2016 13th Working IEEE/IFIP Conference on Software Architecture

Architectural Tactics for the Design of Efficient PaaS Cloud Applications

David Gesvindr
Lab of Software Architectures and Inf. Systems
Masaryk University, Faculty of Informatics
Brno, Czech Republic
Email: gesvindr@mail.muni.cz

Barbora Buhnova
Lab of Software Architectures and Inf. Systems
Masaryk University, Faculty of Informatics
Brno, Czech Republic
Email: buhnova@mail.muni.cz

***Abstract*—Platform as a Service (PaaS) is one of cloud service models associated with effective system operation in the presence of a rich set of platform services. Due to the complexity of the PaaS service model, software architects need to face various challenges when designing cloud applications that have to meet certain quality guarantees.**

In this paper we report on our experience with the design and implementation of highly scalable PaaS cloud applications, which resulted into deep understanding of the weaknesses and benefits of this service model, and helped us to identify and evaluate a number of architectural tactics for PaaS cloud applications. Besides the identification and evaluation of the tactics, this paper studies the mutual relationships and the impact of the tactics on various aspects of application's quality. The findings are demonstrated on a real-world case study of a complex private social network system, based on the Microsoft Azure cloud.

I. Introduction

Cloud computing has became a popular approach to software application operation, utilizing on-demand network access to a pool of shared computing resources, and associated with many benefits including low-effort provisioning, rapid elasticity, maintenance cost reduction and pay-as-you-go billing model [1]. However, application deployment in the cloud is not itself a guarantee of high performance, availability, and other quality attributes, which may come as a surprise to many software engineers who detract from the importance of proper architecture design of a cloud application, although the architecture design plays a key role in the quality of the application.

An especially challenging context for software architects is the *Platform as a Service (PaaS)* cloud model, which relies on a platform providing the developers with an extensive set of services that should be well integrated into the application.

In this paper, we build upon experience with cloud application development, consultancy and evangelization, which helped us to observe many challenges associated with cloud application design and development, especially when PaaS services are employed. The contribution of this paper is the identification and evaluation of architectural tactics for the design and development of efficient PaaS cloud applications, meaning the applications deployed to the cloud that do not make wasteful use of allocated resources. Furthermore, we study the suitability of the tactics in different situations and the benefits of their combination with respect to the targeted application's quality.

The discussion of the tactics is related to one of the biggest real-world projects we have participated in, which involved the architecture design of a private social network supporting education on elementary schools in the Czech republic. Our role as the consultant was the architecture design of the cloud backend under strict performance constraints.

The paper is structured as follows. After the discussion of related work in *Section II* and outline of the case study system architecture in *Section III*, *Sections IV* and *V* are dedicated to the presentation and evaluation of seven architectural tactics for PaaS cloud application design. We conclude in *Section VI*.

II. Related Work

When designing a software architecture, it is a common good practice to use existing design patterns [2], [3], which are however not specifically designed for cloud applications. Therefore new catalogs of patterns relevant for the cloud are emerging [4], [5], that are however not yet supported with demonstration in real world settings, which could reveal the hints for efficient employment and combination of these patterns. One of the most relevant sources of cloud patterns is [6], which provides the most detailed pattern description and outlines their impacts. This source however lacks the discussion of the effective combination of the patterns and their relation to the cloud-application quality attributes. Explanation of software quality attributes generally valid also for the cloud is for instance in [7], [8].

Besides industrial guidance, we have surveyed research papers elaborating specific performance challenges, their root causes and possible solutions. However, we have found that most of the cloud research focuses on the IaaS service model studying the principles of virtualization [9], effective virtual machine distribution [10], behavior of the underlying infrastructure [11] or providing tools for IaaS cloud simulation [12], while the PaaS services are left sideways.

Another source of guidance on the design of cloud applications are the white papers published by cloud providers to help

D. Gesvindr – Professional awards and memberships: Microsoft Most Valuable Professional: Microsoft Azure, Microsoft Certified Solutions Expert: Data Platform, Windows User Group board member

978-1-5090-2131-4/16 $31.00 © 2016 IEEE
DOI 10.1109/WICSA.2016.42

software architects and developers to correctly design scalable and highly available applications [13], [14], [15]. Although such papers provide the developers with very good high-level understanding of the domain, not much attention is being paid to the specific issues addressed in this paper.

Further information on architecting scalable applications for the cloud originates from software architects and chief technology officers working in leading companies [16], [17], [18], [19] and providing the community of cloud practitioners with their personal experience in architecting large-scale cloud applications, detailing their design mistakes and lessons learned from them. Moreover, positive examples of successful cloud application design include a number of large-scale online services running typically in private rather than public cloud. These services include Wikipedia [20], Facebook [18] and Stack Overflow [19]. Although these texts are highly valuable, their discussions are not put into the perspective of wide range of performance-related quality concerns studied in this paper.

III. Solution architecture overview

For the demonstration of the addressed tactics and their quality-related effects, we have chosen a system designed as a private social network supporting education on elementary schools in the Czech republic. This application is intended to support communication between teachers and students. The main component is the *wall*, where users can see and post messages based on the class they belong to and subjects they are enrolled to. Teachers can easily submit new messages, polls, files and homework targeted to specific classes, subjects or students. Our primary responsibility was to design the backend service in such a way that it can sustain the load (over 4 000 schools and 800 000 students in the Czech Republic), which means:

- The service can fluently scale together with growing number of users and generated load.
- The used architecture and technologies do not impose any insurmountable obstacle in service scalability that would later require complete service redesign (minor performance tuning updates are acceptable).
- The service has effective operation costs which scale fluently with the amount of active users.

