as to the user interface demanding redeploy of multiple components. In such a manner, SOA deployment happens as a monolith involving multiple services; one defected service may prevent the entire application deployment with a complex rollback.

The situation becomes worse since processes span across services dedicated to different teams, exacerbating communication overhead, requiring coherence in the development and deploy. On top of this, companies tend to have a single centralized administration unit managing all changes in the SOA service infrastructure which leads to conservatism and limitation on individual service evolution.

One of the main issues in SOA is system versioning since we do not know the service users. There are even cases when a company maintains over 22 different versions of the same service with a slightly modified interface to accept different data [7]. This significantly impacts the operations involvement demanding monitoring and maintenance.

According to Red Hat[1], SOA community considers the transition to µServices because the common SOA practice tights services to complex protocols stacks, such as SOAP, a protocol for web service communication, and WSDL, to describe a service [11]. While this is not an SOA requirement in practical usage it degrades to solely SOAP and web services not considering alternatives.

To summarize, SOA makes it easy to change business processes, although, changing a service requires deploy of the component providing the service. This may cascade to the whole SOA monolith, not mentioning the need to reflect the changes in user interface. The integration platform is usually very complex when it comes to the first deployment and since it is the centralizing particle it can become the system bottleneck that has to deal with communication overhead or distributed transactions. The integration unit is usually an ESB, that serves the purpose of integration, orchestration, routing, event processing, correlation and business activity monitoring. From the communication perspective, SOA is about orchestrating large services.

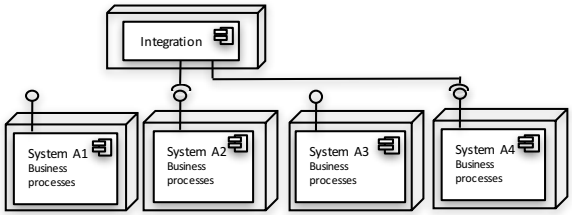

Figure 4: µService architecture for System A from the example in Fig. 3

4. MICROSERVICE ARCHITECTURE

µServices base on three Unix ideas [20]:

- A program should fulfill only one task, and do it well.
- Programs should be able to work together.
- Besides, the programs should use a universal interface.

These ideas lead to a reusable component design, supporting modularization. The major point is that services are brought to production independently of each other, which is one of the main differences with most SOA solutions. It does not only impact deployment, but also evolution and modification efforts. µService followers often cite Conway's law [9], stating that "Organizations which design systems are constrained to produce designs which are copies of the communication structures of these organizations."

The direction µServices head out, is towards lightweight virtual machines. They are implemented as containers (e.g. a Docker) or individual processes. This unbinds dependency on a certain technology, enabling usage of a service-specific infrastructure. µServices usually do share the same database schema as it would predetermine a bottleneck as well as coupling. Each µService is in charge of its own data model, which possibly leads to replication. In the Background section, we mentioned Bounded context, which is the direction for µServices, later in this section we elaborate more details on Bounded context.

Unlike SOA, µServices do not have integration component responsible for service orchestration and prefer choreography. Business processes are embedded in services and there is no logic in the integration. Thus µServices themselves are responsible for interaction with others. This gives limited flexibility to design or adjust business processes over the company IT's. It is a payoff for µService independent service management and deploys. However, Netflix considers even the option to orchestrate[2] µServices, which is not a mainstream path.

It can be noted that µServices emphasize isolation in a way that a particular process and user interaction it's the scope

[1]Paper co-author expertise

[2]https://github.com/Netflix/conductor

of a particular service. A service is usually managed by a single team, however, changes to µServices require user interface propagation. For this reason, both the user interface and µService should be under control of a single team. This gives the team flexibility to manage changes while avoiding bureaucratic negotiation on interface changes. Having µServices independent regarding development and deployment bringing individual scalability and continuous delivery. They provide resilience to failure [3] since a request may be balanced among several service instances.

In [2] authors discuss *DevOps* practice, which goes hand-in-hand with µServices. DevOps is a set of practices aiming to decrease the time between changing the system and deploying the change to production while maintaining software quality. A technique that enables these goals is a DevOps practice [4]. The most notable DevOps practice, which we mention is continuous delivery enabling on-demand automated deployments supporting system elasticity to request load. Similarly, continuous monitoring provides early feedback to detect operational anomalies.

