Design of Scalable and Resilient Applications using Microservice Architecture in PaaS Cloud

David Gesvindr, Jaroslav Davidek and Barbora Buhnova
Lab of Software Architectures and Information Systems,
Faculty of Informatics, Masaryk University, Brno, Czech Republic

Keywords: Cloud Computing, Microservices, Architecture Design.

Abstract: With the increasing adoption of microservice architecture and popularity of Platform as a Service (PaaS) cloud, software architecture design is in many domains leaning towards composition of loosely interconnected services hosted in the PaaS cloud, which in comparison to traditional multitier applications introduces new design challenges that software architects need to face when aiming at high scalability and resilience. In this paper, we study the key design decisions made during microservice architecture design and deployment in PaaS cloud. We identify major challenges of microservice architecture design in the context of the PaaS cloud, and examine the effects of architectural tactics and design patterns in addressing them. We apply selected tactics on a sample e-commerce application, constituting of microservices operated by Azure Service Fabric and utilizing other supportive PaaS cloud services within Microsoft Azure. The impact of the examined design decisions on the throughput, response time and scalability of the analyzed application is evaluated and discussed.

1 INTRODUCTION

Microservice architecture is becoming a dominant architectural style in the service-oriented software industry (Alshuqayran et al., 2016). In contrast to traditional multitier applications where the role of software components is played mainly by software libraries deployed and executed in a single process together with the main application, in microservice architecture, individual components become truly autonomous services (Fowler, 2014). There are multiple advantages of this approach—change in a single component does not require the entire application to be redeployed, communication interfaces become explicit, and components become more decoupled and independent of each other, as illustrated in *Figure 1*.

Separation of services into functions that can interact via interfaces is not new, same as methods to implement such separation in the framework of service-oriented architecture (Sill, 2016). But as emphasized by Sill (Sill, 2016), recent implementations of microservices in cloud settings take service-oriented architecture to new limits. Possibilities of rapid scalability and use of rich PaaS (Platform as a Service) cloud services open new design possibilities for microservices but also bring new threats for software architects, with increasing difficulty to navigate among the enormous number of available design options. This creates the need to examine the impact of applicable design patterns in the context of microservices and PaaS cloud.

Although some guidance on the microservice implementation in the cloud exists, systematic support for software architects interconnecting microservices with other available PaaS cloud services (such as storage and communication services) is not available, leaving them to rely on shared experience with typically a single application scenario, without considering other strategies or alternative designs.

In this paper, we study different architectural decisions which are being considered during microservice architecture design in connection with the PaaS cloud. As a contribution of this work, we elaborate on both

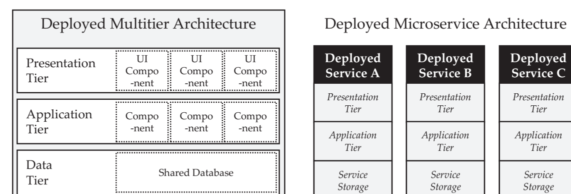

Figure 1: Separation of components and their deployments in a traditional multitier architecture and in microservice architecture.

Gesvindr, D., Davidek, J. and Buhnova, B.
Design of Scalable and Resilient Applications using Microservice Architecture in PaaS Cloud.
DOI: 10.5220/0007842906190630
In *Proceedings of the 14th International Conference on Software Technologies (ICSOFT 2019)*, pages 619-630
ISBN: 978-989-758-379-7
Copyright © 2019 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

documented and undocumented design practices and solutions, and study their effects, including identification of several surprising takeaways. As part of this paper, we have designed and implemented a highly configurable e-commerce application (an e-shop solution), which is designed in a way that its architecture can be easily reconfigured to support thorough evaluation of the impact of various design decisions on multiple performance related quality attributes (throughput, response time and scalability). When designing this application, we paid special attention to the selection of real use cases and realistic architecture design, being overall as close as possible to a production version of such an application. The highly configurable architecture of this application gives us a unique opportunity to provide comparison of multiple versions of the same application using microservice architecture to evaluate and isolate impact of different design decisions, which is rarely seen in existing work. Software architects can benefit from our work while designing their own applications when facing the same design decisions. With the help of this work they shall now be able to make better-informed decisions and choose the right architectural patterns leading to desired quality of the application, or avoid undocumented problems caused by chosen architecture or used PaaS cloud services.

For the purpose of effective benchmark execution, we also implemented an automated client application that can reconfigure the deployed application, initialize sample data seeding on the server based on user requirements, execute any mix of workload, display key performance metrics and export detailed performance counters in JSON format. This application was used for our benchmarks discussed in this paper.

For the implementation of the sample application, we have decided to use Microsoft technologies and cloud services. The application is developed in .NET framework using Azure Service Fabric (Mic, 2018), which is an open-source application platform for simplified management and deployment of microservices that can run in Microsoft Azure cloud, on-premise infrastructure and any other cloud infrastructure. This platform was chosen because of its robustness (Microsoft uses internally this technology to operate large scale services in Microsoft Azure, eg. Azure SQL Database, Cosmos DB and others), and rich platform services, which simplify microservice development. On the other hand, the features that we used can be manually implemented in other frameworks and the same results can be obtained by hosting small web applications communicating with each other via REST APIs, hosted in Docker and orchestrated by Kubernetes. Overall our results are generally valid independently of the platform as the used patterns are platform independent and can be applied also to different cloud provider (Amazon, Google Cloud) offering container hosting services and managed NoSQL databases.

Overall, we have evaluated 105 different benchmark scenarios using 4 cluster configurations (5, 10, 15, 20 nodes cluster) involving 4 different storage services in the PaaS cloud, 2 communication strategies (synchronous and asynchronous) and 11 design patterns.