The solution architecture of the designed cloud application, based on Microsoft Azure cloud, which resulted from the design decisions elaborated in this paper, is depicted in *Figure 1*. The architecture consists of the following components.

a) Web client: The client for this application is a single-page JavaScript application running in a web browser, which takes advantage of the responsive design to be optimized for the use on high variety of devices including mobile and touch-enabled devices. The web client communicates with a REST API, opens a web socket notification channel using ASP.NET signalR and downloads files from Azure Storage.

b) REST API: REST API is the primary communication interface serving a majority of client requests. All data for the client excluding binary files are loaded using this API which is

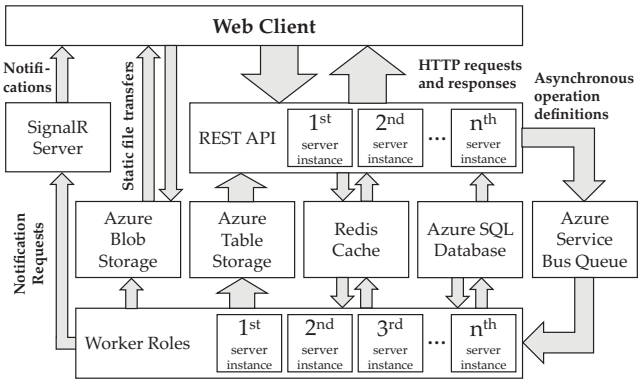

Fig. 1. Architecture of the designed solution.

implemented as a PHP application hosted as Web Application in Azure App Service. This component loads data from Azure Table Storage, Redis Cache and Azure SQL Database.

c) ASP.NET SignalR Server: A notification server which delivers notifications to connected users using web sockets. Notifications are generated by workers and delivered to this server via an internal REST API.

d) Storage Layer: Data of the application is stored in multiple PaaS storage services as explained later, which includes both relational database (Azure SQL Database) and multiple NoSQL databases (Azure Table Storage, Redis Cache).

e) Worker Role: Worker role is an application which runs in an infinite loop and loads messages from the Azure Service Bus Queue where the REST API stores serialized commands with data modifications. This application mediates writes to the relational database, updates data in the NoSQL databases and generates real-time notifications.

IV. Storage and Data Access Tactics

As there is a rich set of storage services provided by PaaS cloud provides, it was necessary to first comprehensively evaluate their functionality, performance, availability and associated costs. Since the case study presented in this paper is based on Microsoft Azure cloud, *Table I* presents the results of our comparison for this provider. The comparison of storage services for other cloud providers is from the perspective of their features and qualities very similar, and also the results of this paper are portable to other cloud platforms.

Based on the comparison in *Table I*, it is clear that there is no universal storage service that would outperform all other services in terms of rich query options, scalability, high throughput, low response time, and operation costs. Therefore, to create a high-throughput storage layer, it is necessary to effectively combine multiple storage services to design the software architecture that meets desired quality criteria. In this section, we examine relevant existing architectural tactics and formulate their analogies for PaaS cloud, with respect to this goal.

TABLE I
COMPARISON OF AVAILABLE STORAGE SERVICES IN MICROSOFT AZURE

	Azure SQL Database	**Azure Table Storage**	**DocumentDB**	**Redis Cache**
Data Model	*Relational*	*Persisted Key-Value*	*Document*	*In-memory Key-Value*
Format of stored data	**Rows** stored in a table with **fixed schema** with one or multiple columns	**Rows** stored in a table **without fixed schema**, where each row can have different columns.	**JSON object**	**Key-Value pair**, where value is type of string or set of strings
Integrity enforcement	**Complex** (Primary key constraint, unique key constraint, data type of the column, foreign key constraint, check constraint)	**Limited** (partition and row key must be together unique)	**Complex** (JSON object validation rules written in JavaScript)	**Limited** (uniqueness of the key)
Query support	**Complex**	**Limited** (using partition and row keys)	**Complex**	**Limited** (using key)
Availability	**Very High** (99.99%)	**High** (99.9%)	**High** (99.95%)	**High** (99.95%)
Scalability	**Limited** (different performance tiers, premium performance is expensive; manual data sharding possible)	**Unlimited** (requires correct use of partition key)	**Unlimited** (requires manual data sharding to multiple collections)	**Limited** (different performance tiers)
Throughput	**Low** (200 concurrent request for standard S3 tier; max. 2 400 request for premium P11 tier)	**High** (1 000 req/s per partition)	**Medium** (250-2 500 req/s per collection depending on performance tier)	**Very high** (600-250 000 req/s depending on performance tier)
Cost	Storage: **low** Performance: **high**	Storage: **very low** Performance: **very low**	Storage: **medium** Performance: **low**	Storage: **very high** Performance: **medium**

A. Materialized View

*1) **Motivation**:* One of the common cloud application issues is the architecture design relying on a single non-scalable relational database serving all application load, because for most on-premise scenarios the database server scales well enough. Another poor practice we identified on multiple projects in this context is the over-usage of database abstraction techniques (O/R mappers etc.), which often leads to unnecessary overloading of the database because developers easily lose track of the associated load costs.

To increase the scalability of a software component or service, Taylor et al. [8] recommend that system bottlenecks should be avoided by design. In case of the cloud application the bottleneck is the relational database with limited throughput. First we should question whether we can easily lower the database load by redesigning the application logic to minimize the number of needless DB calls per user request. Furthermore, we should consider replacement of the relational database with another cloud storage service (e.g. NoSQL database), which may however not be feasible due to the required storage feature set. Relational database is the only storage providing complex transaction support with built-in support for data integrity enforcement (most commonly unique and foreign key constraints), which are both desirable features of the primary data storage.