The difference from SOA can be seen from Fig. 4 that shows the System A earlier mentioned in Fig. 3. Each µService is an individually deployable unit maintained by a separate (or the same) team. Note in the figure that no complex integration technology exists over the enterprise, the integration part can be the user interface part or services can interact directly.

Compare to SOA, the user interfaces part may be integrated into the µService, which avoids communication overhead. The communication among µServices does not require REST or messaging, the user interface integration may talk to other services and involve data replication instead, however both REST and messaging are commonly used.

µService should be comprehensible by individual developers and not developed by multiple teams [12]. At the same time, it cannot become a nanoservice since it would demand high network communication, which is expensive compared to local communication. Transactions spanning multiple µServices become complex. For this reason, the design should target transaction spanning a single µService or involve messaging queue. However, recently a no-ACID transaction type becomes discussed in this context, known as compensation transactions[3]. Similarly, regarding data, a µService should be enough large to ensure data consistency.

For µServices, it is a critical decision when it comes to fragment the data model, we mentioned this when introducing the Bounded context. Single service cannot capture the whole context, but there must have a certain boundary. There are strategies [12] how two systems interact, determining the boundary.

1. The *shared kernel* strategy suggests that each domain model shares common elements, but in the specialization areas, they differ.
2. *Customer/supplier* suggests that the subsystem provides a domain model that is determined by the caller.
3. In *conformist* the caller uses the same model as provided by the subsystem and reuses its knowledge.
4. The *anti-corruption layer* provides a translation mechanism to keep two systems decoupled. This is often used for legacy systems.

[3]http://jbossts.blogspot.cz/2016/10/achieving-consistency-in-microservices.html

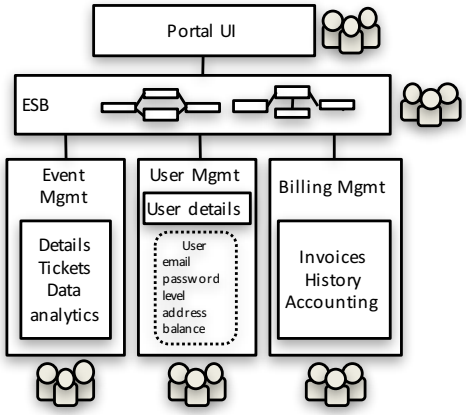

Figure 5: Example system in SOA with highlighted user data format

5. *Separate ways* point no integration among systems.
6. *Open host service* expects a system to provide special services for everybody's use to simplify integration
7. *Published language* has unchangeable linguistic elements (contracts, events, etc.) visible to the outside with a meaning in multiple subdomains.

From the date model perspective, the above strategies (4-6) provide a lot of independence, while (1-3,7) tight the domain models together. From the communication efforts perspective among teams, the (5) requires least efforts followed by (3), (4), (6), (7), (2) and (1) with most efforts.

4.1 Self-Contained Systems

The last architectural variation we mention is a *Self-Contained System* (SCS) [1]. In this approach, a particular system breaks into multiple SCS components that consist of two parts, a user interface and separately deployable µServices sharing the same code-base. Various SCSs communicate asynchronously if necessary. Each functionality is ideally under a particular SCS component. To draw a comparison an e-commerce shop system that would contain 100 µServices, or would only consist of 5-25 SCSs. SCS suggests that a µService has around 100 lines of code. For instance, in practice, in Java EE the project setup would have a multi-module maven project sharing code between the UI and µServices where each part is separately deployable WAR file. SCS can be seen as a specialization of µServices. The approach is promoted by various authors [20].

5. ENLIGHTENING EXAMPLE

To clearly understand the differences, we examine an example application designed in SOA, µServices, and SCS. The aim is to emphasize the characteristics of each particular approach. The example system is an event ticketing system assisting users with ticket selection and purchase. For the purpose of demonstration, it consists of various components. We highlight three: Event Management (EMgm), User Mgmt (UMgm) and Billing Mgmt (BMgm).

