Microservice Complimentary Groups Determination Algorithm for the Effective Resource Usage

Oleksandra Dmytrenko [1], Mariia Skulysh [1], Larysa Globa [1]

[1] *Igor Sikorsky Kyiv Polytechnic Institute, 37, Prospect Beresteiskyi, Kyiv, 03056, Ukraine*

Abstract

One of the primary challenges in cloud computing environments is the efficient utilization of resources. The growing demand for computational power requires the creation of algorithms that can not only optimally allocate the load among microservices but also maximize the use of server and channel capacities. This article discusses algorithms and chooses the most fitting ones that help achieve these goals.

The idea of the article is to describe in details the algorithm of finding complementary microservices that, when combined, they form a common close to maximal load on a server group. This can be achieved by different techniques of division and combining the microservices having the channel load, processor load and memory usage statistics as input characteristics. The algorithm details are described in details and with examples in the article. In particular, Z-Scale, K-Means and FastDTW are well-known algorithms combined with the author ideas are used to receive final matches.

One of the necessary aspects to consider is to correctly perform transition of the initial profiled microservice data to the realities of the cloud system. Yet, if something goes very wrong, there is a stage of regrouping using the cloud profiled data during which a better match will be found.

Using the technique of finding compliments, load balancing is still needed with the difference that it will happen more rarely, and it will be predicted.

Keywords

Complimenting microservices, cloud computing, microservice clusterization, energy efficiency, optimal resource load

1. Introduction

Efficient resource utilization is one of the key challenges in modern cloud computing environments. The increasing demand for computational resources necessitates the development of algorithms that can not only optimally distribute the load among microservices but also ensure full utilization of server and channel capacities.

It is not only the cloud system that matters but also the style the software is developed. In the dynamic landscape of modern software development, microservices architecture has emerged as a pivotal approach, enabling scalable and maintainable systems. As microservices proliferate, optimizing their interactions becomes paramount. This paper explores various algorithms designed to determine the similarities and complementarities between microservices based on their operational methodologies and load amplitudes. By identifying the most effective algorithm, we aim to facilitate the grouping of complementary microservices, thereby enhancing system efficiency and resilience. Furthermore, we delve into the conceptual framework of complementing microservices, offering a detailed exposition of its principles and practical implications.

Additionally, this study presents visual representations of different methods of microservice complementation. These images illustrate the comparative effectiveness of various algorithms in

[1] *14th International Scientific and Practical Conference from Programming UkrPROG'2024, May 14-15, 2024, Kyiv, Ukraine*

* Corresponding author.

† These authors contributed equally.

o_dmytrenko@lll.kpi.ua (O. Dmytrenko); mskulysh@gmail.com (M. Skulysh) lgloba@its.kpi.ua (L. Globa)
0009-0009-4785-4226 (O. Dmytrenko); 0000-0002-5141-1382 (M. Skulysh); 0000-0003-3231-3012 (L. Globa)

identifying and grouping complementary microservices. By analyzing these visual aids, readers can gain a deeper understanding of the problem and better imagine the boundaries of different sets of microservice input data, providing a comprehensive overview of the potential strategies for optimizing microservice interactions.

2. Idea of a Search for a Microservice Compliment

Cloud providers are constantly seeking ways to optimize resource usage. Efficient resource usage not only reduces operational costs but also minimizes the carbon footprint, contributing to more sustainable development. The environmental impact of IT operations is becoming increasingly significant, with the energy consumption of AI technologies notably higher than that of traditional search engines. According to recent studies, the energy required for AI operations can be several times greater than for standard computational tasks [1]. Therefore, optimizing resource allocation and utilization in cloud systems is critical for both economic and environmental reasons.

Ensuring the stable operation of all elements within a cloud system often involves updating servers once their warranty periods expire. Thus, maximizing the utilization of these servers is essential to fully leverage their resources. This paper proposes an overlook on algorithms and choice of the best one that designed to help cloud systems not only efficiently distribute workloads among microservices and orchestrate the initiation or cessation of additional services, but also to maximize server and bandwidth utilization.

When applications run on servers, the servers are typically not fully loaded. The degree of loading depends on the application's purpose, the time of day, and the day of the week. Since loading is a predictable characteristic, this knowledge can be utilized to form server groups or allocate server and bandwidth capacities to host multiple microservices, which collectively approach full load capacity.

Regarding microservice architecture, it is important to recognize that not all software products require this approach or the associated serverless architecture. Monolithic architecture may be more efficient, particularly for smaller applications or those with evenly distributed loads. In such cases, the monolithic application can be treated similarly to a large microservice without possibility to divide them, thus simplifying the problem to that described previously [2], [3].

Microservice architecture has gained immense popularity for large applications with distinct functionalities that are not always tightly coupled with other subsystems. Users of such applications, e.g. an online store, may spend considerable time browsing or making selections while spending less time on other actions like login and payment. Microservices enable the separation of these parts, allowing additional instances to be launched for only the components that require it during high load periods. This method already saves business owners' money, as starting an additional application and distributing the load between them (load balancing) is more efficient than supporting two or more instances all the time [4].

2.1. General Load Distribution Algorithm

The load distribution algorithm we propose aims to optimize resource utilization in cloud computing environments by effectively managing and distributing workloads. This algorithm is designed to predict load patterns, dynamically orchestrate microservices, and balance the load across servers to prevent bottlenecks and underutilization. The algorithm consists of the following stages:

2.1.1. Load Prediction

The first step involves analyzing historical data to forecast future load patterns. By identifying trends and peak usage times, the algorithm can anticipate high and low demand periods. This predictive capability allows the system to prepare for varying loads, ensuring that resources are allocated efficiently.

2.1.2. Server Grouping

The purpose of the server grouping is to strive for maximal utilization of server and channel capacities. The algorithm also focuses on maximizing the utilization of server and channel capacities. By combining multiple microprocessors on a single server, the system can achieve a balanced load distribution. Additionally, optimization techniques, such as data compression and efficient use of communication channels, are employed to ensure that channel resources are used to their full potential.

Based on the predicted load of computing, channel, and RAM resources, the algorithm forms server groups or volumetric servers. These groups are designed to handle the anticipated load, maximizing resource utilization. By clustering servers according to expected demand, the system can create configurations that prevent overloading and underutilization, ensuring a balanced distribution of resources. This part is the one developed in the article further. The ideas and algorithms of how to make the serve grouping are discussed.

2.1.3. Microservices Orchestration

Dynamic management of microservices is crucial for responding to changing loads. The algorithm orchestrates the deployment and shutdown of microservices based on real-time demand. During peak periods, additional microservices are deployed to handle the increased load. The new instances can have the state of cold or hot load, meaning that they are prepared. This is possible with having predicted the time of load increase [5]. Conversely, during off-peak times, unnecessary microservices are deactivated to save resources. This dynamic approach ensures that the system remains responsive and efficient [6], [7].

2.1.4. Load Balancing

To prevent any single server from becoming a bottleneck, the algorithm employs load balancing techniques, the same as on cloud systems or Kubernetes but for the groups. Requests are evenly distributed across servers, ensuring a smooth and efficient operation. Load balancing helps in maintaining system stability and performance by preventing overloading of individual servers and ensuring that all resources are utilized effectively [8], [9], [10].

2.1.5. Server Regrouping

During runtime, the algorithm continuously monitors performance metrics to evaluate resource usage efficiency. If inefficiencies are detected, the algorithm initiates a server regrouping process. This involves reassigning resources and reorganizing server groups to better match current load conditions. By dynamically adjusting server group configurations, the system can improve resource utilization and maintain optimal performance.

2.2. Server Grouping Steps

Typically, when applications run on a server, resources are not used to their full potential. This can be due to various factors such as the application's specifications, time of day, or day of the week. For example, during working hours, servers may be more loaded than during nighttime or weekends. An important characteristic of the load is its predictability, which allows optimizing the distribution of resources.

The key effective management and distribution of workloads, our proposed load distribution algorithm incorporates mechanisms to identify similar or complementing microservices. This capability enhances the system's efficiency by ensuring that microservices are grouped and orchestrated in a manner that maximizes their synergistic potential. Below the key stages of this process are depicted.

2.2.1. Microservice Profiling

The first step in identifying similar or complementing microservices is to profile each microservice within the system. Profiling involves collecting detailed metadata about each microservice, including its computational requirements, typical load patterns, resource consumption, and functional dependencies. This metadata provides a comprehensive view of how each microservice operates. Based on this information conclusion on how to organise interaction of this microservice with others will be made.