*2) **Tactic Description**:* This tactic is described as a software design pattern that generates prepopulated views over data in one or more data stores when the data is formatted in a way that does not favor the required query operations. This pattern can help to support efficient querying and data extraction, and improve application performance [6].

A mechanism of a materialized view (indexed view) is also used in relational databases, in a way that a normal database view (stored SELECT statement) inlines its definition to a calling query and the database server evaluates a single query against stored data, giving database view no performance benefit. In case of materialized view, the query in its definition is evaluated and results are persisted in form of a table, which is recomputed with every change of source data. This approach, when properly used, can bring significant performance benefits, especially when applied to views with aggregations over huge data sets or views with complex queries involving joins over many tables, which are less frequently updated. The more frequently the data is read between updates, the higher the efficiency of this tactic is.

In the cloud, we would like to offload the relational database service, therefore we may use this tactic to precompute views on data which will be stored mostly in NoSQL databases due to their high scalability. Application can then load data from a highly scalable storage in a form that requires minimum processing at the application server and can be immediately sent to the client.

*3) **When not to use**:* This tactic should not be used when source data is frequently updated, because each update triggers either complete or partial updates of the materialized view. This may not be an issue when updates are done asynchronously, but when implemented synchronously (due to strict data consistency requirement), the performance of data modifications significantly decreases. The guidance on when to use this tactic is provided in *Table II*.

*4) **Our implementation**:* The cloud application we use for demonstration employs this pattern to combine two storage services—Azure SQL Database and Azure Table Storage. The objective is to combine those services in such a way that their strengths (see *Table I*) prevail and weaknesses are mitigated.

First, we designed the relational database based on the data

TABLE II
GUIDANCE ON THE MATERIALIZED VIEW TACTIC

		Storage read frequency	
		low	high
Storage write frequency	low	Small benefits relatively to implementation effort, use a single storage	Most benefits, low issues
	high	No benefits, use single storage (relational DB) and if not sufficient use Indirect Dependency Tactic to load data to storage	If strict data freshness required, then not applicable; otherwise use Indirect Dependency Tactic to load data and refresh cache in an asynchronous manner.

model of our application. During the database design, we payed attention to ensure that the designed database tables are in a normalized form to store data efficiently and without redundancy. We did not make any optimization on the schema level to support more efficient querying of the data except index design.

Based on our analysis, most of the load will be caused by the subsystem responsible for storage and retrieval of wall records (95% of expected API calls). Therefore we focused on the wall record entity and how it is displayed in the application. When the user loads the wall, the last 15 wall records are displayed in a compact form including preview of the text and preview of the last two comments. We discovered that the displayed data is (in the normalized DB schema) loaded approximately from 10 tables. If such a complex query or set of queries was executed every time the data is requested from the client, the throughput of this service would be limited to less than 100 requests per second (due to the limited throughput of the database). Vertical scaling of the database is not an option because of expensive high-performance tiers in comparison to other storage services.

To offload the traffic to Azure Table Storage, it was necessary to circumvent the limited querying capabilities of this service to form queries retrieving data based only on the partition and row key. The application logic, which decides which posts and in what form are visible to a specific user, is too complex to be evaluated during every call. Therefore we store in the Azure Table Storage for each user a copy of their completely prepopulated wall. This gives us extremely efficient querying on cost of high data redundancy, which does not cause (due to storage cost $0.07 per GB) any significant additional operation cost. Design of the data partitioning strategy is discussed in *Section IV-B*.

It is important to discuss in what form we store entities in the Azure Table Storage to introduce other related benefits. Our goal was to offload processing from the web servers hosting the API to increase their throughput. We achieved this by storing wall record entities in the most complete form so that it contains all data attributes necessary for construction of any variant of the wall record data transfer object (DTO) which is sent to the client. Two forms of the wall record DTO can be sent from the server—a compact form which is displayed as an item on the wall and a detail form which is displayed when the user selects post on the wall and opens its detail. We persist all records in the full-detail form as the limited one can be easily derived from it by limiting the set of attributes.

All complex transformations are computed and persisted when a new record is stored in the Azure Table Storage and because the record is read in order of magnitude more often than it is updated, it significantly saves resource expenses.

To sum up, our design decreases the load on the wall by:

- Submission of a single query to Azure Storage.
- Retrieval of the first 15 records using the most efficient type of query offered by this service (or next 15 records based on the continuation token).
- Copying some data attributes from objects returned by the query to final data transfer objects without further processing.
- Return of results to the client.

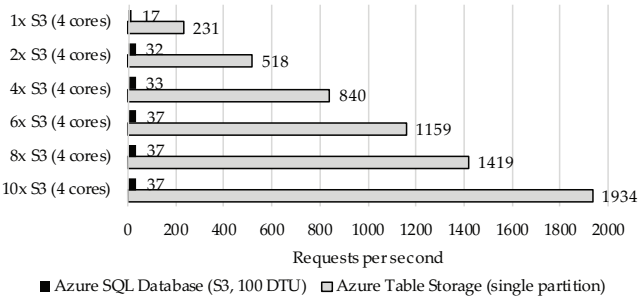

Fig. 2. Comparison of API throughput with and without materialized view

*5) **Tactic impact**:* To evaluate the impact of this tactic on the throughput of the optimized API, we conducted a set of performance test (which loaded the wall of a single user) with and without the use of this tactic, with results presented in *Figure 2*. The API was hosted in Azure App Service using S3 instance size (4 CPU cores per instance). The maximum achieved throughput in the test roughly corresponds to 120 000 concurrent users of the implemented service.