In SOA design we consider the canonical data model, which is enforced by contract agreement on the integration level implemented by ESB, which handles business processes and orchestrates management components. The User Interface (UI) part is a portal also handling user authentication

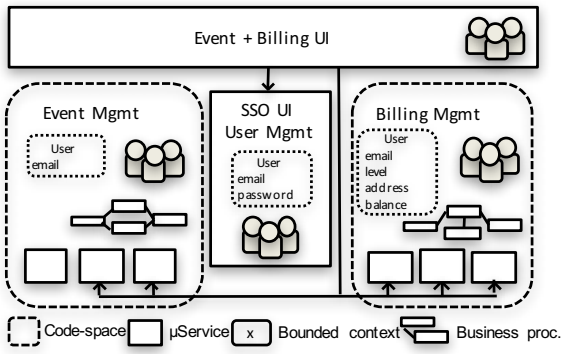

Figure 6: Example system in µServices, showing user data format divided

across the system. Fig. 5 shows possible design decomposition in SOA. Each system is a separate application exposing web services; however deployable as a monolith. Service orchestration goes through the ESB. The portal sits on top of the ESB. Each management component has a separate code space, however, all component agree on data model in order to design processes in the ESB. We highlight *user* data model with multiple attributes managed by UMgm, later in µServices it splits. A change to particular management component must be well discussed and promoted to the ESB by maintaining teams; the change most likely promotes to the portal as well impacting another team. However, the dependencies may impact services in different components. For instance, user data model change may impact other management components and their internals.

µServices approach no longer considers the centralized integration via ESB, instead, it delegates business processes to services. Services may share the same code-space avoiding repetition denoted by the *dashed box*, while still allowing extraction of various deployables - separate µServices. The original UMgm component partially dissolves into the Single Sign-On module (SSO) for authentication, which is implemented by the portal in SOA. Moreover, notice the bounded contexts of *user* data model in each code space and the SSO. Fig. 6 shows the example system. Note the different teams responsible particular components. Moreover, note the location of business processes definition and shared code space. The UI is maintained by a separate team responsible for the service integration. Moreover, a service from EMgm could call a service from BMgm. Changes impacting a particular µServices can be deployed individually, and the data model extension can be service specific. However, these may impact the UI and another team, which is an argument for SCS that we consider next as variant of µServices.

The SCS design is a specialization of µServices, where the same team maintaining a particular service is now responsible for the UI. However, it is not a separate UI application above the services, but a separately deployable application with direct code access, reusing the knowledge. This has multiple advantages, the team is familiar with the knowledge, it fully controls the changes and change-propagation and there is less overhead since no web services need to be used for given UI scope. However, the team deals with the whole development stack including UI, middleware or database development. SCSs should ideally not communicate with each other, while this is fine for microservices. Moreover, SCSs should favor integration at the UI layer. Fig. 7 shows the transition from µServices into SCS. It highlights the integration in the Billing UI involving the SSO and selected EMgm µServices. The distinct UIs for Billing and Events must correlate with the look and feel in order to confirm unity of a single system. When a small change is needed in one SCS, the UI and a particular µServices are redeployed, independent of other µServices in or out of the code base or other SCSs.

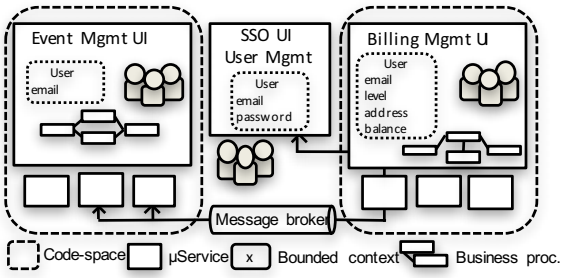

Figure 7: Example system in SCS, showing user data format divided

For a deeper understanding consider a possible component interaction in a use case when user purchases selected tickets. We consider that user finds tickets through the UI, and next aims to process the purchase of his/her selection.