The paper is structured as follows. After the discussion of related work in *Section 2*, outline of the background in *Section 3*, and outline of key architectural decisions that influenced the separation of services in *Sections 4*, *Sections 5*, *6* and *7* are dedicated to the presentation and evaluation of architectural concerned with service storage, communication between microservices and application resilience. We conclude the paper in *Section 8*.

2 RELATED WORK

When designing microservice architectures, software architects are currently often relying on known design patterns and tactics (Gamma et al., 1995; Fowler, 2002), which are however not validated in the context of microservices or PaaS cloud. Alderado et al. point to an absence of repeatable empirical research on the design and development of microservice applications (Aderaldo et al., 2017). New design guidelines for microservice architectures are emerging (Sill, 2016; Wolff, 2016; Nadareishvili et al., 2016; Newman, 2015), which however do not contain evaluated performance impacts of recommended patterns on realistic implementations. At the same time catalogs of design patterns for the design of PaaS cloud applications are becoming available (Erl et al., 2013; Wilder, 2012; Homer et al., 2014; Mic, 2017), but without measured impacts of their combinations and their use in a context of microservices. Validations of microservice architecture design patterns are published by companies that have deployed microservices (Richardson, 2017; Net, 2015) and want to share their experience with transition to microservice design but not mentioning PaaS cloud deployment. Instead they focus on their currently deployed architecture, its behavior and sometimes related performance characteristics rather than transferable takeaways. Due to the size of their projects, they cannot afford to implement multiple variants of their application using different design patterns and compare performance of those to isolate the impact of used de-

sign patterns. This is where our work complements the current state of the art via offering more guidance for the actual decision making. A case-study evaluating the impact of transition from multi-tiered architecture to microservice architecture on throughput and operation costs is described in (Villamizar et al., 2015), but not in the context of PaaS cloud, as it is deployed to IaaS virtual machines. Challenges related to transaction processing and data consistency across multiple microservices are described in (Mihindukulasooriya et al., 2016; Pardon et al., 2018). Criteria for microservice benchmarks and a list of sample applications are presented in (Aderaldo et al., 2017), but without optimization for PaaS cloud and its services.

3 MICROSERVICES IN PaaS CLOUD

An indisputable advantage of operating microservices in the PaaS cloud is the availability of a rich set of complex ready-to-use services, providing software architects with complex functionality, high service quality (hight scalability and availability guaranteed by SLA), low-effort deployment, and thus easy integration within the developed application. Moreover, multiple services are not even available for on-premise deployment, or are costly to deploy and operate with the same quality of service.

Microservices hosted in the PaaS cloud can benefit very well from cloud elasticity and measured service (Mell and Grance, 2011), which allows us to easily scale individual services by allocating new compute resources and pay only for the time when the service instance is running. As part of low-effort deployment of microservices, we can take advantage of container orchestration as a service, which are services provided by majority of the cloud providers, used to manage and orchestrate applications deployed in form of containers. Very often, it is a preconfigured and fully managed Kubernetes cluster. And for example in Microsoft Azure, the Azure Container Service is not even billed. One needs to pay only for the compute resources used to host the containers itself. To support rapid scalability, we do not need to allocate virtual machines with the Kubernetes cluster to host containerized applications. Instead, we can take advantage of PaaS cloud container-hosting services (e.g. Azure Container Istances), which is a fully managed service providing per-second billing based on the number of created instances, the memory and cores selected for the instances, and the number of seconds those instances are run. Such a rapid elasticity allows us to scale microservices almost instantly with very effective operation costs.

Microsoft Azure used for our implementation provides us also with the possibility to host Azure Service Fabric cluster in form of a fully managed service with very low deployment and maintenance effort. The cluster itself is deployed and operated at no cost, we are only billed for virtual machines used to host our services. New virtual machines can be easily provisioned and released based on the overall utilization of the cluster, to take advantage of cloud elasticity and to optimize operation costs.

Despite all mentioned advantages of microservice deployment to the PaaS cloud, there are associated threats related to missing guidance on how to design microservices in the PaaS cloud context. As there is a very rich set of PaaS cloud services currently available that can be utilized by the microservice application (storage, messaging, etc.) and have a direct impact on the quality of the service, it becomes very complex for a software architect to design the microservice application so that it meets all given quality criteria. In this paper we compare and discuss multiple design choices and their impacts learned from running over 100 experiments with variable software architecture of our microservice application.

4 SERVICE DECOMPOSITION DECISIONS

This section describes the key design decisions that shaped the overall architecture of the designed application and led to separation of the application into a set of interconnected microservices, refined from the initial set of domain entities, identified using the Domain Driven Design principles (Evans, 2003).

4.1 Bounded Context

To split an application with a single data model into a set of microservices, the Bounded Context design principle suggests the division of large data model into a set of smaller models with explicitly defined relationships. Proper application of the bounded context principle is one of the major challenges when designing a microservice architecture, as it becomes very difficult to find the right balance between very small microservices having a single data entity, and services handling multiple entities that tend to ultimately end up being too complex.

The following points characterize the advantages of setting the bounded context small (Fowler, 2014): *explicit service dependencies*, *independent scalability* and *high-availability*.