In order to effectively manage and distribute workloads in cloud computing environments, it is essential to accurately measure processor load. Processor load can be quantified by considering

1. the number of cores,
2. core capacity, and
3. the current load on each core.

There are more details on how it is performed in the section Processor Load. Below is the example of a table with the characteristics mentioned above. The primary qualities of each resource should also be included, as far as the similar should be compared with the similar.

Table 1. Example of General Microservice Profiling

Characteristics name	Peculiarities	Value
Channel Throughput	10 Мбіт/с (ethernet)	8 Mb/s
δ_{Channel}	Channel load daily variations	±2%
Processor (CPU) load	2,5 GHz Quad-Core Intel Core i7	75%
Cores	Quad-Core	4
δ_{CPU}	CPU load daily variations	±8%
RAM	8 GB 1600 MHz DDR3	4 Gb
δ_{RAM}	RAM daily variations	±7%
Hard Drive (HD)	Seagate BarraCuda HDD	60 Gb
Can be chunked (canChunk)	Needed if requirements are too high	yes

2.2.2. Similarity Analysis

Using the collected profiles, the algorithm performs a similarity analysis to identify microservices with comparable characteristics. Techniques such as clustering, correlation analysis, and pattern recognition are employed to group microservices with similar load patterns, resource usage, and operational behaviors. This grouping helps in forming server clusters that can handle homogeneous workloads more efficiently.

2.2.3. Complementarity Analysis

In addition to finding similar microservices, the algorithm also identifies complementing microservices—those that, when deployed together, enhance overall system performance. Complementarity analysis involves evaluating the functional dependencies and resource usage patterns to determine which microservices work well together. For example, a microservice with high CPU usage but low memory consumption might complement another microservice with the opposite resource usage pattern.

2.2.4. Dynamic Grouping

Based on the results of similarity and complementarity analyses, the algorithm dynamically forms groups of microservices that are either similar or complementing. These groups are then deployed on the same server or server cluster to optimize resource utilization. By ensuring that similar microservices share resources and complementing microservices balance each other's load, the system can achieve higher efficiency and stability.

2.2.5. Continuous Monitoring and Adjustment

The system continuously monitors the performance of grouped microservices to ensure that the initial grouping remains optimal. If performance metrics indicate suboptimal resource utilization or increased latency, the algorithm re-evaluates the grouping and makes necessary adjustments. This dynamic re-grouping process ensures that the system adapts to changing conditions and maintains high performance.

3. Primary Characteristics and Cloud Environment Specifics

The primary characteristics of microservices, such as processor load, memory usage, and network demand, are initially profiled based on the environment in which they are developed and tested. However, these environments often differ from the actual cloud infrastructure where the microservices will eventually be deployed. This disparity can lead to variations in performance metrics, as cloud providers often optimize their services to work efficiently within their own ecosystems.

Consider Amazon DynamoDB as an example. DynamoDB is a fully managed NoSQL database service offered by Amazon Web Services (AWS). It is designed to provide fast and predictable performance with seamless scalability. The architecture of DynamoDB is optimized for AWS infrastructure, offering enhanced performance characteristics compared to other NoSQL databases that might be used during the development phase. In the sources there is information about comparison of DynamoDB and some other DBs [11], [12], [13].

Developers might initially profile their microservices using a different NoSQL database, such as MongoDB or Cassandra, on local or non-AWS infrastructure. When these microservices are migrated to AWS and start using DynamoDB, they may exhibit different performance characteristics. For instance, DynamoDB might offer lower latency and higher throughput due to AWS-specific optimizations, leading to more efficient resource utilization [14], [15].

Despite these differences, the initial statistics provided by the developer team should still serve as a reasonable baseline for the cloud deployment. These statistics include:

1. Processor Load: the percentage of CPU capacity used by the microservice.

2. Memory Usage: the amount of RAM required for optimal performance.
3. Network Demand: the bandwidth and latency requirements for communication.

While these initial metrics might not perfectly match the cloud environment's performance, they provide a starting point. To manage this transition smoothly, it is crucial to set boundaries based on these initial statistics. These boundaries act as guardrails, ensuring that the microservices operate within acceptable limits once deployed in the cloud.

3.1. RAM Usage

Profiling based on RAM usage involves monitoring the memory consumption patterns of each microservice. Some microservices might require large amounts of memory for caching, data processing, or other operations, while others may have minimal memory needs. When a machine has less RAM than required by the application, it may lead to frequent paging or swapping. This process involves moving data between RAM and disk storage, significantly slowing down the application [16]. This should be prevented as much as possible.

Sufficient or excess RAM ensures that the application data and processes are kept in memory, leading to faster access times and better overall performance. More RAM allows for better support of concurrent processes and threads, enabling higher levels of parallelism and efficient multitasking. Applications that rely heavily on in-memory data processing (e.g., databases, large data analytics tasks) will benefit more from increased RAM which should be the case on large shared memory space in a cloud environment.

Use cloud services that support dynamic resource allocation [17], allowing applications to request additional memory as needed. Services like AWS Auto Scaling and Azure Virtual Machine Scale Sets can help manage resources efficiently. Yet, using them the user should count on horizontal scaling, meaning opening new instances [18]. Well-known cloud providers do not propose automatic vertical scaling, which is adding only RAM without other additional resources or remaking of VM [19].

AWS proposes to use different instance types that have more RAM within predefined in configurations Auto Scaling groups by defining them in the launch template. Azure Automation proposes to create scripts that monitor performance metrics and resize VMs when certain thresholds are met. This approach requires a complex setup and management strategy. Google Cloud Platform works similarly through stopping the virtual machine, reallocation memory resource and starting the VM again.

By categorizing microservices according to their RAM usage, the algorithm can identify which microservices can be co-located on the same server without causing memory contention. Developers should perform load testing on different machine configurations to understand how varying RAM capacities affect performance in order to correctly state the requirements to the software. This helps in making informed decisions about memory allocation and preventing issues related to memory overcommitment or underutilization.

3.2. Channel Load

Channel load profiling involves assessing the network and I/O demands of microservices. This includes monitoring the bandwidth usage, latency sensitivity, and data transfer rates required by each microservice. Similarly to any other characteristic, some microservices need high network bandwidth for data-intensive operations, for example video or other large files transfer, while others might be more sensitive to latency and require low-latency communication channels, for example batch transfer.

The physical distance between the data source and destination affects latency and bandwidth usage. Distributed systems may experience higher channel loads due to long-distance data transfers [9]. The number of concurrent users or devices accessing the network impacts channel load. High concurrency can lead to congestion and reduced performance. Shared network resources can become bottlenecks when multiple applications or services compete for the same bandwidth [8], [20]. Traffic with sudden spikes in data transmission can strain network channels, causing temporary congestion and packet loss [8], [20]. These are the issues that may cause serious troubles, and the cloud environment helps to treat many of those in particular with load balancing and stable connections between its global network of data centres. In case of IoT edge computing comes into turn. Edge computing brings computation and data storage closer to the location where it is needed to improve response times and save bandwidth [9], [21], [22]. Cloud providers offer edge computing solutions that enable processing data closer to the end-users or IoT devices.

It is easier to manage and optimize the applications with a steady, predictable data flow. By understanding the metrics of channel load, its throughput, latencies, possible package loss and jitter, by running load tests on channels with different throughput it is possible to understand the requirements for the channel and make statistics of usage. Using of these requirements at each moment of time, the algorithm can group microservices in a way that optimizes network resource usage. Several microservices grouped together, can make one more predictable system and at the same time receive more resources for the channel. This will provoke reduce in network congestion, improve data transfer efficiency, and ensure that latency-sensitive microservices will operate smoothly.

3.3. Processor Load

Processor load profiling focuses on the CPU usage patterns of microservices. Some microservices might perform CPU-intensive tasks, such as data encryption, complex calculations, or real-time processing, leading to high CPU load. Others might have lower CPU demands. By analyzing the CPU load characteristics, the algorithm can group microservices that either have similar CPU usage patterns or can complement each other by balancing high and low CPU loads on the same server. This ensures efficient CPU utilization and avoids scenarios where some microservices starve for CPU time while others leave the CPU underutilized.