Positive effect of this tactic is in our case clearly measurable and allows us to scale this service to the desired extent.

Negative effects of this tactic related to significantly increased complexity of data modification had to be mitigated by application of the Command and Query Responsibility Segregation in combination with asynchronous write processing using messaging, as described in later sections.

*6) **Our contribution**:* Our contribution to this architectural tactic is the identification of its surprisingly massive impact on scalability and throughput of the PaaS cloud application, from which many read-intensive applications can benefit. We have also identified that despite the fact that the use of this tactic is recommended together with the Command and Query Responsibility Segregation (CQRS) pattern (see below), it is nowhere recognized that this tactic becomes nearly useless without asynchronous update processing via messaging.

B. Sharding Pattern

1) ***Motivation****:* Applications often store data in a single storage. But the problem is that such a single storage can in most cases efficiently scale only vertically—for instance adding more CPU power to the database server which operates the database. The main issue is that vertical scalability has its limits in term that not much processing power or storage can be added to a single storage node. When we reach the limit or upgrade costs exceed certain efficiency (as costs of high-end hardware does not rise adequately with the performance) we will have to separate the data into more storages or databases to scale horizontally and overcome the vertical scalability limit.

2) ***Tactic Description****:* In [6] the Sharding pattern is described as a division of data store into a set of multiple horizontal partitions (shards). For each row, its partition is selected based on a value in a partition key, which is one of the columns in the table. Partitions are then distributed and stored across multiple servers, therefore the load may be distributed, until queries work only with a single partition. Partition key should be selected in such a manner that the load is evenly distributed across as many partitions as possible and at the same time most of the queries are working with data from a single partition.

3) ***When not to use****:* This tactic should not be used when structure of stored data or data access patterns disallow the separation of data set into distinct partitions because of frequent queries that would work with data from multiple partitions. That is because this is for most storage technologies inefficient due to synchronization overhead when results from multiple nodes have to be collected together.

4) ***Our implementation****:* Because we have used Azure Table Storage as the storage for the data generated by the *Indexed View architectural tactic*, it was crucial to design a robust partitioning strategy. In the case of Azure Store, it is the key for unlocking its unlimited scalability potential. Azure Table storage uses two-part shared key with the following properties.

a) Scalability: Rows with the same partition key are stored at the same storage server. Rows with different partition keys may be stored on different servers. This service is responsible for internal data distribution across hundreds or even thousands of servers in the data center. To unleash the unlimited scalability of this service it is necessary to distribute data across reasonable number of partitions depending on the load.

b) Query efficiency: The most efficient queries (with low response time) are when the filter condition contains an exact match of the partition key and at the same time an exact match of the row key (fastest) or range match of the row key (if range is not too wide) or returns only first few records. Queries across multiple partitions have declines in the response time. Performance of queries filtering based on other non-key fields is inconvenient.

c) Batch updates: It is possible to modify multiple rows with the same partition key in the batch gaining significant performance benefit.

d) Results order: Data is returned automatically in an ascending order primarily by the partition key, secondarily by the row key.

e) Comparison operator: Both keys are strings and as strings are also compared (including inequality operators greater than, etc.). This requires us to use numbers with fixed digit count and leading zeros.

For the *partition key*, we decided to use internal user identifier which is provided for each API request by the authentication system. This gives us the desired scalability of read operations as data of each user may be stored on a different server. When multiple users load data, their load is automatically evenly distributed across high number of servers. At the same time we always load data for a single user and a single partition queries have desired response time.

The *row key* in combination with the partition key must be unique in the table. It is an advantage when it also ensures, that wall records for a given user are returned in descending order based on time of their creation. This order is advantageous because the most frequent query is expected to be the load of *n newest* record on the user's wall which can be now translated as the load of *n first* records in the partition.

The biggest challenge related to this tactic for us was to implement data filtering with multiple criteria in the Azure Table Storage which supports only queries filtering based on partition and row key. Because the application supports the concept of class and subject walls, which are intended for communication only with enrolled students, we need to filter wall record accordingly to provide this type of views in addition to the global view, which contains all wall records visible to the user. In both cases we need to filter records based on their type—private message, homework and poll.

To achieve this functionality, we need to store multiple lists of wall records for each user based on filtering criteria. This tactic increases data duplicity (which is not an issue due to very low storage costs) but provides very effective data retrieval. To support filtering based on type of the record, we created in the Azure Table Storage multiple tables, one for each record type plus the global one. The same wall record is stored in multiple tables based on its type (for instance a record of type homework definition is stored in tables WallRecords and WallTasks). Now it is very simple to load all types of records or just a specific one only by changing table name in the query without any performance or scalability degradation.

If the wall record is published for the class or for the subject, additional copy is created and '-class-{ClassID}' or '-subject-{SubjectID}' is added as a suffix to the partition key allowing efficient filtering based on class and subject.

5) ***Tactic impact****:* In our benchmark we were able to achieve a total throughput of a single partition in the Azure Table Storage over **6 000 queries per second** retrieving a single entity, which is a surprising result, because the number

of retrieved entities per second is three times higher than the limit specified in the service documentation [21].

We also measured an average response time of Azure Table Storage, which is **29 ms** for lookup of any entity based on the partition and row key. When the entity was recently loaded, average response time is **9 ms**.