On the other hand, the problems that arise with the utilization of small bounded contexts are:

- *Data integrity enforcement* – Referential integrity of entities stored at a single microservice can be easily enforced at the storage level, but when referring entities are stored in different microservices, it becomes very complex to guarantee that the referenced entity exists.
- *Cross-service queries* – When the user wants to access data that is distributed among multiple microservices, it is necessary to query all services participating in the query and then combine related data, which is a complex operation and may have negative impact on service response time as shown in *Section 5.6*.
- *Cross-service transactions* – Distributed transactions are generally complex to implement and when transaction modifies data across multiple microservices, it requires the developer to implement additional logic to ensure that transactions on all services are either all committed or all rolled back.
- *Data Duplication* – To overcome issues related to cross-service queries, transactions and integrity enforcement, frequently referenced entities can be stored in multiple copies as part of multiple services for the price of additional consistency management.

Our sample application consists of 7 microservices (6 stateful, 1 stateless) depicted in *Figure 2*. We designed every service to manage and store a single domain entity. There are two exceptions we would like to explain here:

- *Product Service* – manages product catalog, which persists **Product** and **Category** entities. We considered separation of these closely coupled entities into isolated microservices, but because they are referencing each other very often and at the same time Categories are only referenced by Products, we decided to store them in a single microservice.
- *Sales Service* – manages stored **Orders** (headers with embedded items) for a specific user and at the same stores a list of **OrderItems** for a specific product. We decided to store sales data in a duplicate manner due to limits exposed by applied partitioning.

4.2 Partitioning

Partioning is very often associated with the storage layer (Homer et al., 2014) but in the context of microservices, partitioning can be propagated up to the service interface depending on the storage technology used. If the microservices are stateful, the use of the partitioning at the storage level is highly advisable, so that every node hosting the service only stores a specific portion of data based on the partition key. Selection of the partition key must be done very carefully as the key will be then required by most of the service methods to be able to determine which instance can process the request, as depicted in *Figure 3*. At the same time, requesting data across multiple partitions becomes a very complex operation, as illustrated in *Figure 4*, which needs to be minimized by design.

4.3 Summary of Recommendations

Data model of the application must be split into multiple microservices with adequate level of granularity. Because of better consistency enforcement and higher communication efficiency closely coupled entities should be in the same microservice. An important point to emphasize is that microservices should also be separated at the storage level. Storage shared by multiple microservices should be strictly avoided by design, as it hinders independent scalability of microservices. The storage becomes a single point of failure, with high risk of becoming a performance bottleneck. To design highly scalable microservies, partitioning at the storage and compute level should be applied.

4.4 Evaluation

The impacts of our partitioning strategy on scalability of the designed application can be observed in *Figure 9*, which shows throughput of the REST API depending on the size of the compute cluster and confirms that applied partitioning strategy leads to design of a scalable microservice application.

5 STORAGE DESIGN DECISIONS

Selection of a storage technology or storage service in the PaaS cloud has a significant impact on the throughput and scalability of the application (Gesvindr and Buhnova, 2016a). As we expect that even in case of microservice architecture the storage tier of stateful service will significantly influence performance metrics of the service, we decided to evaluate four different storage technologies that can be utilized by our application, to assess how they will limit service scalability and what the overall throughput of the service in different scenarios will be. This

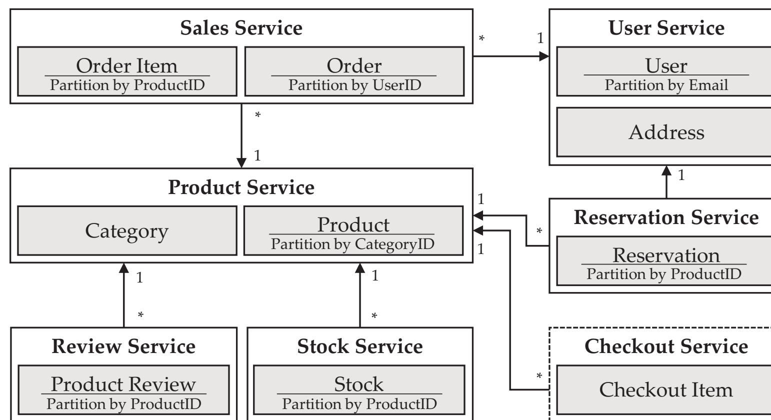

Figure 2: Separation of data model into different microservices.

shall provide software architects with additional guidance for selecting storage technology and design of the storage layer. We evaluate only NoSQL storage services, as it was demonstrated in (Gesvindr and Buhnova, 2016a) that in the PaaS cloud they significantly outperform classical relational databases.

5.1 Storage Abstractions

Individual microservices in our application were implemented without dependency on any storage technology or service, which is still fairly unique while advisable approach, which allowed us to isolate and measure the impact of used storage technology. We designed our own abstraction layer comprised of storage independent repositories and adapters for specific storages. Queries defined to retrieve data from the storage in the application are defined in C# in a form of our own composite predicates, which should be descriptive enough to meet all querying needs of the application. Our storage abstraction layer parses them and generates native queries for given storage technology. Moreover, to define a repository, we only define storage-independent domain entity as a class in C# and all necessary storage structures are automatically created and optimized for a given set of queries, which is most important in case of the key-value storage with limited querying support. This abstraction layer allows us to easily reconfigure what storage technology is used without the need to redeploy the application, making the benchmarking process more effective.

5.2 Key-value NoSQL Database

The first type of storage we decided to evaluate is a key-value NoSQL database as it is provided in a form of PaaS cloud service by every major cloud provider and offers very high throughput, nearly unlimited scalability and low operation costs at the cost of limited querying support which can be partially mitigated by proper data access tier design.