First, let's consider SOA processing in our example in Fig. 5. It would look similar to this:

1. The user sends a request via the portal to buy tickets
2. It propagates to ESB that triggers the business process orchestrated by the ESB
3. EMgm checks whether the tickets are still available, and it locks the selected tickets for 30 minutes
4. BMgm makes sure that user paid previous orders
5. UMgm loads user address and level for possible discount
6. BMgm resolves the discounted price and processes payment
7. EMgm removes tickets locks and makes them not available
8. BMgm issues an invoice for user address
9. UMgm recalculates user-level based on recent purchase

We can see that ESB has rather complex responsibility to orchestrate a lot of services. However, it is easy to reroute the process to different order or consider alternatives on top of existing services.

Next, we consider the interaction in µServices to highlight the differences. The aim is to give the responsibility of a particular process handling to a given µService - in the example we call it payService under the BMgm. If possible we try to avoid distributed transactions, however services may act in a choreography. When considering Fig. 6 the interaction may look like:

1. User selects tickets using the UI involving EMgm µServices and starts the purchase posting the selection the payService
2. The payService triggers the business process
3. It contacts EMgm µService to make sure the tickets are still available, locking them for 30 minutes in EMgm
4. The payService loads current user from bounded context and checks pre-existing payment dues
5. It uses user-level to determine possible discount
6. Next, payService processes the purchase payment
7. payService contacts EMgm µService to remove the ticket lock, updating ticket availability
8. It issues an invoice for an address from bounded context
9. Finally, the payService updates user-level in bounded ctx.

From above we see a dependency of the payService to the locking/unlocking EMgm µService. When considering

SCS in Fig. 7 the system does not dependencies and the interaction is decoupled, e.g. through a message broker:

1. User selects tickets in EMgm UI and starts the purchase by locking the tickets for 30 minutes through EMgm UI; next, it routes and posts the user selection to the BMgm UI
2. The request triggers the business process in the BMgm UI
3. BMgm UI checks user's pre-existing due payments
4. To determine possible discount BMgm UI loads user-level from the bounded context
5. Next, BMgm UI processes the purchase payment
6. To remove the lock and set availability on selected tickets in EMgm, the UI BMgm contacts its µService, which emits an event to a message broker, to which an EMgm µService reacts - updating ticket availability and locks
7. BMgm UI issues invoice for an address from bounded ctx.
8. Finally, it updates user-level in bounded context

We can see that the interaction is delegated to a particular SCS, while it emits events to get outside of the SCS. Changes to the process involve SCS modification and new deployment, however, there is higher autonomy in performing such a change, possibly involving a single team.

In a second use case we consider data analytics. It aims to send an advertisement email to past customers offering a discount to events matching user history and details. We start with SOA and next consider µService approach:

1. EMgm is the initiator of the action over ESB
2. It polls an aggregate user list with details from UMgm
3. In batch for each user, it issues a business process via ESB
4. It fetches the last user event from BMgm
5. For no history it skips the user; for existing history, it determines the event category in the EMgm
6. In EMgm it finds a matching event by category, the earliest date nearby user address
7. BMgm determines the price basing on user-level
8. EMgm sends the advertisement content via user email

In µServices a particular service performs the process:

1. EMgm µService is the initiator
2. It uses BMgm to fetch user filtered list with all details from bounded context and the last attended event
3. It initiates a batch business process for each user
4. It determines the event category, based on users last event
5. Next, it finds the first matching event by category, earliest date, and location near to user address
6. It determines the ticket price using BMgm µService considering the particualr user.
7. Finally, it sends the advertisement through EMgm µService

Similarly, even in this case, we see SOA's flexibility to reorder or extend the process using existing services on the ESB level. In SCS we would again need to involve decoupling, however. it may introduce a redundancy in user history in both BMgm and EMgm to simplify the processing, which is, unfortunately, a common approach for preserving autonomy.

When a third-party service appears providing the distance from user location to the event venue, ESB has a wide pallet of communication adapters to contact it, while most likely introducing a new service as a facade. In µServices with protocol independence, we may contact the service directly, however only if our language provides an implementation of the protocol, which may lead again to the introduction of a new µService. It seems the best solution is to combine µServices with an integration framework providing a collection of adapters.