The primary disadvantage of this tactic is a high redundancy of stored data, where a wall record of type homework definition published to a class with 30 students and one teacher is stored in the Azure Table Storage in $31 \times 2 \times 2 = 124$ copies (not identical records, they vary in their keys, but each copy takes into account the user, their permissions, role in school, etc.). Because the indexed view needs to be updated with every change in any wall record or after posting a new one and this update was previously implemented synchronously, API calls modifying the wall started to time-out which is unacceptable. Therefore we had to implement asynchronous updates using messaging.

6) ***Our contribution****:* Our main contribution is in the identification and demonstration of an alternative practice for storing data, which from the initial point of view of many developers experienced with relational-database development seems to be absolutely inefficient due to needless data redundancy. This was the same in our project where after mind-switch of our developers it was surprising for everyone that the NoSQL databases fulfilled all the quality requirements—high scalability and low response time for read operations and low operation costs, despite the redundancy.

C. Static Content Hosting

1) ***Motivation****:* Even though web applications are currently composed mostly of dynamically generated content, some resources of the web application are static resources, such as images, style sheets, client side scripts or separate downloads (for instance PDF or video files) [6]. Very common anti-pattern we observe is that both dynamic and static content is delivered by a single server which is optimized for dynamic content generation.

In the cloud, environment application servers suitable for generation of dynamic content are billed based on their performance. If we unnecessarily use application servers capable of dynamic content generation for distribution of the static content, we waste valuable resources, thereby also increase operation costs despite the fact that cloud providers offer highly scalable and cost-efficient services dedicated to the distribution of the static content including the support for scalability extension using content delivery network services.

We see an increased need for the application of this tactic because it can significantly decrease operation costs of many nowadays popular single-page applications, which consist of static resources, including client-side scripts that load dynamic content served through REST API, which is in many cases the only component of the application requiring computational resources associated with a specific application server.

Another anti-pattern we observed is that static content (mostly images) is stored in the relational database and is loaded by the client through the application server. The motivation for this approach is the development comfort associated with a single unified storage, which is redeemed on cost of expensive non-scalable data storage for binary data and unnecessary additional load of expensive application servers. For applications depending on a single relational database is this anti-pattern very dangerous, as it may significantly affect the performance of the entire service depending on the overloaded database.

2) ***Tactic Description****:* This tactic encourages the developers to separate hosting of the static content from the dynamic content of the same web application, thus decreasing the demand for expensive computational resources [6].

3) ***When not to use****:* Challenges related to this tactic are the following.

a) Restricted access: For sensitive static content it is necessary to apply user authentication and authorization when accessing the resources. The use of the same authentication mechanism as for the rest of the application may be applicable on proprietary web servers hosting the static content, but most of the cloud providers do not provider an option to implement custom authentication for their storage services. This issue may be solved with the help of the *Valet Key* tactic.

b) More complex application deployment: As the web application itself is deployed across multiple services, by separating dynamic and static content the deployment process becomes more complex and it is desirable to automate application deployment to prevent mistakes.

c) Limitations of the storage service: Deployment of this tactic may become complicated due to lacking support of required technologies at the storage service side—HTTPS support, custom domain support (especially in combination with HTTPS). Missing custom domain support requires proper handling of cross-origin resource sharing at the application level.

4) ***Our implementation****:* Because the web client of our application is a single-page application, we take advantage of this tactic to store all publicly available static content in a public container of Azure Blob Storage service. This freed capacity of application servers and increased the throughput of our REST API.

5) ***Tactic impact****:* The use of a specialized service for the distribution of static files instead of an application server offers significant throughput advantage as depicted in *Figure 3* and response time advantage as depicted in *Figure 4*. Performance measurements were conducted using *wrk tool*, a HTTP benchmarking tool, running on Azure 8 CPU core VM in different data center, using 12 threads and 256 connections for the benchmark. In the test, a 85 KB JavaScript file was downloaded.

6) ***Our contribution****:* Our contribution is in the quantification of the performance advantage of the dedicated storage service. Our benchmarks discovered an insurmountable

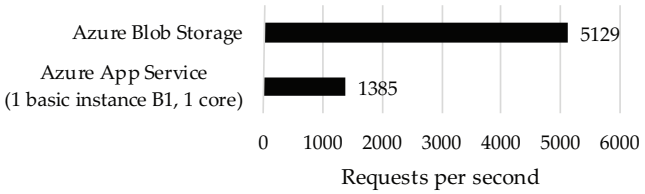

Fig. 3. Throughput of Azure Blob Storage vs. Azure App Service.

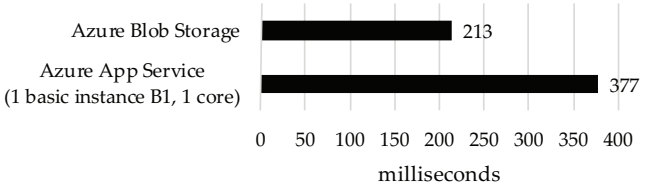

Fig. 4. Response time of Azure Blob Storage vs. Azure App Service for 90% of fastest requests.

throughput limit of 2 200 requests per second even for over-provisioned services. This unexpected and surprising limit is likely caused by the performance limit of the load balancer which is a part of the platform. With the discovery of this limit, the importance of this tactic significantly increases, as many small file transfers served by the same application server as the API can easily saturate the load balancer, thus decreasing the throughput of the API and keeping the application servers underutilized and cost inefficient.

D. Command and Query Responsibility Segregation

1) ***Motivation****:* Common practice for current web and line-of-business applications is that the application has a single data model used for both data querying and data manipulation (CRUD, i.e. create, read, update and delete operations).