Modern processors have multiple cores, each capable of handling separate tasks simultaneously. The total processing capability of a server is directly related to the number of cores it has. Depending on the program, it might be important to give a microservice a certain amount of cores. This information is clear only to the developers because they are the ones who could use special algorithms of parallelization that are effective for a certain number of cores, e.g. to 2, 3, 4 or any multiplier of 2, 3, 4 correspondingly.

Each core has a specific capacity, typically measured in GHz. The core capacity determines how many instructions per second a core can process. The current load on each core is usually expressed as a percentage of its total capacity. This load represents the proportion of time the core is actively processing tasks versus being idle. Time, for which this information is taken, is also important. Some microprocessors might have random load because tasks are of different quality and come in a random time. Still, the idea of microprocessors is to solve a defined type of tasks and so usually the regularity in time of processor load is followed if not from day to day than from week to week.

Transforming these three factors into a single, representative digit can simplify the process of resource allocation and optimization. Yet, computers on which the software is tested can be of a different quality than the ones present in the server rooms of a cloud provider. The brand can be neglected, but the power is what should be extracted and normalized to the possibilities of the cloud environment. Later, on the iteration of 2.1.5. Server Regrouping the microservice could be replaced if its initial characteristics were estimated very wrongly. The steps are below.

3.3.1. Calculation of Processor Load

To calculate the overall processor load and transform it into a single digit, the following steps are needed [23]. For each core, the load can be calculated using the formula:

$$\text{Load per Core} = \left(\frac{\text{Current Load (\%)}}{100}\right) \times \text{Core Capacity (GHz)} \tag{1}$$

The total load for the processor is the sum of the loads for all cores:

$$\text{Total Load} = \sum_{i=1}^{N}\left(\left(\frac{\text{Load}_i(\%)}{100}\right) \times \text{Capacity}_i(GHz)\right) \tag{2}$$

where N is the number of cores, Load_i is the load of core i, and Capacity_i is the capacity of core i.

To transform the total load into a single digit, it is necessary to normalize it relative to the maximum possible load. The maximum possible load occurs when all cores are at 100% capacity.

Therefore:

$$\text{Max Possible Load} = \sum_{i=1}^{N} \text{Capacity}_i(GHz) \tag{3}$$

The normalized load is then:

$$\text{Normalized Load} = \left(\frac{\text{Total Load}}{\text{Max Possible Load}}\right) \times 10 \tag{4}$$

3.3.2. Example of Processor Load Calculation

Let us assume a processor with

- 4 cores, each with a capacity of
- 2.5 GHz, and
- current loads of 60%, 75%, 50%, and 90% respectively.

Calculate the load per core using (1):

$$Core1 : \left(\frac{60}{100}\right) \times 2.5 = 1.5GHz$$
$$Core2 : \left(\frac{75}{100}\right) \times 2.5 = 1.875GHz$$
$$Core3 : \left(\frac{50}{100}\right) \times 2.5 = 1.25GHz$$
$$Core4 : \left(\frac{90}{100}\right) \times 2.5 = 2.25GHz$$

Summing the loads of all cores using (2)

$$\text{Total Load} = 1.5 + 1.875 + 1.25 + 2.25 = 6.875 \text{ GHz}$$

Calculate the maximum possible load using (3)

$$\text{Max Possible Load} = 4 \times 2.5 = 10 \text{ GHz}$$

Normalize the total load using (4)

$$\text{Normalized Load} = \left(\frac{6.875}{10}\right) \times 10 = 6.875$$

The normalized load, transformed into a single digit, is approximately 6.88. This value can be rounded to one decimal place, making it 6.9.

3.3.3. Transforming the Load Percentage Calculation to the New Environment

When performing transition of the values counted on developer environment to the cloud environment, several complexities should be taken into consideration. Here are the factors to consider.

1. Inter-core communication overhead - more cores may lead to increased overhead for managing communication between cores, particularly if the microservices require frequent synchronization or data sharing.
2. Task scheduling inefficiencies - distributing tasks across a larger number of cores can introduce inefficiencies in task scheduling, potentially leading to underutilization of some cores.

Due to the reasons mentioned above, as the number of cores increases, the performance gain from adding each additional core might diminish. To incorporate these factors, we can introduce an adjustment coefficient α that accounts for the overhead of distributing tasks across more cores. This coefficient can be derived empirically or estimated based on typical system behaviour. This should be done based on cloud statistics with their specific environment. Adjustment to the cloud needs by itself is changing the base of the processor.

The formula for the adjusted load percentage per core becomes:

$$\text{Adjusted Load Percentage per Core} = \frac{\text{Total Required Load}}{\text{Max Possible Cloud Load}} \times (100 + \alpha) \quad (5)$$

Taking an adjustment coefficient $\alpha = 0.1 or 10\%$ multiplies the unadjusted load percentage by 110% to account for the inefficiencies.

Let us apply the adjustment factor mentioned in (5) to our example described in 4.1.5. Example of Processor Load Calculation transporting the computations onto cloud environment to calculate the adjusted load percentage.

Given a server group with the following characteristics:

Total Required Load: 6.875 GHz

Number of Cores: 8

Core Capacity: 3.0 GHz each

Adjustment Factor: 0.1 or 10% (together with 100% gives 110%)

Calculate the total capacity of the cloud environment called Max Possible Cloud Load, we use formula (3):

$$\text{Max Possible Cloud Load} = 8 \times 3.0 = 24 \text{ GHz}$$

The adjusted load percentage per core is calculated as follows:

$$\text{Adjusted Load Percentage per Core} = \left(\frac{6.875}{24}\right) \times 110 = 28.65 \times 1.1 = 31.515\%$$

Summing it up, the initial value of 68.7% of total general load should be replaced by 31.5% of total load on a cloud server. A small percentage of adjustment to the new environment allows us to take into account the additional overhead and inefficiencies when distributing tasks across more cores. This adjusted load percentage provides a more accurate reflection of the actual processor load in the cloud environment. 10% in the example is a pretty big percentage and is taken only for comfortable counting. In practice, it should be less.

4. Organizing and Matching Microservices Algorithm

Efficient management of resources in cloud computing involves several steps of microservice preparation, grouping based on resource usage patterns, and after searching for complementary microservices within these groups. When a group is formed, it can be deployed. This approach enhances performance and resource utilization. Additionally, microservices with high requirements should be divided into smaller parts if possible. In this paragraph, there is a detailed explanation of the process.

4.1. Pre-steps - preparing data for the algorithm

These steps are cloud-specific and have to be based on the hardware the cloud provider has. The idea is to transform the metrics collected by the developer team to metrics relevant to the resources present in the server rooms. Normally the majority of the servers are similar or the same, so this makes the task easier.

4.1.1. Characteristics’ Adoption to Cloud Infrastructure

To ensure the input data is correct and corresponds to the cloud environment, all the metrics has to be divided by the corresponding load characteristic of a standard server of the cloud environment, taking into consideration possible error or coefficient of adjustment. Formula (5) corresponds to this action. The steps are the following:

1. Collect the resource usage data (CPU, RAM, Channel) from the microservices.
2. Change the base for the profiled data (normalize) setting on the characteristics of the cloud servers using formula (5).

4.1.2. Dividing High-Requirement Microservices

Before grouping and searching for similar and complementary microservices, it is necessary to filter out the microservices that might not fit. These are the microservices with extremely high resource requirements. Let us mark $P_{\mathrm{t}_{\min}}$ as a moment of time when load reaches its maximal value. Then criteria for the division looks like the following:

$$P_{\mathrm{t}_{\max}} + \delta \geq P_{\mathrm{boundary}} \tag{6}$$

These microservices should be divided into smaller parts if they can be chunked. Possibly this is a monolith application, because an idea of microservices has to do with usually small tasks that are separated from others [24], so they should not consume loads of any space. More information on dividing microservices and using boundaries can be found in [25]. This division can help to predict the time when microservices should be started and prepare additional servers, installing pre-start data and copping last actual information onto them, e.g. cold, warm or hot standby state [5].

When decomposing a monolithic application into smaller components, it's advisable to base the segmentation on the usual or average load that the system handles. For periods when the load exceeds the average, it is efficient to provision additional instances of the monolith that are configured to handle the increased demand. This strategy ensures that the system remains responsive and stable during peak usage times, while optimizing resource utilization during normal operations. Please see Figure 1 for visualization.