6. COMPARISON

µServices and SOA both divide the system onto services, however in different ways. SOA can still be seen as a monolith from the deployment perspective [20], while µServices lead towards independent deploys. Industry relates µServices to container technologies simplifying automated deployment [12]. Containers can be given the credit for building such self-contained µService deployment units. Moreover, µService push towards design autonomy with plenty of small teams resulting in heterogeneity of components, which some may criticize.

SOA has a wide enterprise scope, while the intention of µServices is to do "one thing well" [15]. SOA gains it flexibility from centralized management and µServices inclines for distribution. µServices fit well to the context of cloud computing. Researchers refer to µServices as to cloud-native, while SOA is rarely referenced that way in the literature [12]. The key features here is the individual and automated service deployment. In general, service-based approaches are vital for cloud-native approaches [10].

Considering industry demands and recent research directions [12] µServices seems to be the future direction, while SOA becomes the legacy. However, there are counterexamples. In [14] author argues that SOA and µServices are not the same as nothing in µServices build on SOA and multiple pieces are missing for µServices. [18] reminds the fundamental concepts:

> µServices architecture is a share-as-little-as-possible architecture pattern that places a heavy emphasis on the concept of a bounded context, whereas SOA is a share-as-much-as-possible architecture pattern that places heavy emphasis on abstraction and business functionality reuse.

SOA is a better fit to a large, complex, enterprise-wide infrastructure than µServices [18]. SOA suits well to the situation with many shared components across the enterprise. µServices do not usually have messaging middleware and fit better to smaller, well-partitioned web based applications. Moreover, SOA better enables services and service consumers to evolve separately, while still maintaining a contract. µServices fail to support contract decoupling, which is a primary capability of SOA. Finally, SOA is still better when it comes to integrating heterogeneous systems and services. µServices reduce the choices for service integration.

Table 1 provides a summary comparison between SOA and µServices, which we described in this paper.

7. RESEARCH CHALLENGES IN SERVICE INTEGRATION

Both architectures come with drawbacks and features that are complex or cause difficulties. These are challenges to address in research. Clearly, SOA has the issue with monolithic deploy, centralization, bound data model leading into canonical data model or complex protocol stack. On the other hand, it is flexible with business process changes, giving a centralized view on system processes. µServices bring higher autonomy to services reducing data structure dependencies, relocating business processes to particular services. This together with the connection of virtual boxes brings the benefits of individual service deploy enabling elastic service scalability.

Table 1: Comparing µServices and SOA

Concern	µServices	SOA
Deploy	Individual service deploy	Monolithic deploy, all at once
Teams	µServices managed by individual teams	Services, integration and user interface managed by individual teams
User interface	Part of µService	Portal for all the services
Architecture scope	One project	The whole company/enterprise
Flexibility	Fast independent service deploy	Business process adjustments on top of services
Integration mechanism	Simple and primitive integration	Smart and complex integration mechanism
Integration technology	Heterogeneous if any	Homogeneous/Single vendor
Cloud-native	Yes	No
Management/governance	Distributed	Centralized
Data storage	Per Unit	Shared
Scalability	Horizontally better scalable. Elastic	Limited compared to µServices. Bottleneck in the integration unit or a message parsing overhead. Limited elasticity.
Unit	Autonomous, un-coupled, own container, independently scalable	Shared Database, units linked to serve business processes. Loosely coupled.
Mainstream communication	Choreography[1]	Orchestration
Fit	Medium-sized infrastructure	Large infrastructure
Service size	Fine-grained, small	Fine or coarse-grained
Versioning	Should be part of architecture, more open to changes	Maintaining multiple same services of different version
Administration level	Anarchy	Centralized
Business rules location	Particular service	Integration component

Multiple issues can be found in service composition involving both SOA and µServices. In [13] authors consider such issues. They, for instance notice, cross-cutting concerns that repeat across services, such as exceptions, transactions, security or service level agreements. One of the main issues they point out is knowledge reuse. We may want to reuse a component, a data transformation rule, a process fragment or templates. To some extent, SCS does it, however in a limited scope. One possible reuse approach is to describe the certain rules in machine readable or queryable format, e.g. it is easy to make a query to the database to find expected data constraints. The most common approach is to manually evaluate the knowledge and copy/paste it to a secondary location. However, this only exacerbates the complexity of evolution management, since once the knowledge changes, it has multiple locations to maintain. Next, there exist recommender systems [13] to facilitate the composition process. For instance, they perform on-the-fly similarity search over a knowledge base of reusable patterns.

