# Case study on data communication in microservice architecture

**Conference Paper** · September 2019

DOI: 10.1145/3338840.3355659



**3 authors:**

Antonin Smid

**1** PUBLICATION **2** CITATIONS

SEE PROFILE

Ruolin Wang

**1** PUBLICATION **2** CITATIONS

SEE PROFILE

Tom Černý

Baylor University

**127** PUBLICATIONS **964** CITATIONS

SEE PROFILE

# Case Study on Data Communication in Microservice Architecture

Antonin Smid
Baylor University
Waco, TX , Texas
Smid_Antonin1@baylor.edu

Ruolin Wang
Baylor University
Waco, TX , Texas
Ruolin_Wang1@baylor.edu

Tomas Cerny
Baylor University
Waco, TX , Texas
Tomas_Cerny@baylor.edu

## ABSTRACT

Microservice Architecture is becoming a design standard for modern cloud-based software systems. However, data communication management remains a challenge. This is especially apparent when migrating from an existing monolithic system into microservices. In this paper, we report on data synchronization and improvement of the data-source performance. We faced these challenges in production-level development. Two case studies illustrate and describe our approach. To address data synchronization we propose using an automated data streaming system between databases. To improve the performance of a data-source we introduced a solution with the distributed cache. We discuss the balance between the performance and coupling and point out situations where our architectures are appropriate.

## CCS CONCEPTS

• **Information systems** → **Enterprise applications**; *Data exchange*; • **Applied computing** → *Enterprise applications*; *Service-oriented architectures*; *Enterprise data management*;

## KEYWORDS

Microservices, Cloud-computing, System Integration

**ACM Reference format:**
Antonin Smid, Ruolin Wang, and Tomas Cerny. 2019. Case Study on Data Communication in Microservice Architecture. In *Proceedings of International Conference on Research in Adaptive and Convergent Systems, Chongqing, China, September 24–27, 2019 (RACS '19),* 6 pages.
https://doi.org/10.1145/3338840.3355659

## 1 INTRODUCTION

Microservices [7, 12] are the latest trend in software design, development, and delivery. A number of benefits are often associated with microservices, including faster delivery, improved scalability, and greater autonomy. A greater autonomy also provides features such as smaller code bases, strong component isolation, and organization around business capabilities. These benefits promise improved maintainability over traditional monoliths.


*RACS '19, September 24–27, 2019, Chongqing, China*

ACM ISBN 978-1-4503-6843-8/19/09…$15.00
https://doi.org/10.1145/3338840.3355659

In this context, it is not surprising that demand has grown for migrating legacy monolith applications to microservices. The research in this area which provides design patterns and guidelines on how to implement the migration is substantial. However, most of these studies are from the macro architecture perspective, and they target issues such as identifying candidates for microservices on the monolithic system or separating these candidates into a hybrid architecture [12]. In our work, we have discovered a number of implementation challenges with microservices which are rarely mentioned, if at all, in existing research. These issues have a significant impact on the performance of microservices.

One of the main challenges is how to manage data communication from the original monolith to the new microservices, and between the distinct microservices themselves. A good design for data communication of microservices will reduce overhead for system communications and improve data transmission performance..

In this paper, we introduce two architectures to improve data communication performance of microservices by demonstrating them in two case studies from our production-level systems. One is about data synchronization between the legacy monolithic system's database and the miroservices' databases. The architecture proposed here uses message queue and streaming platforms such as Kafka [3] and Debezium [8] to automatically capture and synchronize database changes. The other case study shows how to improve data communication performance between microservice instances by applying the cache and message broker Redis [21]. For both cases, we also compare our solutions with traditional approaches, analyze the benefits and shortcomings of our solutions, and present a discussion about it.

The remaining content is organized as follows. Section 2 reviews related work. Section 3 provides background. Section 4 provides a detailed analysis, architecture, and a description of the two case studies. The final section concludes the paper.