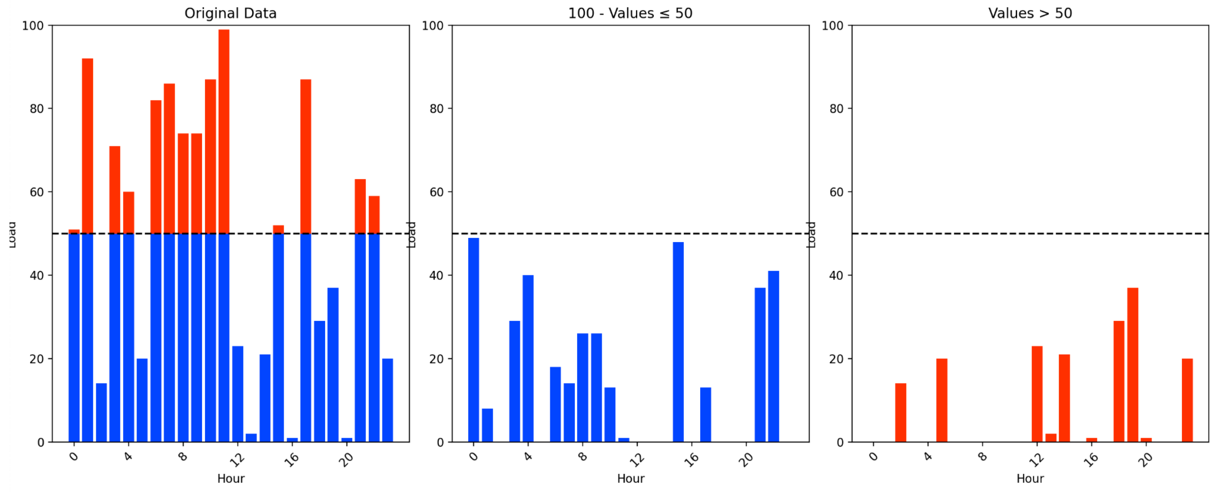

Figure 1: Division of highly loaded microservice onto 2 instances.

There are situations where creating additional instances of an application may not be feasible or beneficial. For instance, if the application begins to receive unusually large messages sequentially—rather than small, frequent ones—this could lead to overloading of the channel resources. Such scenarios, which may only be evident to the business analysts and developers familiar with the operational context, should prompt these stakeholders to set the `canChunk` attribute to `false` in Table 1, indicating that chunking is not advisable. This should be considered an exceptional case. Alternatively, it may be possible to redesign the application to manage data transfer more effectively, thereby alleviating the need for additional instances.

For this part of the algorithm, there will be the following input values:

- "canChunk" - this characteristic indicates whether a microservice can be divided into smaller, more manageable parts,

and maximal possible values based on which the cut-off of excessive loads will occur:

- $P_{\text{boundary, RAM}}$
- $P_{\text{boundary, CPU}}$
- $P_{\text{boundary, Channel}}$

The division algorithm is the following:

1. Identify microservices with resource requirements exceeding $P_{\text{boundary, RAM}}$, $P_{\text{boundary, CPU}}$ or $P_{\text{boundary, Channel}}$.
2. Check the "canChunk" characteristic.
3. If (6) is met, Divide High-Requirement Microservices (description and example are below).
4. Form a separate group of 'High-Requirement Microservices' for those that cannot be chunked but exceed the boundaries. Elements from this group will take separate large resources and won't participate in the following algorithm.

Division, mentioned in the 3rd point, happens if a microservice or a monolith can and should be chunked, it gets divided into smaller parts with resource requirements below the maximum thresholds. This division is performed only when a specific characteristic exceeds its minimal value by a certain percentage, known as "Δ" (Delta), and their sum is below the threshold. The Δ represents the additional percentage that should be added to the minimal value of the characteristic causing the division.

Let P_{tmin} be the minimal threshold for a characteristic, Δ be the percentage increase, ranging from 0% to 100%, and P_{boundary} be the maximum threshold. This formula should also include a small percentage of variations δ that may occur daily. This parameter is provided together with the initial metrics. The algorithm works for all the criteria in the same way, so formulas and examples are written without specifying for which exact characteristic they are taken. The division should occur if the characteristic exceeds P_{boundary}:

$$P_{\text{chunk}} = P_{t_{\min}} + \Delta P_{t_{\min}} + \delta < P_{\text{boundary}} \qquad (7)$$

where:

- P_{chunk} is the threshold for initiating the division.
- $P_{t_{\min}}$ is the moment with minimal value for a certain characteristic.
- Δ is the percentage increase required for division.
- P_{boundary} is the maximum threshold value.

For example, if Δ = 30%, δ = 5%, P_{tmin} = 10, and P_{boundary} = 20. Placing values in the formula, we receive the following:

$$P_{\text{chunk}} = 10 + 0.30 \times 10 + 0.05 \times 10 = 10 + 3 + 0.5 = 13.5$$

Since 13.5 is less than the boundary value of 20, the microservice should be divided if the characteristic exceeds 13.5 units. This ensures that microservices are efficiently divided only when necessary, optimizing resource usage in the cloud environment. The division will still happen by boundary value rather than the Δ percentage.

To determine the number of parts m into which a microservice should be divided, we use the formula (8). This formula takes into account the sum of the maximum load P_{max} and a small percentage to account for daily variations δ, dividing this sum by the boundary value P_{boundary}, and rounding to the nearest integer which is marked by the floor-ceiling function $\lfloor \cdot \rceil$.

$$m = \left\lfloor \frac{P_{\max} + \delta}{P_{\text{boundary}}} \right\rceil \tag{8}$$

The resulting last microprocessor load might be too small and finally neglected. The criteria should be cloud and other calculations specific. We propose to neglect creating the last unit if the remainder does not exceed δ. This parameter will be used in the other formulas and should be taken into consideration when counting the total group load. In formulas, it looks as follows:

$$\text{if } \left(\left\lceil \frac{P_{\max} + \delta}{P_{\text{boundary}}} \right\rceil - \frac{P_{\max} + \delta}{P_{\text{boundary}}} \right) \leq \delta \text{ then } m = \left\lfloor \frac{P_{\max} + \delta}{P_{\text{boundary}}} \right\rfloor \text{ otherwise } m = \left\lceil \frac{P_{\max} + \delta}{P_{\text{boundary}}} \right\rceil \tag{9}$$

If $P_{\max}$ is 50, δ is 8% or 0.08 which makes 50 * 0.08 = 4, and P_{boundary} is 20, the results of the application of (8) and (9) are calculated as following:

$$m = \left\lfloor \frac{50+4}{20} \right\rceil = \left\lfloor \frac{54}{20} \right\rceil = \lfloor 2.7 \rceil \mid \text{as } 3 - 2.7 = 0.7 \text{ which is < 0.08, then we round up} \mid = 3$$

4.2 Search for the Similarity Groups

Interest in finding similarity groups is related to the faster search for complementary microservices. Yet, this step might not be necessary as it involves an additional check of all vectors, so if a huge amount of data needs to be processed, it might be omitted.

Grouping by similar behaviour and, further, by similar amplitudes will make the search for complements faster and more efficient. If processing the whole big initial array of data, it might be useful to agree on the first more or less okay match using a greedy algorithm. However, having a group of similar ones will make it possible to find the best match. That is why this step might be used not when starting the practice of complementary microservices for the first time on a large scope, but sooner for the regrouping phase for the ones, for which the match wasn’t found perfectly.

The similarity should be searched only after normalization of vectors. This is the different normalization from the one in the pre-steps described in “Characteristics’ Adoption to Cloud Infrastructure”. Here, the idea is to put all the characteristics of the microservices onto the same scale rather than adopt them to the cloud potential. This normalization will be used only for searching similar behaviour and clusterization, which will be the next step, rather than for the resulting match search. So the plan is the following:

Normalization of amplitudes to see the same usage pattern.

Clusterization on groups of the similar behavior.

4.2.1. ormalization Using Z-Score

The performance pattern serves as the primary criterion for identifying similar microservices. There are 3 points to consider during similarities computation, described in the third section “Primary Characteristics and Cloud Environment Specifics”. Their statistical data could be compared separately or concatenated into one array, which will be of the same size for every microservice. We propose collecting microservice usage statistics every hour, so having 3 characteristics together will make 3 by 24 which is 72 places in the resulting array.