As our needs has become more sophisticated, we have realized that having unified model and data transfer objects (DTO) for read and write operations brings some issues [22]:

- When reading data, we may want to read the information from the data store in a different form—applying aggregations, adding computed attributes and on the other hand hiding some internal attributes, forming virtual records, etc. The same information can have multiple representations.
- When updating data, we usually need to apply additional validation rules and we may want to update attributes of the entity, which are never again read in their original form, therefore missing in the read model where they might be present only in their recomputed form.

2) ***Tactic Description****:* Command and Query Responsibility Segregation (CQRS) is a pattern that segregates the operations that read data (Queries) from the operations that update data (Commands) by using separate interfaces. This implies that the data models used for querying and updates are different [6].

3) ***When not to use****:* In case of this pattern, it is important to understand that the use of a single data model for read and write operations is not automatically a bad practice just because there is a pattern that can separate them. Implementation of CRUD using a single data model is a well suitable practice for many applications, which does not bring unnecessary complexity to their software architecture.

The CQRS pattern should only be used when the application gets performance benefits by its implementation or the read and write model are significantly different. Otherwise, this pattern will significantly increase software architecture complexity and related development costs.

4) ***Our implementation****:* When we implemented the first version of the materialized view tactic in our application, we realized that it is necessary to capture all data modifications which lead to an update of the materialized view in a highly structured way not to get lost in dependencies. We found out, that capturing data modifications at the repository level is not a good practice because many update operations are complex transactions which consist of multiple individual operations and we needed more precise control over updates of the materialized view.

Therefore we created a separate command class for each update operation that can be submitted via the API by the user of the service. Other important decision is, that we separated the command into 2 classes—one class contains only serializable inputs of the command, the second is only a command implementation that receives an instance of the class with inputs as a parameter of the Execute method.

By implementation of isolated update commands we naturally separated read and write model for performance-critical operations. The read model is reading most of the data from indexed views in Azure Table Storage. Few operations where a high rate of request is not expected still query directly the relational database (if operation will prove problematic with increasing the load, the database access will be replaced with the materialized view).

It is important to mention that not all components of the application use CQRS—the administration interface with very low expected load is written using a single data model for CRUD operation. Only in case any operation would impact validity of any materialized view, specific command to update the materialized view is executed.

Despite the implementation of CQRS which in our case made the software architecture more transparent we still encountered significant performance issues (their roots explained in Section IV-B5) when the update commands were executed synchronously by the application server which processed the API calls. Because the CQRS pattern itself does not specify how the command should be executed we decided to shift from synchronous processing to message based asynchronous processing.

5) ***Tactic impact****:* The CQRS tactic itself does not have a direct impact on a performance of the application as it is just a way to restructure the code without any changes to its functionality. Performance issues of the update operation are measured in *Section V-A5*. But the application of this tactic is a required condition to shift from synchronous to

asynchronous command execution, which is intended to significantly improve update performance.

*6) **Our contribution**:* Our contribution is in a seamless combination of these patterns with other described tactics, which leads to significant performance improvements and well-organized software architecture with clear dependencies.

V. Messaging and Data Processing Tactics

In this section we present tactics related to a design of asynchronous messaging infrastructure we have used in the implemented project. Our primary motivation for implementation of asynchronous messaging patterns in the project were performance issues bound to updates of complex materialized views, which caused synchronous update commands sent by the client even to time-out. Because of poor scalability and throughput of update operations we had to rethink the architecture in a way, that would lead to a reliable high throughput read and write operations.

A. Asynchronous Messaging

*1) **Motivation**:* Software architecture of an application determines how an incoming request from the client is going to be processed, which components are called in which order, how the results are being generated and transferred back to the client. Most of the current web applications utilize synchronous request processing, i.e. when a request is received by the web server, it is put into a queue of incoming requests in the web server and is processed as soon as the web server has an available processing thread. Request processing is done synchronously, the request is blocking the processing thread until the result is generated and the response is sent back to the client. This involves synchronous calls to the database storage, and execution of methods generating the output.

Synchronous request processing has an advantage in low development costs, because it does not introduce additional complexity into the architecture. The main disadvantage is low scalability because every incoming request that is immediately processed has many associated resource costs and when certain resource becomes overloaded, processing of the request fails due to time-out, often with no retry mechanism being implemented. In case there is a load balancer in front of the web servers, random load distribution among web servers may in case of resource intensive requests lead to uneven load distribution, with some servers being overloaded and the remaining being idle.

*2) **Tactic Description**:* An opposite approach, described by the Indirect Dependency Tactic [8] or Asynchronous Messaging Primer [6] is to receive a request, store this request in a queue service and let a continuously running worker withdraw requests from the queue and process them. This architectural tactic is recommended as a way to increase system scalability by utilization of indirect dependencies.

*3) **When not to use**:* For a software architect it is challenging to assess if the benefits of asynchronous request processing in the designed application outweigh the drawbacks in terms of the increased architecture complexity and higher development and operation costs.

Application of this tactic becomes unfeasible when the client requires immediate response which is based on results of asynchronously executed command, for instance return of inserted record identifier which is generated in the database as part of the transaction that stores the record.

*4) **Our implementation**:* To solve our problem with the performance of update commands we altered the way how these commands are processed by the application server and when they are executed. To increase performance, the application server does only very basic command validation which includes:

- Authentication of the user.
- Authorization for the operation but only in the case that it does not require complex permissions evaluation, otherwise it is and authorization part of message processing.
- Basic validation of the received command, which does not require data access.
- Storing of the serialized command in the scalable queue service.