## 2 RELATED WORK

The popularity of microservice architecture (MSA) has grown consistently over the past five years [5]. As an example of this just look at the Google search trend for microservices during the last five years 1. During this time, many businesses migrated their systems from monolith or Service Oriented Architecture (SOA) into MSA [7]. MSA has become core architecture concept for many big tech companies [16, 20, 23]. However, microservices aren't silver bullets, and the difficult process of migration has drawn the attention of the industry as well as academia.

Taibi et al. [27] conducted a study interviewing experienced developers and examined the motivations, issues, and benefits behind migration to MSA. The study concluded that the critical drivers


**Figure 1: Google search trends for microservices**

for migration are the overall maintainability and scalability. However, the main technical issues were monolith decoupling, database migration, and data splitting.

Another empirical study with MSA practitioners [9] describes the incremental migration. The researchers decompose the process into three parts: reverse engineering - gaining knowledge about the legacy system; architecture transformation - domain decomposition and applying domain-driven design practices; and forward engineering - the actual implementation of the new system. They recommend by adding new functionalities written as microservices and then incrementally migrate the existing ones.

Knoche and Hasselbring [19] also highlight maintainability as the key reason supporting modernization. They argue that monolith systems are difficult to extend since any change requires extensive testing and rework. And that limiting the number of entry points and establishing platform-independent interfaces allows the future evolution of the system to become more feasible. They describe modernization from COBOL to Java and, for complex high-value systems, they suggest to first define the service facades, then implement them in the legacy technology, and to finally re-implement them again as Java microservices.

In 2017, Balalaie et al. [5] published a catalog of migration and rearchitecting patterns derived from observation of several industrial projects. They provided general guidelines with a concrete technology stack for implementing them.

One of the challenges with migration is the identification of the microservice candidates on the monolithic system. Levcovitz et al. [22] proposed a technique based on mapping the database tables on business areas and facades, which creates a dependency graph that can be used to identify the subsystems. In 2018, Zhongshan et al. [25] went even further with a division approach which analyzes both the application data model and the data flow.

Since the migration to MSA varies from project to project, many authors published case studies concerning certain business domains. Belalie et al. [4] reported their experience with migrating to a cloud-native environment. They put emphasis on the continuous delivery and importance of service contracts. The implementation of a service can evolve, however service contract should remain the same across all implementation versions.

Gouioux and Tamzalit [15] published a case study of migrating large-scale system to MSA. They also highlighted the importance of a good continuous delivery pipeline for its significant reduction in deployment costs. This allows higher optimal microservice granularity. In regard to MSA integration, the authors argued in favor of lightweight passive choreography over orchestration solutions like the Enterprise Service Bus [7], which can be too heavy for MSA. For the former, they report higher reuse of components as well as significantly decreased response time.

In 2018, Mazzara et al. [10] presented an extensive case study on migrating a bank system. Many of their motivations were common: the system had too many functionalities, the coupling between components was too high, it was hard to understand, and the deployment was complex due to extensive testing. They have migrated to MSA running at Docker Swarm and introduced choreography based on the messaging system RabbitMQ [18].

Furda et al. [14] also see microservice migration as a promising technique of modernizing monoliths and elaborate on three challenges: multitenancy, stateful and data consistency.

However, all these works lack when it comes to the data communication perspective and its optimization, which is addressed in this work.

## 3 BACKGROUND

Monolithic architecture produce large systems that are deployed as atomic units, which makes them hard to evolve or update. When one component in such a system was modified, it usually meant extensive testing and redeployment of the entire structure. Also, scaling a single component meant scaling the whole application.

Microservice architecture aims to solve the challenges of monolithic systems. The main empathise is on systems' modularity. Software built with MSA is composed of multiple component services. So each of them can be tweaked, updated and deployed separately without compromising the integrity of the application. Therefore, the developers can update the system and redeploy just a single module instead of the whole app. From the business perspective, this also means that, instead of having different teams handling the back-end, front-end, operations and quality assurance, each small team owns a microservice. In other words, the team not only creates it, but also takes responsibility for deployment and maintenance.