For more detailed clusterization which might be meaningful having given a large set of microservices, a week data might be combined. As a smaller and faster alternative, a combination of a working day and a weekend day may be considered. When having a few microservices, feeding a high-detailed set of input data into the algorithm provokes a risk of too high clusterization level. This can lead to too many clusters to check for matches, reducing the quality of complementarity as additional characteristics might be treated as noise. Consequently, the threshold for defining complementarity would need to be lowered.

Z-score normalization addresses the issue of differing amplitudes while preserving the inherent performance patterns. This approach is more suitable than min-max normalization, which ensures all features have the same scale but does not handle outliers well [26]. In our scenario, outliers—very distant values—may exist, and our focus is on identifying the underlying patterns rather than strictly maintaining scale. Hence, min-max normalization is not appropriate in this context.

Normalization using the Z-score allows data to be scaled to a common scale, which is particularly useful for comparing vectors and visualizing differences in the behavior. This is done using the mean and standard deviation of the vector elements. The formula for normalizing a vector is:

$$Z_i = \frac{X_i - \mu}{\sigma} \tag{10}$$

Where:
- Z_i is the normalized value for the i-th element.
- X_i is the i-th element of the original vector.
- μ is the mean of the vector elements.
- σ is the standard deviation of the vector elements. This is a measure of the amount of variation or dispersion in a set of values.

The mean μ and standard deviation σ for the vector X are calculated as:

$$\mu = \frac{1}{N}\sum_{i=1}^{N} X_i \quad , \qquad \sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(X_i - \mu)^2} \qquad (11), (12)$$

Where:
- N is the number of elements in the vector.

4.2.2. Vector Difference with FastDTW Algorithm

To count the strength of similarity between vectors and visualize it, the Dynamic Time Warping (DTW) algorithm can be used. It is designed to find the optimal alignment between two time series, making it useful for determining time series similarity, classification, and identifying corresponding regions between series. However, DTW has quadratic time and space complexity, limiting its application to small datasets. FastDTW, an approximation of DTW, addresses this limitation by providing linear time and space complexity. FastDTW employs a multilevel approach, recursively projecting a solution from a coarse resolution and refining it, enabling efficient processing of larger time series datasets [27].

The goal is to find the path that minimizes the total cumulative distance. Here are the key steps involved in the FastDTW algorithm:

1. Coarsening - reduce of the resolution of the sequences by downsampling. This involves creating a lower-resolution version of the sequences. E.g. every second number in the vector is taken.
2. Recursive calculation of path at the coarsened resolution. This step finds an approximate path at the lower resolution.
3. Projection of the path from the coarsened resolution back to the original resolution.
4. Refinement of the path at the higher resolution using a local search around the projected path. This involves adjusting the path to minimize the distance further.

The DTW distance D between two sequences X and Y is calculated using:

$$D(X,Y) = \min\left\{\sum_{k=1}^{K} d(x_{i_k}, y_{j_k})\right\} \tag{13}$$

Where:
- $d(x_{i_k}, y_{j_k})$ is the distance between points x_{i_k} and y_{j_k}. It can be measured by euclidean algorithm or another chosen.
- K is the warping path length.

To demonstrate how the similarity between the statistics data from 2 microprocessors with time series can be counted and that using the Z-normalization will even the values even if they strongly differ but find the pattern, a graph is provided after processing the datasets below. The difference in their amplitudes is 2 times. But the time of being more and less active is similar. Please see Figure 2 for an image.

1: [10, 8, 15, 21, 25, 17, 13, 15, 11, 7, 12, 8, 7, 6, 5, 10, 12, 13, 22, 25, 15, 12, 14, 11]

2: [20, 24, 30, 40, 50, 36, 26, 24, 22, 20, 18, 16, 14, 12, 10, 20, 24, 30, 40, 50, 36, 26, 24, 22]

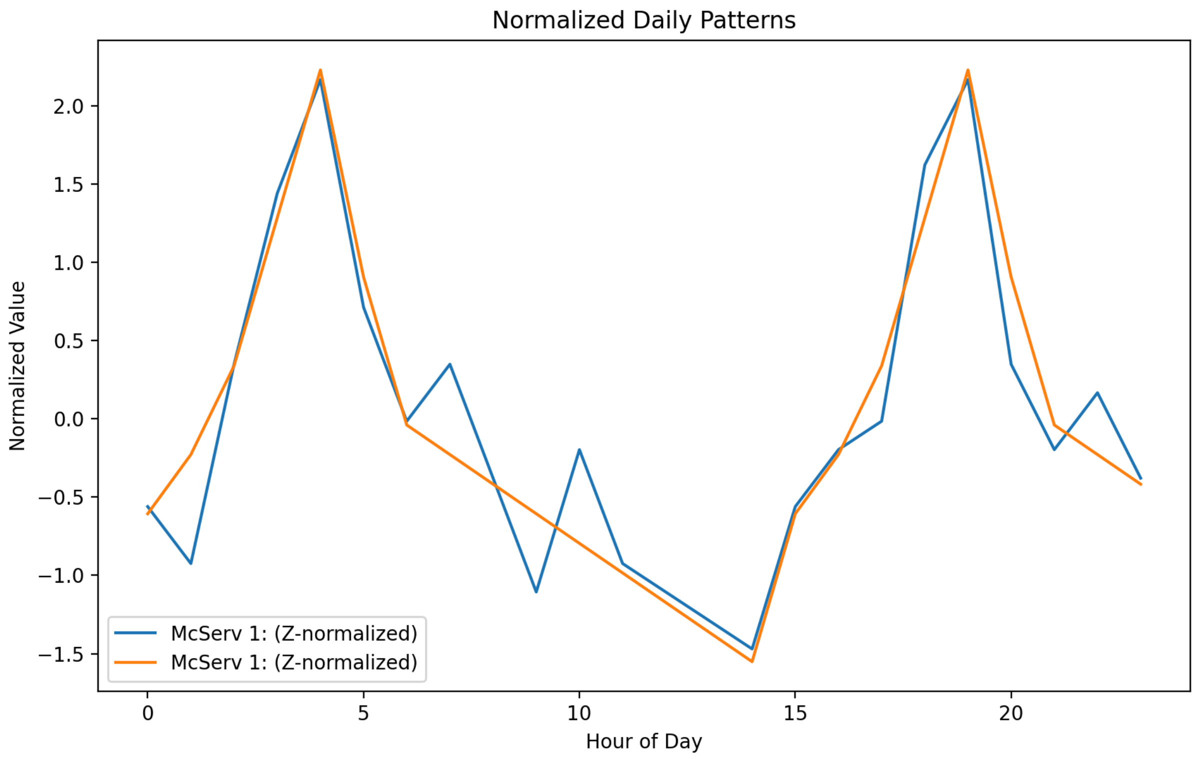

Figure 2: Normalization + FastDTW for time series which initial amplitudes 2 times differ.

4.2.3. Grouping based on Clustering Algorithms KMeans and DBSCAN

To group microservices by similar behavior, clustering algorithms such as KMeans and DBSCAN (Density-Based Spatial Clustering of Applications with Noise) can be employed. Objects in the same group called a cluster are more similar to each other than to those in other groups. KMeans and DBSCAN are two widely used clustering algorithms, each with its own peculiarities. This section provides a detailed comparison between these two methods, focusing on their core principles, advantages, limitations, and use cases.

DBSCAN algorithm identifies clusters based on the density of points. It can find arbitrarily shaped clusters and can handle noise. In the task, the data points are expected to form predominantly circular clusters, K-Means clustering emerges as the more suitable choice over DBSCAN. The following points describe the algorithms and highlight why K-Means is better fitted for this scenario:

1. The data points are expected in a radius of a circle. K-Means excels at identifying spherical or circular clusters, which aligns well with our expected data distribution. DBSCAN, while capable of detecting arbitrary shapes, may unnecessarily complicate the analysis for our circular clusters.
2. For the further search of the compliments the centroid points are needed as a base for which the compliment will be searched. K-Means produces explicit cluster centroids, which is valuable.
3. Outline handling is not expected as it should be filtered on the previous stage. DBSCAN's ability to identify noise points can be advantageous in some scenarios, however, the task requires including outline points in the clusters as they are. K-Means naturally incorporates

all points into clusters, which is beneficial since outlines are not to be treated as noise. This pre-processing step enhances K-Means' effectiveness for our specific task.