Paths addressing above issues in service integration use automation. A synthesis [13] reuses knowledge of integrated services. For instance, symbolic model checking may generate an executable business process for the integrating component [16]. Artificial intelligence can be applied involving Semantic service with machine-readable descriptions of service properties and capabilities with reasoning mechanisms to select and aggregate services [17]. Naturally, the problem with this approach is the extensive development effort to provide and maintain the semantic information in correlation to the system internal knowledge. Finally, involving Model-Driven Development approach on service design allows to transform the knowledge across various services. However, because of the high-level of abstraction used in the approach demanding complex generalization and design of transformation rules, the approach is rarely used. Developers aim to focus on specific problem description rather than its abstraction.

Specifically for µServices, the issue can raise from the service-specific data model or business processes hidden from others, which facilitates the service autonomy. On the other hand, this leads to replication of knowledge among services with service integration. Moreover µServices sacrifice the centralized view on business processes, or generally the knowledge, that is now distributed and hidden across services. For example, consider two communicating services A and B. These services exchange information while both maintain information in distinct data formats, conforming business rules or processes, or even security restrictions. Once the service A changes its internal knowledge it must still correlate to service B. While the correlation applies only to some extent, it could cause system failure if not properly maintained. Let's consider that service B could base its reasoning on a service A internal knowledge in computation. This could leverage the maintenance efforts in service composition. A great example where this happens often is the user interface.

When low-level data format changes, the user interface forms, tables, and reports must reflect such change. However, this is very difficult to preserve [6] since components are maintained by distinct teams and limited type safety exist in user interfaces. In [6] author suggest to utilize metaprogramming and aspect-oriented programming, to let the source service to stream meta-information to the integration component. In particular case, web browser JavaScript library weaves the together provided information and templates at runtime to assemble the user interface fragments reflecting the actual information from the service. Such approach minimizes maintenance efforts on the user interface part since any change in the service is immediately reflected in the user interface fragment. The approach complies well with [8] contemporary client-based frameworks such as Angular2 or React. Moreover, having the meta-information exposed allows rendering the user interface fragments on various platforms using native components, while all adapting

to the changes in services or even context.

While [6] provides an approach for minimizing change impacts of structural data manipulation across integrating services, there is still a dependency on business rules and security. In [5] authors suggest that services capture their constraints, business rules and security untangled from the service source code, e.g. in domain specific language description or involving annotation descriptors. This not only improves readability of the above but also allows their automated inspection and transformation into a machine-readable format that can be shared across services. [5] shows a system with its internal constraints and business rules captured in Java EE and Drools being automatically inspected and transformed into a JavaScript description applied across user interface enforcing the constraints and rules already at the client-side.

There are multiple open challenges left to address, beyond the scope of this paper. We highlight the main areas:

1. How should be changes and evolution communicated across different teams working in distinct services?
2. How to detect/test incompatibilities in service integration?
3. How to determine the scope/boundary of a particular µService?
4. How to effectively mitigate failures in service integration?
5. Distributed transactions across services, e.g. compensating transactions, and no-ACID transactions.
6. Cross-cutting issues with code replication across services.
7. Security across services must correlate when one service allows half of the process the second can deny it.
8. How to migrate existing systems onto µServices?
9. How to deal with replicated info. in service specific databases?

8. CONCLUSION

This paper aims to demystify ambiguous usage of term and meanings of SOA and µServices. It specifies characteristics and differences of both architectures pointing out their strengths, weaknesses, and differences. While both architectures address system integration, the industry seems to move toward µServices, leaving SOA as the legacy term. The main credit for this tendency can be given to the ability of independent service deploy and elastic scalability.

For instance, in the market of integrated systems, Gartner[4] predicts a boom where hyperconverged systems reach 24% of the overall market by 2019. Moreover, Gartner states that in this segment recently started new era bringing the continuous application and µServices delivery.