To process queued commands, we implemented and deployed a worker role, which is a stateless virtual server where our application is deployed, which in an infinite loop fetches commands from the queue and executes them.

During implementation of this tactic we identified multiple trade-offs we had to deal with. The major challenges related to this tactic include:

a) New record identifiers: When a user inserts a new wall record, the record appears on the wall as soon as the API confirms that it has successfully received the request. It passed basic validations and the command is stored in the queue, so that it is a guaranteed that it will be processed by the worker. The problem is that the primary key of the record is generated in the database (due to benefits associated with numeric auto increment primary keys), and even write to the database is done asynchronously, therefore this operation cannot return identifier of the inserted entity to the client. But when the record is visible to the user, the user can invoke another operation (for instance mark the post as favorite) that requires a record identifier. To solve this issue we decided to add an GUID alternate key to the entity which is used only by the client instead of the primary key. The benefit of this alternate key is that the API is responsible for its generation, thus it is stored in the insert command and immediately returned to the client and the database only enforces its uniqueness.

b) Results from asynchronous operations: Because most of the resource-intensive operations are processed asynchronously, the client is not aware of their completion and cannot immediately consume their results. On the other hand, the results visible to the client are written by asynchronous operations to the database and mostly to indexed views, but

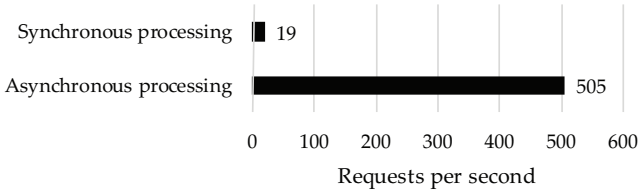

Fig. 5. Comparison of the API throughput between synchronous and asynchronous inserts of new wall records.

still the client does not receive the impulse to download them. It is inefficient to poll the API for new data every few seconds therefore we used web sockets technology to allow the worker to send notifications directly to the clients and efficiently notify them about completed operations. This approach has brought a completely new level of interactivity and efficiency, because as a part of the change notification we deliver via the web socket straight the modified entity which is immediately displayed by the client. When somebody inserts a new wall record, based on the permissions this record is automatically displayed to other connected clients with a few seconds delay.

c) Interoperability: It was an implementation challenge to connect REST API of our service which is implemented in the PHP with a worker process implemented in .NET. We had to define identical classes holding command input data as explained in *Section IV-D4* both in PHP and .NET and ensure that instances of the class are serialized to JSON and stored to the queue in the same way as they are loaded in the .NET worker process where they are deserialized from JSON into instances of the same class and passed to the command to execute.

*5) **Tactic impact**:* As depicted in Figure 5 this tactic had a significant impact on performance of update commands which lead to update of indexed views.

*6) **Our contribution**:* Our contribution related to this tactic is in the implementation of infrastructure stack for asynchronous messaging which significantly simplifies the implementation of this tactic and individual commands. Another contribution is a solution of the described problems related to missing record identifiers and efficient use of real-time notifications via web socket technology which have not been discussed in other sources at sufficient level of detail yet.

B. Competing Consumers

*1) **Motivation**:* The implementation of the asynchronous messaging tactic will lead to uneven allocation of processing power in case of multiple message producers and a single message consumer, which may become a bottleneck in the architecture, significantly overloaded in the time periods of higher load as the messages pile up in the queue. Despite the fact that the asynchronous messaging tactic will guarantee that none of the client requests will timeout, as the responses are generated immediately, the message stored in the queue may be processed after significant delay which may not be in compliance with user requirements and specified quality criteria.

*2) **Tactic Description**:* Competing consumers tactic is a natural extension of *asynchronous messaging tactic*, which enables multiple concurrent consumers to process messages received on the same messaging channel. This tactic enables a system to process multiple messages concurrently to optimize throughput, to improve scalability, and to balance the workload [6].

*3) **When not to use**:* This tactic is not suitable when the messages (tasks, commands) have to be processed in an exact order. Because of parallel processing it happens that due to different processing times of individual messages, tens or hundreds of messages can outrun a single long processing message.

*4) **Our implementation**:* Implementation of this additional tactic which is an extension of the asynchronous messaging tactic, was very straightforward, because the worker roles service we used for the hosting of the worker process supports horizontal scalability, which allows the deployment of the identical application code on multiple virtual servers. Because the instances of the worker process share also configuration they all point to the same queue from which they load messages for processing.

Despite great increase of scalability, during implementation of this tactic we came across multiple architecture and algorithm design problems that we had to resolve:

a) Message ordering: When multiple workers process messages before a single message is completed, multiple newer messages can be already processed, which is an issue when messages have dependencies and should run in a guaranteed order. We have solved this issue by letting a message during its processing submit other messages to the queue, which gives us guarantee of the order in which the messages are processed in case it matters.

b) Idempotency: Our implementation of the queue service provides a guarantee that message will be processed at least once, but not exactly once. Despite the fact that a message is locked in the queue when a worker loads it, to prevent other workers from processing the same message, it may happen that due to an error in processing, the message is rescheduled and processed once again. Therefore the system should be designed in a way that this causes no problems—for instance insertion of a duplicate record.

c) Poison messages: A poison message is a message that cannot be successfully processed by the worker and usually causes an exception during its processing. The .NET client library which we use for loading of the messages from the Azure Service Bus Queue service automatically detects when a message is loaded form the queue and its processing fails multiple times. Such a message is automatically moved to the Dead Letter Queue where it waits for further analysis by the developer of the service. We consider it important to analyze messages in this queue as they can reveal bugs in the system.

d) Throughput of the messaging service: The messaging service has its throughput limits we have to be aware of. We recommend to use batch processing when messages are

written to the queue. It is possible either to increase a queue throughput by separating the load into multiple queues or decrease the load by reading messages in batches.

e) Message scheduling: Azure Service Bus Queue supports submission of messages that become visible to workers at specified time. We use this feature to implement some of the future actions—for instance deadlines to submit homework. Every time somebody sets a deadline for homework, we schedule a message with a command that becomes visible few seconds after the deadline. This command on the beginning of its execution has to check its validity (if the deadline was not changed again) and refresh indexed views so that users see the homework locked.