4. When initializing the new complimentary structure following the idea described in the paper, there might be need to treat plenty of microservices. That's why the computational efficiency is an important point to consider. K-Means generally offers better scalability for larger datasets. In scenarios where processing speed is crucial, K-Means can provide faster results, especially if the number of clusters is known or can be estimated.

KMeans clusters data into k groups by minimizing the variance within each cluster. The objective function is to find an argument - centroid with which the distance function to a point will have the minimal value. In other words, the closest centroid should attract a point to its cluster:

$$\arg\min_{S} \sum_{i=1}^{k} \sum_{x \in S_i} \|x - \mu_i\|^2 \tag{14}$$

Where:

- S is the set of clusters.

- μ_i is the centroid of cluster i.

- The algorithm iterates to update the centroids and reassign points to the nearest centroids until convergence.

There is also another method that could be considered - cosine similarity. It measures the cosine of the angle $\cos(\theta)$ between two vectors, indicating how similar they are regardless of magnitude. This method would work well of only the vectors were multidimensional in reality. The vectors in the problem are in fact time series. The angels in different parts of the same timeline are controversial. The cosine similarity method works rather with the line instead of a curve, so does not fit in this case.

4.2.4. Example of how grouping works

To make an example close to real, 24 microservices that accomplish each other were generated for each type of five behaviors. The behaviors with the number of generated microprocessors are presented below:

- 2 microservices with stable load for.
- 4 microservices with peak load during working hours (9 AM to 5 PM).
- 2 microservices with peaks after midnight and before 6 AM.
- 2 microservices with peaks from 5 PM till night.
- 2 microservices with random peaks.
- Complements for each of these patterns (load in the opposite time).

The mentioned above microservices went through the following steps:

1. Data preparation through normalization of microservices data through Z-Score method. This data has 24 dimensions (one for each hour of the day). An example can be seen in Figure 2.

2. Clusterization through KMeans algorithm. Principal component analysis (PCA) was used for vectors and centroids to make a clear and understandable image [28]. The algorithm is used only for graphical representation for better understanding of the KMeans algorithm quality of work, so it is not described in the paper. More information about it can be found here [29], [30]. It reduces the 24-dimensional data to 2 dimensions, what allows visualization of the high-dimensional data in a 2D plot. Yet, it is necessary to mention that some information is inevitably lost in the dimension reduction process.

On a scatter plot in Figure 3 each point represents a microservice. The centroids are plotted as large 'X' markers. Points of the same color form a cluster. Each point's x and y coordinates are the first and second principal components, respectively. This visualization allows observation of how well-separated the clusters are and if there are any clear groupings in the data.

Figure 4 shows 24-D vectors divided into clusters marked by certain colors. Dashed lines are the centroid vectors. The symmetry of an image is explained by the method of complementary vector generation. Addition of noise changes the picture. For the purpose of a more explicit and clear to understand example the “ideal” case is shown.

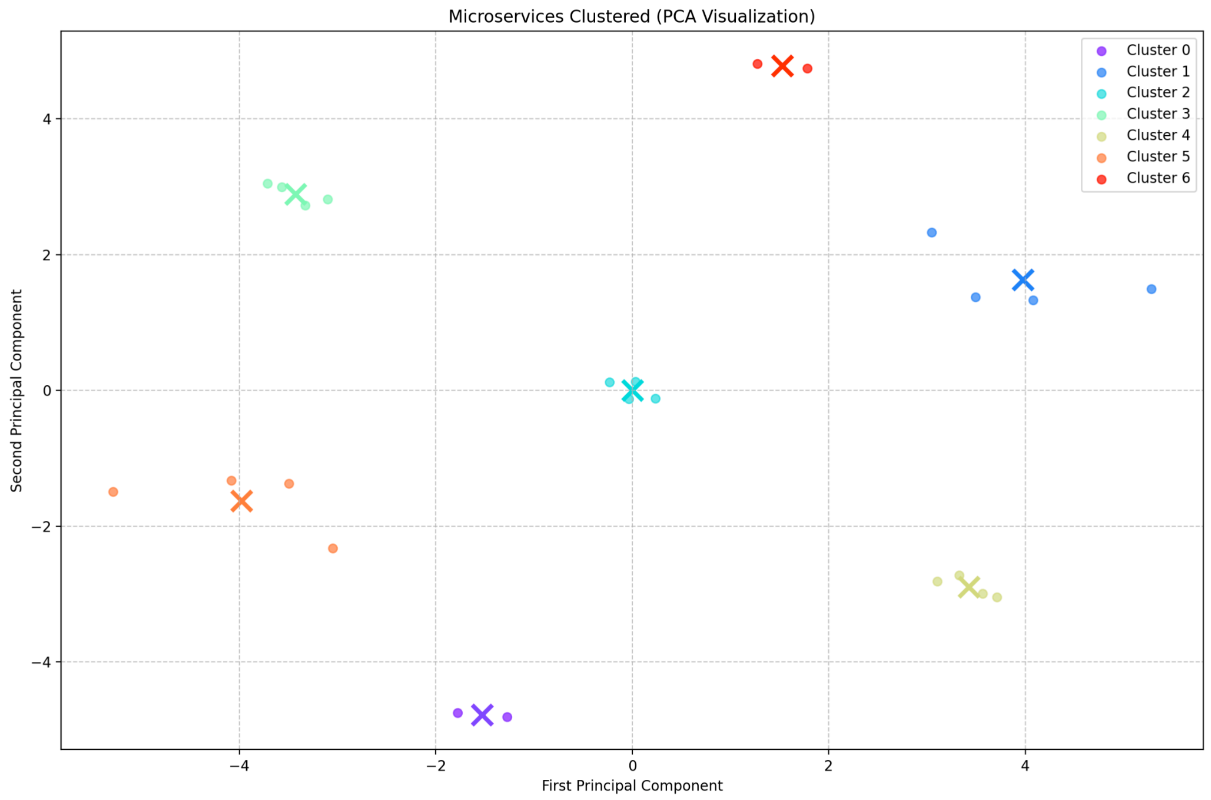

Figure 3: 2D vector-centroid visualization after KMeans algorithm after principal component analysis. X - centroids, point - 24-D vectors turned into 2-D vectors with PCA. The color indicates belonging to certain centroids.

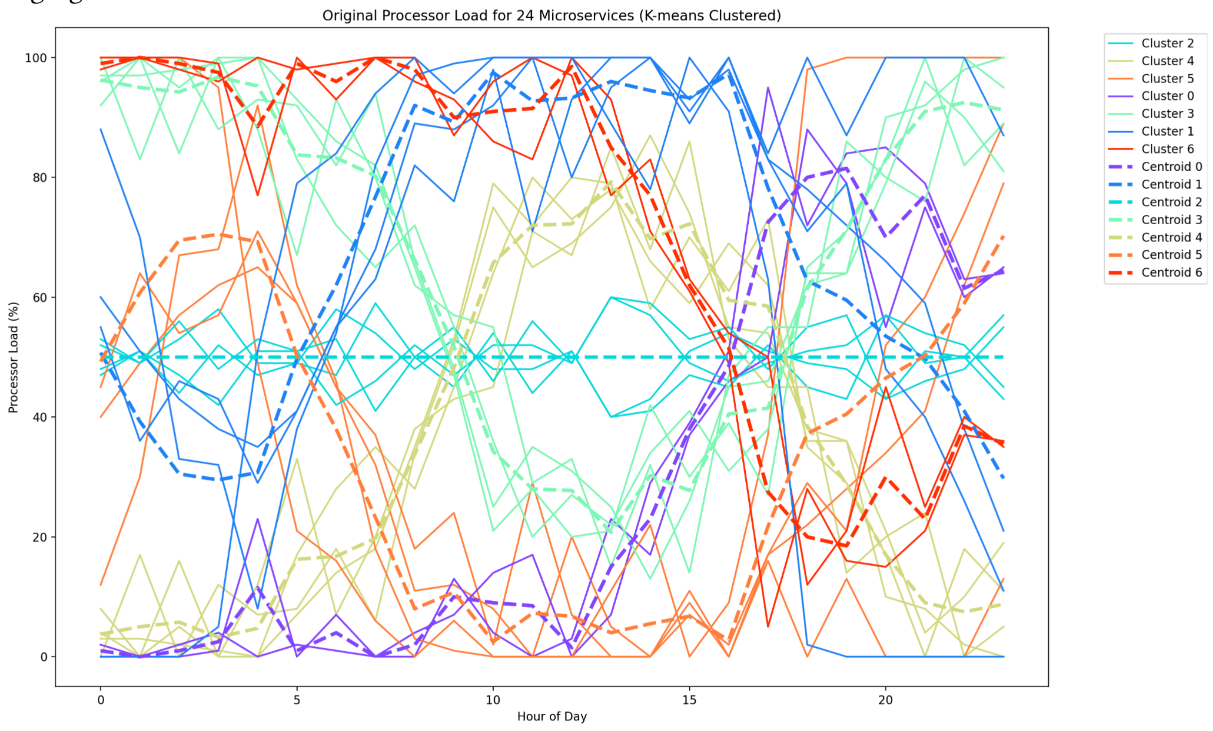

Figure 4: 24-D vectors divided into clusters which can be seen by the colour. Dashed lines are the centroid vectors. The symmetry of an image is explained by the method of complementary vector generation.