Microservices are usually deployed in service containers like Docker [17]. An infrastructure like this requires an orchestration system to provide the necessary features such as automated deployment, scaling, service discovery, load balancing or externalized configuration. There are open-source container orchestration solutions such as Kubernetes [13] or Docker Swarm Mode that can be used.

Each microservice defines an interface that other components can consume, and the services communicate via RESTful APIs or through a message broker. The message routing is simple. There is no centralized element integrating the services; the governance as well as the data management is distributed. This interaction style is called dumb pipes and smart endpoints.

Since each service uses a different data-store, there is no need to share the data model across the whole app (canonical data model). Instead, each service operates on a subset of the data model in a so-called *bounded context* [12]. Since each service specializes in a different business case, naturally not all services need to operate with all entities. A service may even consider only certain attributes of some object and ignore others. For example, in our system, a person management service uses all the information about users, including the degree they pursue. However, our hotel service does not need the degree attributes at all.

The challenge here could be how to maintain consistent data states across various microservices that have distinct databases.

Another challenge that one could run into is how to share newly-produced real-time data with a large number of clients. These two distinct scenarios are quite common and so they are what we will be focusing on as our case studies in the next section. The case study demonstrates an applied solution in deployed production-level systems.

## 4 CASE STUDY

The International Collegiate Programming Contest (ICPC) [1] is a multi-tier team-based programming competition. The contest involves a global network of universities hosting competitions that advances teams to the ICPC World Finals. Participation has grown to tens of thousands of the finest students and faculty in computing disciplines at almost 3,000 universities from over 100 countries. The scale and distribution of the ICPC system creates a difficult management problem.

The ICPC Contest Management System (CMS) system is a web application designed to simplify organization of the contest and coordinate distributed execution. The two case studies are selected from our project to migrate our old monolithic CMS system to a modern microservices architecture.

### 4.1 Scenario 1: Data Synchronization

The motivation behind our first case study was increasing performance of a monolith system by applying MSA with separate databases. We were looking for a solution that is able to synchronize data between different databases efficiently.

For this study, we wanted to synchronize the hotel management system and the ICPC CMS system. In the old version of the ICPC system, both management systems are developed and deployed together as a monolith application. This makes development, deployment, and maintenance difficult and complex. Thus, we decided to separate the hotel management system from the monolith application as a microservice based on the functional distinction. Accordingly, a database for the hotel microservice distinct from the monolith database was introduced. This brought the following benefits:

- Loosely coupled services; each microservice is self-contained, which makes the microservice an atomic unit of failure instead of the entire application.
- Reduced load on individual database.
- Each service can use the type of database that is best suited to its needs. For example, a service that does text searches could use ElasticSearch [11]. A service that manipulates a social graph could use Neo4j database [24].

However, introducing a microservice also brings challenges. The most significant one is how to implement these features that need data in the database of a separate module.

One way to solve the issue is to fetch data from other services using a REST API. This is a commonly used approach for inter-service communication and could help to some degree. However, there are still many problems preventing this from being a good solution. First, relying on an API for communication creates a huge amount of complex data communication between services for various use cases. For example, in our case we manage hotel room information related to person information which is managed by the CMS.

For example, there are 100 rooms in the hotel system. Now, we need to display a table to show room information as well as the room holder's information. If we have the data at one place, we just need one query to get all the data we need. However, by using the Rest API, we need to create 100 calls by the room holder's identifier to fetch their information from CMS or use a complex API which contains all the room holder's identifiers together as a URL body and which will get a massive response with their information. Both methods require a lot of data transportation which will reduce the performance of the service, increase the design difficulty and introduce inefficiency in the communication.