The representative key-value NoSQL database we evaluated is Azure Table Storage which is a fully managed key-value NoSQL database with almost unlimited scalability when data partitioning is properly used, but it comes at the cost of very limited query support, which needs to be taken into account by the architect. Data is stored in tables without any fixed structure (every row can have a different set of columns), where every row must be uniquely identified by a pair of partition key and row key. Data with the same partition key is stored at the same server, therefore it is important to generate the partition key for stored rows in such a way that the rows are distributed across multiple servers, which leads to high scalability of this storage. It is important to be aware of limited querying support as data can be efficiently filtered only based on partition and row key, not by other columns, which are not indexed. The fastest queries are those accessing the data from a single partition with an exact match of a row key. When query is executed to retrieve data across multiple partitions, its response time significantly increases. The service is billed based on the amount of used storage and number of transactions. There are no performance tiers or billing based on performance.

To effectively retrieve data from Azure Table Storage, it is necessary to store the same table in duplicate copies but with different set of partition row keys, as illustrated in *Table 1* for the list of products in a catalog. Data is partitioned by CategoryID, as we are always displaying the list of products in a single category and stored sorted in an ascending order based on a different row key, which supports filtering and sorting based on that column (e.g. in the table: Name, EAN Code and Price).

Table 1: Tables generated to store duplicate data in Azure Table Storage for effective querying.

Table name	PartitionKey	RowKey
Product_CategoryID	CategoryID	ProductID
Product_Name	CategoryID	Name_ProductID
Product_Name_DSC	CategoryID	Name(inverted)_ProductID
Product_EANCode	CategoryID	EANCode
Product_EANCode_DSC	CategoryID	EANCode(inverted)
Product_Price	CategoryID	Price_ProductID
Product_Price_DSC	CategoryID	Price(inverted)_ProductID

We took an experimental approach where for some queries with unpredictable performance, we take advantage of low transaction costs and high scalability of Azure Table Storage and execute multiple variants of the query leading to the same result using different tables. Only the fastest query returns data.

5.3 Document NoSQL Database

The second type of storage service we want to evaluate is a NoSQL document database, which in comparison to key-value storages provides complex querying support by internal indexing of stored documents, which decreases complexity of effective implementation for the developer.

As a representative within MS Azure, we evaluated Azure Cosmos DB, which is a multi-model NoSQL database as a service, built by Microsoft especially for a highly distributed cloud environment. It supports data in multiple different data models—*key-value storage* with a client protocol compatible with Azure Table Storage, *document storage* using either its own protocol or it offers support for MongoDB clients, *columnar families* with support for Apache Cassandra clients, *graph data* with support for Gremlin clients.

Microsoft provides SLA on performance of this service—Azure Cosmos DB guarantees less than 10 ms latencies on reads and less than 15 ms latencies on (indexed) writes at the 99th percentile. Throughput is influenced by the number of reserved Request Units (RU), and based on the amount of storage and reserved RU the service is billed.

Our application implements adapters for both the document storage and also for the key-value storage as we wanted to compare their performance in terms of a single service. Despite the fact, that key-value storage was evaluated using Azure Table Storage, CosmosDB does not share its limitations related to limited query support. Azure Cosmos DB automatically indexes all columns in the table, so that we do not have to store data with high duplicity to be able to query them efficiently. In the document mode, we store collections of complex entities and same as for the key-value storage, all properties are indexed, so we can efficiently load data based on complex queries.

Results of this experiment are even more interesting when we also compare pricing models and related costs of used storage services, as Azure CosmosDB requires us to reserve performance in form of costly Request Units (RU) and every operation costs certain amount of RU based on operation complexity known prior its execution and when all RU are consumed by queries, new operations are rejected with an error. So the client does not utilize the full potential of the service's hardware, but is logically limited so that performance SLA can be guaranteed. Azure Table Storage exposes very cheap transaction fee and provides maximum performance of the service without any need to prepay certain performance tier but without any performance guarantees.

5.4 Reliable In-memory Storage at Compute Nodes

The last type of storage we wanted to compare in our sample application is a distributed storage using reliable in-memory collections, which are locally present at the compute nodes of the cluster.

Stateful services in Azure Service Fabric have their own unique storage called Azure Service Fabric Reliable Collections, so such a hosted microservice does not have to use external storage services. The main goal is to collocate compute resources and the storage on the same node in the cluster to minimize communication latency. The storage itself is provided in form of Reliable Collections, which is an evolution of classical .NET framework collections, but it provides the developer with a persistent storage with a multi-node high availability achieved through replication (one active replica, multiple passive replicas) and high scalability achieved via data partitioning. There are three types of supported collections: *Reliable Dictionary*, *Reliable Queue* and *Reliable Concurrent Queue*.

To distribute data and workload across the Service Fabric cluster, it is advisable that every stateful

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

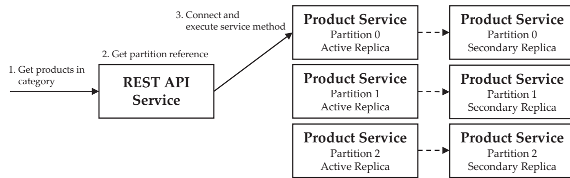

Figure 3: Accessing partitioned data stored in Azure Service Fabric stateful services.

service implements data partitioning and stores data based on a partition key in separate partitions. When the service is requested to load the data, Service Fabric provides very simple API, which based on the partition key opens connection to an instance of the service that stores the partition and executes our code of the microservice that loads data from the local storage of the service partition as depicted in *Figure 3*. A problem arises when we need to access data across multiple partitions. Then the request is sent to a random service partition, it loads local data and over network requests data from other service partitions as depicted in *Figure 4*. This leads to delays in responses. Efficient partitioning is a key to the design of highly scalable and efficient storage using Reliable Collections. Partitioning keys we applied on our data model are depicted in *Figure 2*.