4.3 Search for Compliments

The idea of search of the compliments comes from the lattice theory. More information on this theme can be read in [31]. When having found the clusters of similarity, it will be faster to find the compliments matching the main characteristics of clusters instead of full enumeration of all the microservice vectors. Several methods might be used that are described in the following subsection.

4.3.1. Algorithm for a Complementary Microservice Search

1. Firstly, it is necessary to find the clusters which contain potentially accomplishing vectors. For each pair of clusters, including oneself, we calculate the discrepancy between their centroids to determine if they are complementary. Combination with oneself is necessary for the case of the even load during all the period of time. The discrepancy is defined as the sum of the differences between the maximal sum of corresponding vector elements sum_max and sum of the centroids, normalized by the maximal possible sum of those values sum_max. The formula for centroid discrepancy D_{centroid} is:

$$D_{\text{centroid}} = \frac{1}{n} \sum_{i=1}^{n} \left(\frac{\text{sum\_max} - (\text{vector1}_i + \text{vector2}_i)}{\text{sum\_max}} \right) \tag{15}$$

where n is the number of parameters in the vector, and vector1_i and vector2_i are the i-th parameters of the two centroids. A pair is considered complementary if $D_{\text{centroid}} \leq 30\%$, where 30% is a variable threshold that can be adjusted.

2. Another or additional way to determine if the clusters are complementary is to apply the KMeans algorithm to determine if the complementary centroid, counted as maximal value minus centroid vector, can be part of the matching cluster.

3. The results from the first two verifications are saved in a dataset called complementation_stats. This array will be used to search for exact matches of microservices. While sorting is optional at this stage, it may be beneficial to sort the pairs by their discrepancy values for the subsequent steps. This will allow fast finding the next centroid pair to match by going to the next pair in a loop. Downside of the sorting at this stage is possible size change of the array because of deleting or marking empty some of the clusters that may be parts of the other pairs.

4. Taking *complementation*_stats, we process pairs starting with the smallest discrepancy. For each pair of clusters, microservice matches by comparing the minimum, maximum, and range characteristics of the microservice data are identified. Range is the difference between max and min values. This is done to ensure the discrepancy of these characteristics is less than allowed value $D_{\text{microservice}}$ which is taken also equal to 30% in the example. This threshold is separately adjustable. Not all vectors may find matches due to varying amplitudes and possible different number of vectors in a centroid.

5. Once a match is found, the vectors are recorded in an array named `complimentary_microservices`. They are also removed from their respective clusters.

6. If a cluster becomes empty, its pairs, especially not visited ones, are marked as inactive in *complementation_stats.* An ongoing search involving this cluster is also stopped.

7. The steps 4-6 are repeated for all active clusters pairs in *complementation_stats* until no further matches can be found.

An example of the result of cluster matching is presented below. The mark “Active: No” means that one or both of the clusters contain zero elements after the matching took place:

Clusters 3 and 4:
Discrepancy: 2.53%
Centroid Match: Yes
Active: No
Clusters 1 and 3:
Discrepancy: 16.44%
Centroid Match: Yes
Active: No

8. If not all microservices found their matches from the first iteration, the algorithm should be repeated starting from the clusterization with the KMeans method. The number of clusters can be chosen as the number of possible groups of microservices described in 4.2.4 Example of how grouping works possibly multiplied by 2, to better divide the “cases and anti-cases”. This makes 10 in total. If there are not so many microservices, more than 27, the cluster number can be picked as the number of microservices divided by 3 plus 1 as in (16). This formula is empirical and is based on thoughts that when the microservices are too different, it is hard to unite them in one cluster, or the cluster quality will be low. Too many clusters are not needed either. The microservices' pattern strongly varies when the majority is processed and little is left which cannot find their pairs. If no matches are found in an iteration, this is a reason to stop the algorithm and process to the next step.

$$\text{Cluster Number} = \left\lceil \frac{\text{Number of Microservices}}{4} \right\rceil + 1 \leq 10 \tag{16}$$

Where $\lceil x \rceil$ denotes the ceiling function, which rounds up to the nearest integer.

4.3.2. Handling Empty Matches

There is a chance that all the microservices will find their perfect matches from the first iteration. Yet, in general case it is not so probable. The reasons can be an odd number of the microservices or amplitudes that strongly vary. These both cases can be fixed by combining more than two microservices for a match or devoting separate servers to very odd clusters with often or unpredictable high load. The criteria are the following:

1. If the max value of the microservice is high, close to maximal, and the average load is over a threshold equal to, let's say 70%, then this microservice requires a separate instance.
2. If the load is hard to predict, and it can strongly vary from day to day, then a separate instance is also required.
3. In the other case, microservices with minimal load can be combined based on the best possible match determined by (15) and in a combined state enter the clusterization algorithm with all the next steps again.

An example of using this algorithm with an image is proposed in this article [25].

5. Conclusion

The paper describes in details the algorithm of finding the complimentary microservices providing explicit examples of how the stages work and interact with each other. While the general load distribution algorithm for complimentary microservices integrates predictive analysis, dynamic microservices orchestration, load balancing, and server regrouping to optimize resource utilization in cloud computing environments, it is drastically different from the present solutions for the microservice cloud management. Relying on anticipated load patterns and efficiently managing resources through putting a combination of matching microservices onto server groups, the algorithm ensures high performance, reliability, and cost-effectiveness.

The incorporation of algorithms for finding similar microservice groups and combining them in search for the complementing microservices adds a critical layer of optimization to the load distribution process. By profiling microservices and regrouping combinations when they don't already work well, the system maximizes resource utilization and enhances overall efficiency.

A very important stage of planning is to receive correct input data. Taking in consideration the differences between initial servers on which tests were made and a real production environment, the normalization should be applied when transforming data to the new cloud-server capabilities. By introducing an adjustment coefficient to account for the overhead and inefficiencies of distributing tasks across multiple cores, we can derive a more accurate estimate of the processor load in the cloud environment and give space for small errors during calculations. This approach acknowledges that simply having more cores does not linearly translate to better performance and accounts for the practical limitations of multi-core processing. In practice, the value of the adjustment coefficient can be fine-tuned based on empirical data and specific characteristics of the microservices and cloud environment. Chance of errors and possible differences in data is also taken into consideration with the help of small delta provided in initial data statistics and collected during metrics of real usage.

During implementation of the compliment search algorithm Z-Score algorithm is used for data normalization which leaves the usage pattern and removes attachment to the load amplitude. KMeans algorithms separates the load time series into groups of similarity. Those groups are combined to find, which ones work the best together using our own algorithm. As a result, microservices that can be combined to provide a maximal possible common load, are extracted. FastDTW method is used for visualisation of intermediate results.

In the paper, other relative methods are also discussed. It is grounded why they are not leading to a better result. The methods are DBSCAN for clusterization, cosine similarity for grouping load vectors, and minimax algorithm for normalization.

Combining the mentioned algorithms enables a robust approach to finding, grouping and matching complementary microservices.