A better way to solve the problem is to replicate necessary data from among services so that they are located in the database of the interacting modules, which means each service has its local replication of the data. This approach normally preserves one writer and multiple readers. What's more, by identifying bounded context [12], we could choose the necessary attributes for the shared elements of the business model. This allows us to implement functions by queries inside a particular microservice and reduce the cost of data communication as much as possible with an acceptable storage payoff. However, this strategy works based on a good implementation for data synchronization. We addressed this by using a streaming platform Kafka and Debezium. This way, all changes in the master database get promoted to all other databases that replicate the particular data fragments.

**Context:** There is a working monolith and a newly-introduced microservice running alongside. The microservice has its own database and it replicates some fragments of the monolith's database.

**Problem:** When the data change in the master database, how to synchronize the data replicas in the microservices' databases.

**Solutions:**

- (Not Suggested) Develop a scheduled job to extract all of the records from the monolithic system database, then remove and insert those records into the microservices' databases. This is very simple to implement but it does result in periods where the data is out of sync. Also, the service will need to stop for a while to undergo the data update. This could be acceptable for some systems in which data is not updated frequently, but this is far from ideal.
- (Not Suggested) Modify the monolithic system to add logic that updates the records in the microservices' databases when updating the monolith database. This is easy to implement but is prone to problems with coupling and transactions. It needs the monolithic system to have a decent design to handle events such as failures for both databases.
- (Suggested) Use a data streaming platform, e.g. Kafka, Debezium. Since we would like to reduce service coupling as much as possible, we need each service to manipulate its own database. Thus, an ideal way to synchronize the database is to send messages from the monolithic system to the microservice about what changes have made and make the microservice decide what to do for its own database corresponding to the changes. For this purpose, data streaming and message distribution can be used. One can build a message "Topic" between the monolithic system and the microservice. Then it is necessary to build a producer in the monolithic system. For each change in the monolithic system database, the producer

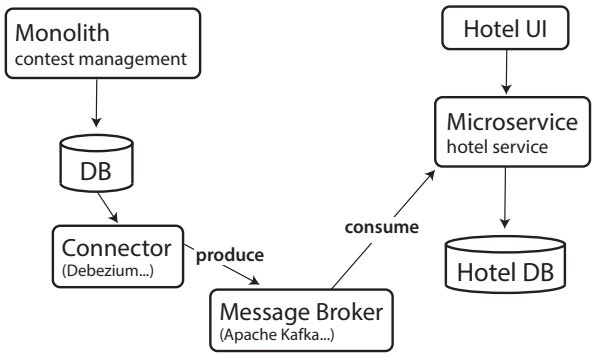


**Figure 2: The architecture diagram for data synchronization by using data streaming platform**

produces a message containing the change information and send it to the Topic. The microservice consumes the message and performs the operations according to the data change. In this way, the monolithic system does not need to care about the manipulations in the microservice database, and the microservice only listens to the Topic; there is not direct communication between the microservice and the monolith systems. This preserves independence for involved modules. Using this approach, we still need to add logic code to produce messages according to database change although this is easy to implement. With the help of streaming provided by the Debezium platform, this approach could be more simplified and efficient. Debezium is an open source platform to help capture the database changes and automatically produce messages. Thus, basically it only requires a configuration for the connection to the involved databases and indicate the bounded context. Then the monolithic system can focus solely on its purpose and there is not interference with the business logic. Using this method, even direct changes to the database by an administrator will be promoted to all replicas.

## 4.2 Scenario 2: Increasing Performance

The motivation behind our second case study was increasing performance of a monolith system by enabling horizontal scaling through microservices. We were looking for a solution that is able to handle high peaks in load, allow higher number of concurrent websocket connections and preserves the legacy data.