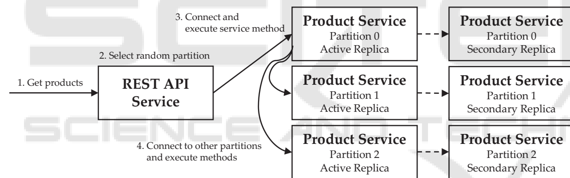

Figure 4: Separation of data model into different microservices.

5.5 Summary of Recommendations

Our experiments confirm strong dependency of application's throughput on used storage technology. None of the used storage technologies outperformed others in all scenarios, therefore selection of storage technology should be accompanied with benchmarks of implemented microservices to determine if selected technology meets required performance criteria. Use of in-memory storage collocated to compute resources (Reliable Collections) leads to great scalability and lowest operation costs. Due to high communication overhead observable in complex scenarios, we would recommend the use of microservice architecture for scenarios where microservices do not have to communicate frequently with each other.

5.6 Evaluation

We have evaluated the use of the four different storage technologies with our microservice application. The results of benchmark evaluating throughput of microservices hosted on 5-node cluster (Azure Service Fabric 6.0.232, node hosted on Azure Virtual Machines D11_V2 2 cores, 14GB RAM, 100GB SSD cache) depending on the used storage technology are depicted in *Figure 5*. We evaluated five different workloads because we expected that various storage services will have different performance characteristics depending on the type of the workload. The most surprising and important result is that there is indeed no single storage service that would outperform all the others in all scenarios. Another fact is that throughput of complex operations is very low due to high overhead of cross-service communication, which is one of the disadvantages of microservice architecture.

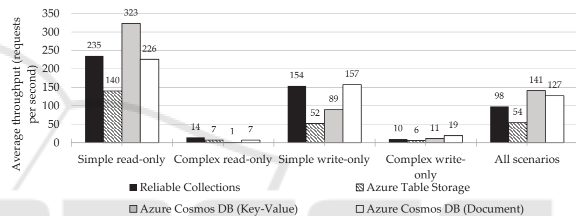

Figure 5: Throughput of REST API hosted on 5-node cluster with different storage services using synchronous communication for different scenarios.

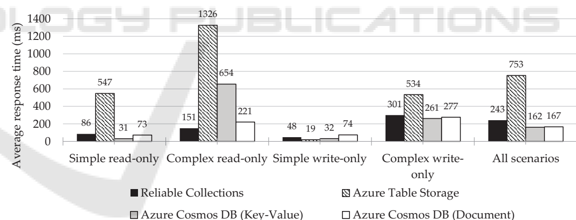

Figure 6: Response time of REST API hosted on 5-node cluster with different storage services using synchronous communication for different scenarios.

To assess scalability of the evaluated storage services, we deployed the same application on 20-node cluster. The results of the benchmarks are depicted in *Figure 7*. The benchmarks confirm that our service is scalable and its scalability is significantly influenced by the used storage service as the throughput increase does not have the same ratio for all storage services and scenarios. We also learned that the use of Azure Cosmos DB is despite its high throughput very tricky, because one needs to pay for reserved storage performance (Request Units) and it is challenging to adjust performance of individual collections to achieve best service performance without wasting allocated request units.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

One can see from our experiments that a single service instance of the stateless Public API Service, which resends client requests to individual services, was not overloaded even for 20-node cluster, which may be a sign of the used ASP.NET Core framework efficiency.

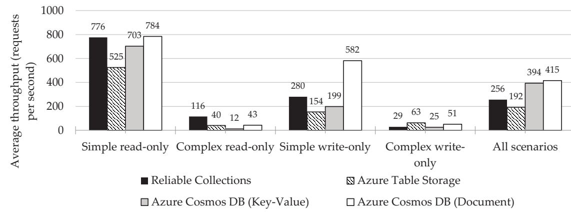

Figure 7: Throughput of REST API hosted 20-node cluster with different storage services using synchronous communication for different scenarios.

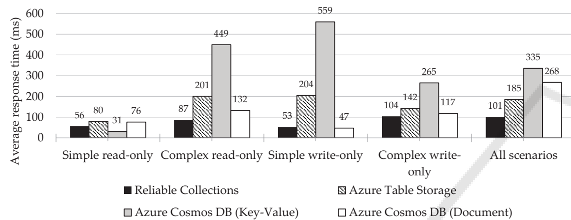

Figure 8: Response time of REST API hosted on 20-node cluster with different storage services using synchronous communication for different scenarios.

We were further interested in the performance of Reliable Collections because they are hosted as an integral part of Azure Service Fabric without any additional costs. From the benchmarks we can confirm that performance of Reliable Collections is strongly dependent on the size of the cluster as the workload is evenly distributed on multiple nodes using partitioning, as depicted in *Figure 9*. On the 20-node cluster (*Figure 7*), read operations are outperforming even very expensive Azure Cosmos DB (cost comparison per request is depicted in *Figure 10*), especially in complex scenarios. Unfortunately, the write operations are slower due to complex data replication among nodes to provide data redundancy.

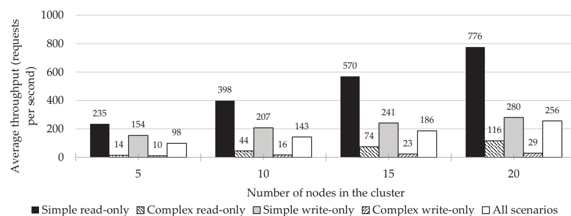

Figure 9: Throughput of REST API hosted on 5, 10, 15 and 20 node cluster with reliable collections storage using synchronous communication for different scenarios.