References

[1] "Generative AI has a clean-energy problem," The Economist. Accessed: Jul. 14, 2024. [Online]. Available: https://www.economist.com/business/2024/04/11/generative-ai-has-a-clean-energy-problem?utm_medium=cpc.adword.pd&utm_source=google&ppccampaignID=18151738051&ppcadID=&utm_campaign=a.22brand_pmax&utm_content=conversion.direct-response.anonymous&gad_source=1&gclid=CjwKCAjw7s20BhBFEiwABVIMrV-JpPhT92Zwu5KdjtZu1zRhuBCW2sIxQ25KwY180FhpWimVIV_tchoCjbwQAvD_BwE&gclsrc=aw.ds

[2] Aspire Systems, "Microservices Architecture: The Foundation of Cloud-Native Applications," Software Engineering. Accessed: Jul. 14, 2024. [Online]. Available: https://blog.aspiresys.com/software-product-engineering/microservices-architecture-the-foundation-of-cloud-native-applications/

[3] S. N. A. Jawaddi, A. Ismail, and V. Cardellini, "Modeling and Verifying Microservice Autoscaling Using Probabilistic Model Checking," In Review, preprint, May 2022. doi: 10.21203/rs.3.rs-1682990/v1.

[4] Y. Sharma, "Key Strategies for Implementing AWS Network Load Balancer," DEV Community. Accessed: Dec. 29, 2023. [Online]. Available: https://dev.to/aws-builders/key-strategies-for-implementing-aws-network-load-balancer-35fc

[5] O. Dmytrenko and M. Skulysh, "Fault Tolerance Redundancy Methods for IoT Devices," Infocommunication Comput. Technol., vol. 2(04), no. University "Ukraine," pp. 59–65, Dec. 2022.

[6] P. Salot, "A Survey of Various Scheduling Algorithm in Cloud Computing Environment," Int. J. Res. Eng. Technol., vol. 02, no. 02, pp. 131–135, Feb. 2013, doi: 10.15623/ijret.2013.0202008.

[7] Kubernetes Team, "Kubernetes Scheduler." Kubernetes Documentation, Dec. 14, 2023. [Online]. Available: https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/

[8] Y. Sharma, "Key Strategies for Implementing AWS Network Load Balancer." Sep. 27, 2023. [Online]. Available: https://dev.to/aws-builders/key-strategies-for-implementing-aws-network-load-balancer-35fc

[9] H. Liu, Y. Li, and S. Wang, "Request Scheduling Combined with Load Balancing in Mobile Edge Computing," IEEE Internet Things J., pp. 1–1, 2022, doi: 10.1109/JIOT.2022.3176631.

[10] R. L. Collins and L. P. Carloni, “Flexible filters: load balancing through backpressure for stream programs,” in Proceedings of the seventh ACM international conference on Embedded software - EMSOFT ’09, Grenoble, France: ACM Press, 2009, p. 205. doi: 10.1145/1629335.1629363.

[11] I. Journal, “The Research Study on DynamoDB – NoSQL Database Service”, Accessed: Jul. 14, 2024. [Online]. Available: https://www.academia.edu/8818008/The_Research_Study_on_DynamoDB_NoSQL_Database_Service

[12] G. Weintraub, Dynamo and BigTable — Review and comparison. 2014, p. 5. doi: 10.1109/EEEI.2014.7005771.

[13] S. Dutta, “MySQL vs DynamoDB: Comparing Relational and NoSQL Database Solutions,” Sprinkle. Accessed: Jul. 14, 2024. [Online]. Available: https://www.sprinkledata.com/blogs/mysql-vs-dynamodb-a-comprehensive-comparison

[14] “Amazon DynamoDB Customers | AWS,” Amazon Web Services, Inc. Accessed: Jul. 14, 2024. [Online]. Available: https://aws.amazon.com/dynamodb/customers/

[15] “Database Migration - AWS Database Migration Service - AWS,” Amazon Web Services, Inc. Accessed: Jul. 14, 2024. [Online]. Available: https://aws.amazon.com/dms/

[16] R. E. Bryant and D. R. O’Hallaron, Computer systems: a programmer’s perspective, 2. ed., vol. Chapter 9: VM as a Tool for Caching. Boston, Mass.: Prentice Hall, 2011. [Online]. Available: http://54.186.36.238/Computer%20Systems%20-%20A%20Programmer%27s%20Persp.%202nd%20ed.%20-%20R.%20Bryant%2C%20D.%20O%27Hallaron%20%28Pearson%2C%202010%29%20BBS.pdf

[17] A. Belgacem, “Dynamic resource allocation in cloud computing: analysis and taxonomies,” Computing, vol. 104, no. 3, pp. 681–710, Mar. 2022, doi: 10.1007/s00607-021-01045-2.

[18] “AWS vs Azure vs GCP cloud comparison: Databases.” Accessed: Jul. 14, 2024. [Online]. Available: https://www.pluralsight.com/resources/blog/cloud/aws-vs-azure-vs-gcp-cloud-comparison-databases

[19] “Vertical Pod autoscaling | Google Kubernetes Engine (GKE),” Google Cloud. Accessed: Jul. 14, 2024. [Online]. Available: https://cloud.google.com/kubernetes-engine/docs/concepts/verticalpodautoscaler

[20] R. L. Collins and L. P. Carloni, “Flexible filters: load balancing through backpressure for stream programs,” in Proceedings of the seventh ACM international conference on Embedded software - EMSOFT ’09, Grenoble, France: ACM Press, 2009, p. 205. doi: 10.1145/1629335.1629363.

[21] S. J. Bigelow, “What is edge computing? Everything you need to know,” Search Data Accelerator. Accessed: Jun. 07, 2024. [Online]. Available: https://www.techtarget.com/searchdatacenter/definition/edge-computing

[22] X. Cao, F. Wang, J. Xu, R. Zhang, and S. Cui, “Joint Computation and Communication Cooperation for Energy-Efficient Mobile Edge Computing,” IEEE Internet Things J., vol. 6, no. 3, pp. 4188–4200, Jun. 2019, doi: 10.1109/JIOT.2018.2875246.

[23] B. Gregg, Systems Performance, 2nd edition. Pearson, 2020.

[24] O. Al-Debagy and P. Martinek, “A Comparative Review of Microservices and Monolithic Architectures,” in 2018 IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI), Nov. 2018, pp. 000149–000154. doi: 10.1109/CINTI.2018.8928192.

[25] O. Dmytrenko and M. Skulysh, “Determining microprocessor groups for efficient utilization of processor capacities” Problems of programing Forthcom., vol. 1, no. №2-3, Aug. 2024, [Online]. Available: https://docs.google.com/document/d/1ocqA-e7d-OAibFVGiEE2UcxZRKgYNn_yPX6e94dHwYM/edit?usp=sharing

[26] Codecademy Team, “Normalization,” Codecademy. Accessed: Jul. 18, 2024. [Online]. Available: https://www.codecademy.com/article/normalization

[27] S. Salvador and P. Chan, Toward Accurate Dynamic Time Warping in Linear Time and Space, vol. 11. 2004, p. 80. [Online]. Available: https://cs.fit.edu/~pkc/papers/tdm04.pdf

[28] I. T. Jolliffe, Ed., "Graphical Representation of Data Using Principal Components," in Principal Component Analysis, New York, NY: Springer, 2002, pp. 78–110. doi: 10.1007/0-387-22440-8_5.

[29] I. T. Jolliffe, Ed., "Principal Components as a Small Number of Interpretable Variables: Some Examples," in Principal Component Analysis, New York, NY: Springer, 2002, pp. 63–77. doi: 10.1007/0-387-22440-8_4.

[30] I. T. Jolliffe, Ed., "Choosing a Subset of Principal Components or Variables," in Principal Component Analysis, New York, NY: Springer, 2002, pp. 111–149. doi: 10.1007/0-387-22440-8_6.

[31] O. Dmytrenko and M. Skulysh, "Method of Grouping Complementary Microservices Using Fuzzy Lattice Theory," vol. 12, no. 1, pp. 11–18, Mar. 2024, doi: 10.25673/115636.

[32] M. Skulysh, "The method of resources involvement scheduling based on the long-term statistics ensuring quality and performance parameters," *2017 International Conference on Information and Telecommunication Technologies and Radio Electronics (UkrMiCo)*, Odessa, Ukraine, 2017, pp. 1-4, doi: 10.1109/UkrMiCo.2017.8095430.

[33] M. Skulysh and S. Sulima, "Management of multiple stage queuing systems," *The Experience of Designing and Application of CAD Systems in Microelectronics*, Lviv, Ukraine, 2015, pp. 431-433, doi: 10.1109/CADSM.2015.7230895.