The study relates to another ICPC system - MyICPC [2, 6]. This system provides information for attendees of the World Finals event. The application contains a feed of posts harvested from social media, schedule, team information, scoreboard, gallery, and a challenge game called Quest. The legacy version of MyICPC was a monolith Java application with a server-rendered front-end. Although the application was deployed as ten instances in a high-availability cluster [26], the performance was insufficient, leading to two challenges to address:

(1) The overall performance did not grow with a number of cluster instances, and the response time was too high.
(2) During the web traffic peak, the application could not handle a load of concurrent connections, and request over a certain threshold failed.

First, we will look at the overall performance issue. The system was designed as a monolith running at an application server, and used a relational database. By deploying the application in a high-availability cluster [26] (multiple instances of the same server), we were able to increase the availability and reliability, but not the performance. The main reason was that all the instances shared the same database but did not share any query results. This is shown in Fig. 3 (before). Hence, the same query was often executed many times by different server instances, and the database became the system's bottleneck. The second reason for the poor scaling was the high overhead of the application servers running on each separate machine. Since the system used a messaging service, the application server had to run in full enterprise profile (Java EE), which is by itself very heavy with respect to memory and resources. Running multiple instances of the application server turned out to be inefficient.

To enable efficient horizontal scaling, seven microservices were separated from the system and communicate with a Javascript front-end. The original project was well organized in modules (timeline, schedule, scoreboard...), so the decomposition was straightforward. We have replicated the minimal set of common functions into each service and rewritten the dependencies between modules as REST calls. The services are highly configurable and independent. Therefore, only the necessary dependencies are included, and the final production artifacts are quite small (~50MB). Moreover, these microservices may be run without the application server.

Since the majority of the database interactions were read-only, the distributed data structure Redis [21] was introduced. It serves as a cache (see Fig. 3 after). The distributed cache scales very well with the microservices, since it only has one node to write to, but many to read from. When the microservice performs a query on the database, it saves the results into the cache using a JSON format. Then, another microservice, looking for the same data, checks the cache first. In case the data is changed in the database, the cache record is invalidated. So, next time data is read, the cache is empty and the microservice loads new data from the database directly. With this design, much of the work is no longer dependent on the relational database. Since a query is performed only once and until the data changes all the consequent reads rely on the cache only.

Second, we needed to address an issue with real-time connections. MyICPC system should deliver real-time push updates to the users through websockets. However, the previous version did not handle more than a few hundreds concurrent connections, which was not enough. To resolve this, we could exploit the pre-existing microservice structure. We introduced a new microservice handling exclusively real-time connections using Redis as a message broker (shown at Fig. 4). All the instances of this service subscribe to dedicated Redis topic, and react on the messages by pushing them to clients. Therefore, any microservice can send a real-time push to clients just by publishing a message to Redis topic. Since the microservice and Redis both scale independently on the rest of the system, we could easily allocate more resources for for real-time push delivery to prevent crashes during the World Finals.

Lastly, it is worth mentioning why we chose to go with a single database rather than implementing a full microservice solution with a separate database for each service. The main reason is that most of the queries from our services are read-only. With the initiation