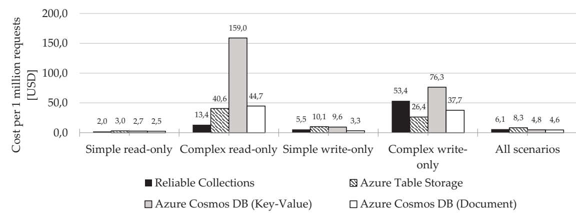

Figure 10: Cost per 1 million REST API requests hosted on 20 node cluster with different storage services using synchronous communication for different scenarios.

6 COMMUNICATION STRATEGY DESIGN DECISIONS

When handling communication between individual services, there are two major strategies that can be applied when microservices communicate with each other:

- *Synchronous* – The service requests a response from another microservice and waits for the response before it continues in an execution.
- *Asynchronous* – the service send a request (message) to another service using messaging service and does not actively wait for the response. When the response is generated, it is delivered through the messaging service back to the original service.

Impacts of asynchronous messaging on multiple quality attributes of the application are well described in (Taylor et al., 2009) and specifically for the PaaS cloud environment in (Homer et al., 2014; Gesvindr and Buhnova, 2016a). Due to different complexity and frequency of communication between microservices, we find it to be desirable to validate impacts of both communication strategies on throughput, response time and scalability of the application. To this end we implemented our sample application in such a way that the communication strategy can be easily switched thanks to adequate abstractions in its architecture.

6.1 Synchronous Communication Strategy

The client application communicates with REST API service, which redirects requests received via publicly available REST API to individual microservices running in the Azure Service Fabric cluster. To discover and communicate with services running in the cluster, the following steps need to happen:

1. *The location of the service needs to be resolved* – The service instance can be migrated between different nodes of the cluster and can be running

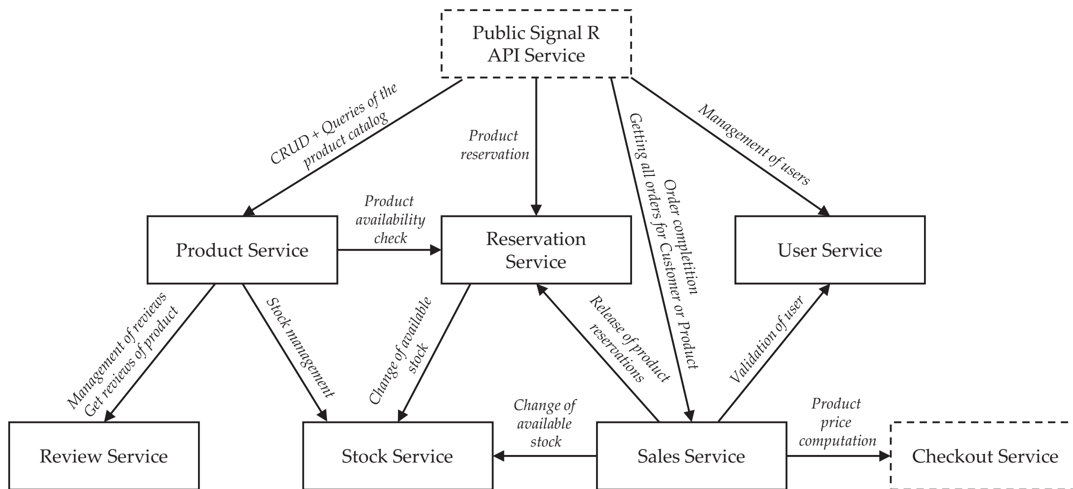

Figure 11: Diagram of service interactions.

in multiple instances. If the service is stateless, requests can be evenly distributed. If the service is stateful, then the service partition should receive only requests for which it does have locally stored data.

2. *Connection to the service* – When the location of the service is resolved by reverse proxy, which is a service running in Azure Service Fabric cluster, a direct connection can be opened between two services hosted in the cluster and requested operation can be executed. When the connection fails, there is a retry logic implemented as part of Azure Service Fabric infrastructure.

Interactions between individual microservices and types of requests are depicted in *Figure 11*.

6.2 Asynchronous Communication Strategy

An alternative communication strategy does not open direct connections between microservices hosted in the cluster, but the service sends a request (message) to another service using messaging service. When the message is delivered, the request is processed and the response is sent back to the messaging service and is delivered to the service waiting for the response. The advantage of this approach is a looser coupling of the microservices as they do not rely on a defined communication interface but only on the format of request and response messages. The disadvantage is the higher implementation complexity, thus higher implementation costs.

Messaging service is not part of Azure Service Fabric, therefore we use the ServiceFabric.PubSubActors library, which hosts a new microservice that works as a reliable messaging stateful service internally using Reliable Queue as the storage for messages. Other services then can subscribe to receive messages of specified type, which are unfortunately broadcasted across all service partitions if it has more than one.

6.3 Summary of recommendations

Based on the results of the experiments, it is advisable to use synchronous communication as a primary communication pattern due to significantly lower overhead, higher throughput and faster response time. Asynchronous communication is desirable for long running operations or operations that need to be completed reliably (e.g. corrective actions described in Section 7.1).

6.4 Evaluation

We implemented both communication strategies in our sample application, which provided us with an opportunity to evaluate and compare the behavior of both communication strategies, which is rarely seen in experience reports form industry on the same application, as having both implementations is costly and not suitable for large production applications.