*5) **Tactic impact**:* We consider this tactic to be an advantageous extension of the asynchronous messaging tactic which significantly improves its scalability. Especially in the cloud environment, we can benefit from elasticity of the cloud and easily horizontally scale the number of workers consuming and processing messages from the queue.

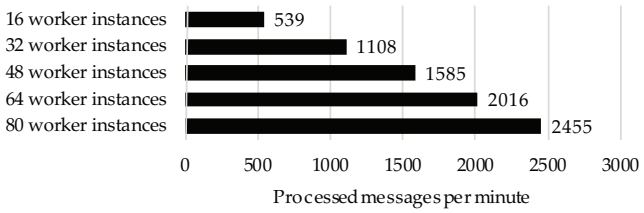

Fig. 6. Throughput comparison of worker processes depending on the number of deployed instances

Scalability of this tactic is demonstrated in Figure 6, where the numbers of processed messages per minute are compared based on the number of deployed worker servers. It is evident that the throughput of our worker layer scales nearly linearly with the number of processing nodes. The throughput of 80 worker instances roughly corresponds to 300 000 users of the implemented application.

*6) **Our contribution**:* Our contribution to this tactic is in the solution of the described issues, especially how we deal with a message ordering and idempotency, which allows us to deploy workers at massive scale as demonstrated, without the risk of data corruption by parallel processing of queued commands. Our approach to message ordering also allows us to divide complex commands into a set of small atomic commands, which can be in case of failure repeated. This leads to an increase in the resiliency of the asynchronous message processing. Another contribution is in our approach to idempotency, when we clearly divided commands into two groups: the first where there are commands which cannot be repeated due to the nature of the operation that would create duplicates records—those commands are protected by using unique identifiers generated at the time of command scheduling which blocks addition of the same record twice, and the second group with the commands that can be safely repeated.

VI. Conclusion

In this paper, we have reported on our experience with the architecture design of PaaS cloud applications, and destiled our observations into a set of architectural tactics (based on existing tactics used in other domains), which are aimed at supporting software architects in their PaaS cloud application design. As distinct to related work, we have focused on both pattern and anti-pattern effects of the studied tactics, and their combinations, highlighting their impact on the quality characteristics of the cloud applications based on PaaS infrastructure.

In future, we would like to implement an automated support for the evaluation of the tactics and their impact on various quality characteristics of cloud applications in the PaaS.

References

[1] P. Mell and T. Grance, "The nist definition of cloud computing," 2011.

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

[3] M. Fowler, *Patterns of Enterprise Application Architecture*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

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

[5] B. Wilder, *Cloud Architecture Patterns*, 1st ed. O'Reilly Media, 2012.

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

[7] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*. Addison-Wesley, 2003.

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

[9] J. Sahoo, S. Mohapatra, and R. Lath, "Virtualization: A survey on concepts, taxonomy and associated security issues," in *Computer and Network Technology (ICCNT), 2010 Second International Conference on*, April 2010, pp. 222–226.

[10] J. T. Piao and J. Yan, "A network-aware virtual machine placement and migration approach in cloud computing," in *Grid and Cooperative Computing (GCC), 2010 9th International Conference on*, Nov 2010, pp. 87–92.

[11] N. M. K. Chowdhury and R. Boutaba, "A survey of network virtualization," *Computer Networks*, vol. 54, no. 5, pp. 862 – 876, 2010.

[12] R. Buyya, R. Ranjan, and R. Calheiros, "Modeling and simulation of scalable cloud computing environments and the cloudsim toolkit: Challenges and opportunities," in *High Performance Computing Simulation, 2009. HPCS '09. International Conference on*, June 2009, pp. 1–11.

[13] T. Ganesh. (2012) Designing a scalable architecture for the cloud. [Online]. Available: https://www.ibm.com/developerworks/community/blogs/theTechTrek/entry/designing_a_scalable_architecture_for_the_cloud3

[14] J. Varia, "Architecting for the cloud: Best practices," *Amazon Web Services*, 2010.

[15] B. Jimerson, "Software architecture for high availability in the cloud," *Oracle Technology Network*, Jun. 2012.

[16] Y. Nelapati and M. Weiner, "Scaling pinterest," InfoQ, 2013.

[17] S. Anand, "Netflix's cloud data architecture," InfoQ, 2011.

[18] A. Sobel, "Scaling the social graph: Infrastructure at facebook," InfoQ, 2011.

[19] M. Cecconi, "The architecture of stack overflow," code.talks, 2011.

[20] "Wikimedia servers," https://meta.wikimedia.org/wiki/Wikimedia_servers, 2015, accessed: 2015-10-25.

[21] "Azure storage scalability and performance targets," https://azure.microsoft.com/en-us/documentation/articles/storage-scalability-targets/, 2015, accessed: 2016-01-23.

[22] M. Fowler. (2011) CQRS. [Online]. Available: http://martinfowler.com/bliki/CQRS.html