While µServices seems the be the winner of the two, there are still multiple challenges that come with the architecture and leaves developers with restatements and complex processing. These can be addressed in future research, while we show multiple of these issues are actively addressed by researchers. Readers may wonder about the symbiosis with Internet of Things, Cloud computing - Platform as a Service model, and systems for handling Big Data, all these can be addressed using µService architecture, however these are beyond the scope of this paper and left for future work.

In the end, we raise a philosophical question. Since the evolution path went from heterogeneous system integration, through SOA into µServices, can we expect in near future another step back towards SOA?

Acknowledgments

This research was supported by the ACM-ICPC/Cisco Grant 0374340.

[4] http://www.gartner.com/newsroom/id/3308017

9. REFERENCES

[1] Self-contained systems, 2017. http://scs-architecture.org.

[2] A. Balalaie, A. Heydarnoori, and P. Jamshidi. Microservices architecture enables devops: Migration to a cloud-native architecture. *IEEE Software*, 33(3):42–52, May 2016.

[3] A. Balalaie, A. Heydarnoori, and P. Jamshidi. *Migrating to Cloud-Native Architectures Using Microservices: An Experience Report*, pages 201–215. Springer International Publishing, Cham, 2016.

[4] L. Bass, I. Weber, and L. Zhu. *DevOps: A Software Architect's Perspective*. Addison-Wesley Professional, 1st edition, 2015.

[5] Cemus, Karel and Klimes, Filip and Kratochvil, Ondrej and Cerny, Tomas. Separation of concerns for distributed cross-platform context-aware user interfaces. *Cluster Computing*, pages 1–8, 2017.

[6] Cerny, Tomas and Donahoo, Michael J. On Separation of Platform-independent Particles in User Interfaces. *Cluster Computing*, 18(3):1215–1228, sep 2015.

[7] Cerny, Tomas and Donahoo, Michael J. *Survey on Concern Separation in Service Integration*, pages 518–531. Springer Berlin Heidelberg, 2016.

[8] Cerny, Tomas and Donahoo, Michael Jeff. On Energy Impact of Web User Interface Approaches. *Cluster Computing*, 19(4):1853–1863, dec 2016.

[9] M. E. Conway. How do committees invent. *Datamation*, 14(4):28–31, 1968.

[10] M. Fowler. Microservices resource guide, 2016. http://martinfowler.com/microservices.

[11] N. Josuttis. *Soa in Practice*. O'Reilly, 2007.

[12] N. Kratzke and P.-C. Quint. Understanding cloud-native applications after 10 years of cloud computing - a systematic mapping study. *Journal of Systems and Software*, 126:1 – 16, 2017.

[13] A. L. Lemos, F. Daniel, and B. Benatallah. Web service composition: A survey of techniques and tools. *ACM Comput. Surv.*, 48(3):33:1–33:41, Dec. 2015.

[14] J. McKendrick. 3 situations where SOA may be preferable to microservices, 2017. http://www.zdnet.com/article/3-situations-where-soa-may-be-preferable-to-microservices.

[15] S. Newman. Building microservices : designing fine-grained systems, 2015.

[16] M. Pistore, P. Traverso, P. Bertoli, and A. Marconi. Automated synthesis of composite bpel4ws web services. In *IEEE International Conference on Web Services*, ICWS '05, pages 293–301, Washington, DC, USA, 2005. IEEE Computer Society.

[17] J. Rao, P. Küngas, and M. Matskin. Composition of semantic web services using linear logic theorem proving. *Inf. Syst.*, 31(4-5):340–360, June 2006.

[18] M. Richards. *Microservices Vs. Service-oriented Architecture*. O'Reilly, 2015.

[19] V. Vernon. *Implementing Domain-Driven Design*. Addison-Wesley Professional, 1st edition, 2013.

[20] E. Wolff. *Microservices: Flexible Software Architectures*. CreateSpace Independent Publishing Platform, 2016.

[21] Z. Xiao, I. Wijegunaratne, and X. Qiang. Reflections on soa and microservices. pages 60–67, 2016.