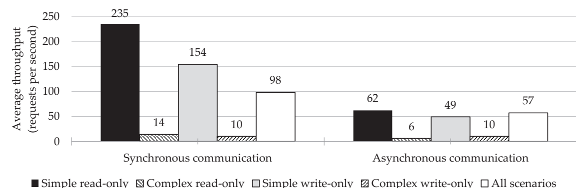

Figure 12: Throughput of REST API hosted on 5-node cluster using synchronous and asynchronous communication with reliable collections storage for different scenarios.

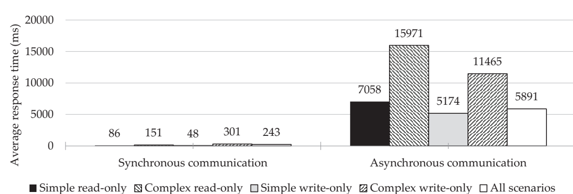

Figure 13: Response time of REST API hosted on 5-node cluster using synchronous and asynchronous communication with reliable collections storage for different scenarios.

The results of the benchmark are depicted in *Figure 13*. Despite the fact that synchronous calls are considered harmful (Fowler, 2014) it is very surprising how significantly synchronous service communication outperforms its asynchronous alternative, which uses messaging as a form of reliable communication between services. The use of reliable messaging services leads to increased availability as very short outages of individual services are not propagated to the client but based on our implementation and tests, this communication strategy for simple scenarios has four times worse throughput than synchronous calls. In case of the complex scenarios, the difference is significantly lower, by which one can conclude that asynchronous messaging is worth considering for long lasting complex operations where high throughput is not required and reliability is more important. Similar conclusions for PaaS cloud applications (not in context of microservices) are also mentioned in (Gesvindr and Buhnova, 2016a).

7 RESILIENCE DESIGN DECISIONS

When designing a microservice architecture, one of the biggest challenges is to enforce data consistency across multiple services, and to manage cross-service transactions. Software architects are nowadays provided with hardly any guidance on addressing these design challenges in microservice architecture, we therefore came with our own implementation, which is a modified version of the *compensation transaction pattern* (Homer et al., 2014) combined with an *event sourcing pattern* (Homer et al., 2014) to handle reliable cross-service compensations. Our implementation does not increase complexity of successfully executed operations, i.e. there were no performance impacts found during benchmarks.

7.1 Cross-service Transactions

To deal with the mentioned issues, we implemented a component called *Event Sequence Source*. This component is used to describe a complex transaction across multiple services, but the transaction itself is split into multiple atomic blocks—the internal block executes operations inside the service, the external block wraps communication with another service. As depicted in *Figure 14*, in the atomic block there are actions to be executed and also pair actions to revert changes if the transaction fails on any of the blocks. This sequence of operations is executed exactly in the order of their definition. We discovered that data validation should be executed as the initial action because when this fails, no corrective actions are needed. As no data integrity is enforced by the storage across services, we run all necessary validations as part of the transaction. When any of the operations fails, an execution of corrective actions is initiated. Corrective actions are implemented as always-succeed actions, which means that they are stored in a highly reliable queue to ensure that the action is repeatedly executed until it succeeds. With this approach, the corrective action overcomes even temporary service failures.

Our implementation of cross-service transactions does not ensure as high level of consistency and atomicity as known from relational databases but it provides us with structured transaction description and sufficient guarantees for the purpose of our business transactions with minimum performance impact.

7.2 Constraints Enforcement

Since the data is stored across multiple microservices with isolated storages, it is not possible to enforce referential integrity at the storage level. Therefore this needs to be enforced with transactions. Every transaction as described in the previous section runs its own data validation, during which it validates the existence of referenced entities in different services.

Another issue we have addressed was how to generate unique identifiers of stored entities, because in many current applications the identifier of a record (primary key) is generated by a relational database, which cannot be followed in our project, and not all storage services offer support for generation of a unique identifier of a record. Therefore, we generate globally unique identifier using service code when new entity is created in sour service code before it is persisted. The uniqueness of the value is then enforced by the storage.

Data validation operations are implemented as a part of the service code, mostly in a constructor of an entity to prevent invalid entity from being instantiated.

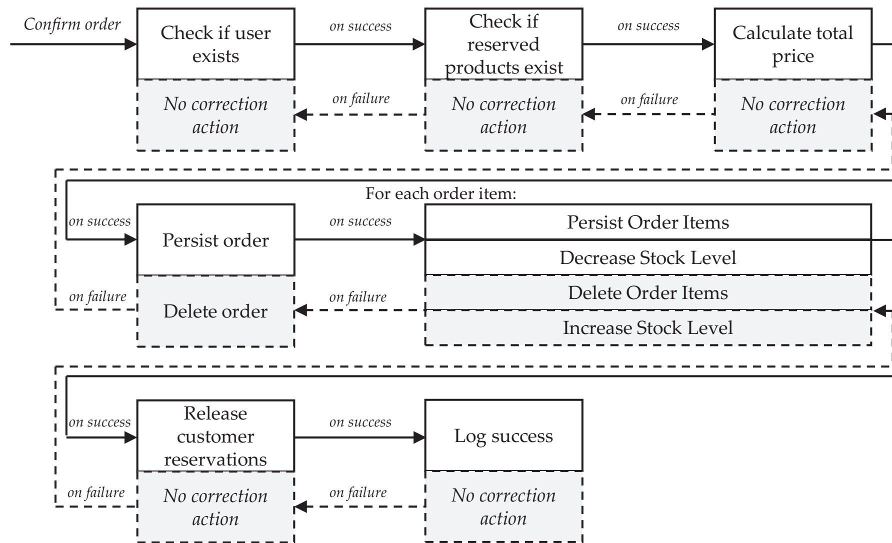

Figure 14: Transaction workflow using Event Sequence Source.