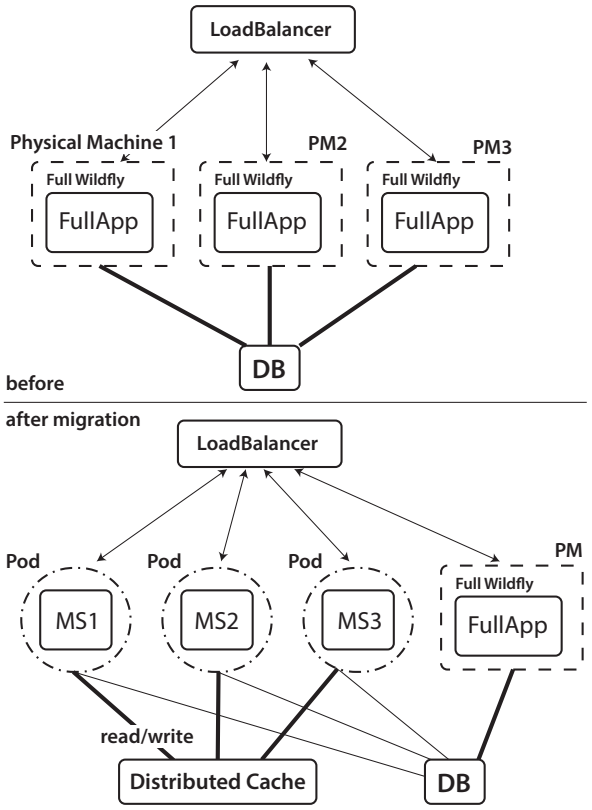


**Figure 3: The original architecture diagram (top) depicts several instances of an application server running the full monolithic application on distinct physical machines using the same database. The new architecture (bottom) consists of containerized lightweight services which use a distributed cache as a primary data source. However, there is still a remaining part of the monolith for services that do not need to scale.**

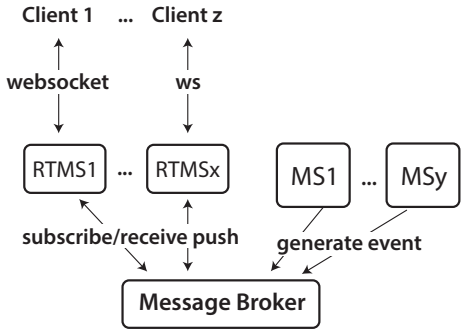


**Figure 4: Scalable real-time data delivery. One microservice is dedicated entirely to websocket communication with clients, receiving events from other services through a message broker.**

of a distributed cache, the database does not need to scale anymore. Keeping a single database simplifies the migration process; all the services can keep the same data model as the legacy system (or it is a subset). The new system is natively compatible with the legacy data, which is a great benefit. Usually, the preexisting data is ignored in the legacy system as reported by Francesco et al. [9](page 34). Moreover, there is no need to address synchronization issues as is mentioned in the previous example.

We have separated parts of the system that needed to scale as microservices. However, we still kept the monolith alive to be used for content administration tasks, which do not need to scale. This technique is called a hybrid migration pattern and was described by K.Finnigan [12].

### 4.3 Discussion

**Data synchronization:** As mentioned in the first case study, when migrating from monolithic system to microservices, applying a separate-database architecture is a good practice which will bring lots of benefits for the whole system. However, efficiently synchronizing data between different databases is a big challenge. We introduce the streaming platform (Kafka) with the data change capture platform (Debezium) as an approach to address this challenge since this approach is simple, efficient and preserves independence for involved modules.

This solution works well in our system, but that doesn't mean this is universally the best solution for all scenarios. This approach still has some limitations. For example, one limitation is that there is overhead for deployment and maintenance for applying the streaming platform. Thus, this approach is not worthwhile in a situation in which data rarely needs to be synchronized. In such case, adding some simple logic to the master system is more acceptable. Another limitation of this approach is that the microservices need to be synchronized under a similar data model with the master system, and extra source code must be introduced. For example, in our case study, the master database has two tables called *person* and *personinfo*. There is a foreign key called *personinfo_id* in the *person* table which connects these two tables. For the microservice, we only need the attribute *name* from *person* and attribute *address* from the *personinfo* table. Ideally, we would like to create only one data model in the microservice called *person_new* which contains *id*, *name*, *address*. However, this is not practical for a system with Kafka and Debezium since Debezium could only fetch the data change from a particular table. Thus, if some *address* is changed, the microservice can only get the message that address is changed in table *personinfo* but it won't know which record in *person_new* should be updated. Thus, we need to add new attributes *personinfo_id* in *person_new* table. As the relationship gets more complex, so will the table since there is no element to match other than a synthetic identifier. For a complex joined table, it would be more flexible and efficient to have a message producer in the master system. Essentially, the design should be flexible and take the whole system cost balance into consideration.

**Performance increase:** The design solution proposed in example 2 is not a pure microservice since the services share a single database. However, the distribution is achieved by introducing a distributed cache, which bring both advantages and drawbacks. The

horizontal scaling is ensured by introducing a distributed cache, so that the database experiences much lower load.

The positive aspects of this architecture are that the solution scales well and does not require splitting the data model into bounded contexts [7]. That makes the migration easier and saves development time on rearchitecting the data model.

The overall complexity of the implemented solution is similar to the legacy monolith, which is a benefit since MSA usually brings replication [7]. Minimal common code is replicated (i.e. database connectors, caching, messaging, basic queries). Furthermore, most of the controllers, business logic and repositories are well-separated.

However, the solution only works in cases when the data does not change frequently, when most services mainly read and writes appear rather sporadically. Sharing the data model is easier for migration, but also represents coupling between the services remaining from the monolith. One service can not change the model nor the cache format independently of the others since it would eliminate the performance improvement. A pure MSA would allow complete independence. However, the synchronization problem discussed in the first scenario would arise and the system would become more complex.

Lastly, we conclude that using the message broker is an efficient way of communication between microservice. The publish/subscribe model is very flexible and provides a faster mechanism than HTTP requests with the benefit of the persistent messages.

## 5 CONCLUSIONS

Migrating from monolith structure to MSA is the latest trend in software design. An efficient design for data communication between the original monolith and new microservices or among distinct microservices is very important for improving the performance of the whole system after migration. In this paper, we proposed two architectures targeting two common scenarios for data communication between microservices. One was to implement data synchronization between different databases of distinct microservices or between microservice and monolith by building a data stream system with technologies such as Kafka and Debezium. By applying this approach, we can preserve independence for involved modules and maintain the master system's focus solely on its business logic without adding extra logic for manipulation of the microservices' databases. In the second case study, we proposed a 'semi microservice' technique for increasing the database performance by introducing a distributed cache instead of splitting the data model into multiple bounded contexts.

Both approaches work well in our project. However, they are far from universally optimal architectures for every scenario. This paper provides an inspiration to apply our approach for situations with similar challenges.

In future work, we will continue to examine microservice architecture in both existing and new systems. First, we plan to further divide the centralized system from our example one. The approach is to follow the path depicted by this paper. Moreover, we will consider extension of example two with serverless functions. We are currently developing a third application that is meant to go to production, which will be a pure microservice design from the beginning. This portfolio of production-level software will give us more context to extend our work.

### ACKNOWLEDGMENT

This material is based upon work supported by the ICPC Foundation.

## REFERENCES

[1] 2019. Contest Management System. (2019). http://icpc.baylor.edu.
[2] 2019. MyICPC: Second Screen Experience Platform. (2019). http://my.icpc.global.
[3] Apache. 2019. Kafka: a distributed streaming platform. (2019). Retrieved Jun 6, 2019 from https://kafka.apache.org/
[4] Armin Balalaie, Abbas Heydarnoori, and Pooyan Jamshidi. 2015. Migrating to cloud-native architectures using microservices: an experience report. In *European Conference on Service-Oriented and Cloud Computing*. Springer, 201–215.
[5] Armin Balalaie, Abbas Heydarnoori, and Pooyan Jamshidi. 2016. Microservices architecture enables devops: Migration to a cloud-native architecture. *IEEE Software* 33, 3 (2016), 42–52.
[6] Tomas Cerny and Michael J. Donahoo. 2018. Second Screen Engagement of Event Spectators. *Adv. Human-Computer Interaction* 2018 (2018), 3845123:1–3845123:20.
[7] Tomas Cerny, Michael J. Donahoo, and Michal Trnka. 2018. Contextual Understanding of Microservice Architecture: Current and Future Directions. *SIGAPP Appl. Comput. Rev.* 17, 4 (Jan. 2018), 29–45. https://doi.org/10.1145/3183628.3183631
[8] Debezium. 2019. Debezium: an open source distributed platform for change data capture. (2019). Retrieved Jun 6, 2019 from https://debezium.io/
[9] Paolo Di Francesco, Patricia Lago, and Ivano Malavolta. 2018. Migrating towards microservice architectures: an industrial survey. In *2018 IEEE International Conference on Software Architecture (ICSA)*. IEEE, 29–2909.
[10] Nicola Dragoni, Schahram Dustdar, Stephan T Larsen, and Manuel Mazzara. 2017. Microservices: Migration of a mission critical system. *arXiv preprint arXiv:1704.04173* (2017).
[11] ElasticSearch. 2019. ElasticSearch: a distributed, RESTful search and analytics engine. (2019). Retrieved Jun 6, 2019 from https://www.elastic.co/products/elasticsearch
[12] Ken Finnigan. 2019. *Enterprise Java Microservices*. Manning Publications.
[13] The Linux Foundation. 2019. Kubernetes, Production-Grade Container Orchestration. (2019). Retrieved Jun 10, 2019 from https://kubernetes.io
[14] Andrei Furda, Colin Fidge, Olaf Zimmermann, Wayne Kelly, and Alistair Barros. 2017. Migrating enterprise legacy source code to microservices: on multitenancy, statefulness, and data consistency. *IEEE Software* 35, 3 (2017), 63–72.
[15] Jean-Philippe Gouigoux and Dalila Tamzalit. 2017. From monolith to microservices: Lessons learned on an industrial migration to a web oriented architecture. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*. IEEE, 62–65.
[16] S. Ihde. 2019. From a Monolith to Microservices1 REST: The Evolution of LinkedIn's ServiceArchitecture. (2019). Retrieved Jun 6, 2019 from https://www.infoq.com/presentations/linkedin-microservices-urn
[17] Docker Inc. 2019. Docker, Enterprise Container Platform. (2019). Retrieved Jun 10, 2019 from https://www.docker.com
[18] Pivotal Software Inc. 2019. Rabitt MQ. (2019). Retrieved Jun 10, 2019 from https://www.rabbitmq.com/
[19] Holger Knoche and Wilhelm Hasselbring. 2018. Using microservices for legacy software modernization. *IEEE Software* 35, 3 (2018), 44–49.
[20] S. Kramer. 2019. The Biggest ThingAmazon Got Right: The Platform. (2019). Retrieved Jun 6, 2019 from https://gigaom.com/2011/10/12/419-the-biggest-thing-amazon-got-right-the-platform/
[21] Redis Labs. 2019. Redis: an open source in-memory data structure store. (2019). Retrieved Jun 6, 2019 from https://redis.io/
[22] Alessandra Levcovitz, Ricardo Terra, and Marco Tulio Valente. 2016. Towards a technique for extracting microservices from monolithic enterprise systems. *arXiv preprint arXiv:1605.03175* (2016).
[23] Tony Mauro. 2019. Nginx - Adopting Microservices at Netflix:Lessons for Architectural Design. (2019). Retrieved Jun 6, 2019 from https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/
[24] neo4j. 2019. neo4j: A Graph Platform Reveals and Persists Connections. (2019). Retrieved Jun 6, 2019 from https://neo4j.com
[25] Zhongshan Ren, Wei Wang, Guoquan Wu, Chushu Gao, Wei Chen, Jun Wei, and Tao Huang. 2018. Migrating Web Applications from Monolithic Structure to Microservices Architecture. In *Proceedings of the Tenth Asia-Pacific Symposium on Internetware*. ACM, 7.
[26] Red Hat Software. 2019. High Availability Guide. (2019). Retrieved Jun 10, 2019 from http://docs.wildfly.org/12/High_Availability_Guide.html
[27] Davide Taibi, Valentina Lenarduzzi, and Claus Pahl. 2017. Processes, motivations, and issues for migrating to microservices architectures: An empirical investigation. *IEEE Cloud Computing* 4, 5 (2017), 22–32.