7.3 Handling Transient Errors

It is very important in the PaaS cloud to properly handle transient faults (Gesvindr and Buhnova, 2016b) by implementing a retry strategy so that when a cloud resource is not currently available, an error is not propagated to the client, but instead the operation is retried multiple times with an increasing delay. This is already implemented in majority of client libraries and it just needs to be enabled.

7.4 Recoverability

To increase recoverability of the application in the PaaS cloud, it is advisable to implement the Circuit Breaker Pattern (Homer et al., 2014). This applies also to microservice architecture as the pattern prevents an application repeatedly trying to execute an operation that is likely to fail without wasting resources. It detects if the fault has been resolved and then it gradually increases the load as more and more requests are permitted to execute.

7.5 Summary of Recommendations

Cloud computing services frequently deal with very short outages in duration of few seconds, therefore adequate transient error handling policies in a form of retry strategy needs to be implemented. Especially for microservices, these outages could lead to costly rollback of cross-service transactions. Validity of data must be enforced mostly at the application level, as due to the use of isolated storage services constraints enforcement cannot be applied at the storage level.

7.6 Evaluation

Based on our observations, none of the presented resiliency design decision has a measurable impact on application performance for successful requests, as no additional actions need to be executed. Thus the evaluation of these strategies with respect to the performance metrics studied in this paper is not relevant. However it might be interesting to study their effects on resilience-motivated quality attributes, which are out of scope of this paper.

8 CONCLUSION

In this paper, we have identified, discussed and evaluated a set of design principles that influence service decomposition, storage, communication strategy and resilience in microservice architecture deployed in PaaS cloud. On the sample application, we measured their impact and presented numerous findings, which support the observation that microservice architecture leads to high scalability, but brings new design challenges further amplified by operation in the PaaS cloud and richness of design choices that the architects have. Decomposition of the services needs to be carefully validated, selection of the storage provider cannot be done without knowledge of a specific workload and benchmarks, synchronous communication strategy was found to perform way better despite recommendations in literature. Additional effort shall be invested in extension of the studied design principles and patterns for microservice design in the context of the PaaS cloud.

ACKNOWLEDGEMENT

This research was supported by ERDF "CyberSecurity, CyberCrime and Critical Information Infrastructures Center of Excellence" (No. CZ.02.1.01/0.0/0.0/16_019/0000822).

REFERENCES

(2015). Why you can't talk about microservices without mentioning netflix. *https://smartbear.com/blog/develop/why-you-cant-talk-about-microservices-without-ment/*.

(2017). Cloud design patterns. *https://docs.microsoft.com/en-us/azure/architecture/patterns/*.

(2018). Microsoft service fabric. *https://github.com/microsoft/service-fabric*.

Aderaldo, C. M., Mendonça, N. C., Pahl, C., and Jamshidi, P. (2017). Benchmark requirements for microservices architecture research. In *Proceedings of the 1st International Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering*, ECASE '17, pages 8–13, Piscataway, NJ, USA. IEEE Press.

Alshuqayran, N., Ali, N., and Evans, R. (2016). A systematic mapping study in microservice architecture. In *2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA)*, pages 44–51.

Erl, T., Puttini, R., and Mahmood, Z. (2013). *Cloud Computing: Concepts, Technology & Architecture*. Prentice Hall Press, Upper Saddle River, NJ, USA, 1st edition.

Evans, E. (2003). *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley.

Fowler, M. (2002). *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Fowler, M. (2014). Microservices a definition of this new architectural term.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Gesvindr, D. and Buhnova, B. (2016a). Architectural tactics for the design of efficient PaaS cloud applications. In *2016 13th Working IEEE/IFIP Conference on Software Architecture (WICSA)*.

Gesvindr, D. and Buhnova, B. (2016b). Performance challenges, current bad practices, and hints in PaaS cloud application design. *SIGMETRICS Perform. Eval. Rev.*, 43(4).

Homer, A., Sharp, J., Brader, L., Narumoto, M., and Swanson, T. (2014). *Cloud Design Patterns: Prescriptive Architecture Guidance for Cloud Applications*. Microsoft patterns & practices.

Mell, P. and Grance, T. (2011). The NIST definition of cloud computing.

Mihindukulasooriya, N., García-Castro, R., Esteban-Gutiérrez, M., and Gómez-Pérez, A. (2016). A survey of restful transaction models: One model does not fit all. *J. Web Eng.*, 15(1-2):130–169.

Nadareishvili, I., Mitra, R., McLarty, M., and Amundsen, M. (2016). *Microservice Architecture: aligning principles, practices and culture*. O'Reilly Media, Inc., 1st edition.

Newman, S. (2015). *Building Microservices*. O'Reilly Media, Inc., 1st edition.

Pardon, G., Pautasso, C., and Zimmermann, O. (2018). Consistent disaster recovery for microservices: the BAC theorem. *IEEE Cloud Computing*, 5(1):49–59.

Richardson, C. (2017). Who is using microservices?

Sill, A. (2016). The design and architecture of microservices. *IEEE Cloud Computing*, 3(5):76–80.

Taylor, R. N., Medvidovic, N., and Dashofy, E. M. (2009). *Software architecture: foundations, theory, and practice*. Wiley Publishing.

Villamizar, M., Garcs, O., Castro, H., Verano, M., Salamanca, L., Casallas, R., and Gil, S. (2015). Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. In *2015 10th Computing Colombian Conference (10CCC)*, pages 583–590.

Wilder, B. (2012). *Cloud Architecture Patterns*. O'Reilly Media, 1st edition.

Wolff, E. (2016). *Microservices: Flexible Software Architectures*. CreateSpace Independent Publishing Platform.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS