Method of Grouping Complementary Microservices Using Fuzzy Lattice Theory

Oleksandra Dmytrenko[1] and Mariia Skulysh[1,2]
[1]*Institute of Telecommunication Systems, National Technical University of Ukraine "Igor Sikorsky Kyiv Polytechnic Institute", Beresteiskyi Avenue 37, 03056 Kyiv, Ukraine*
[2]*Anhalt University of Applied Sciences, Bernburger Str. 57, 06366 Köthen, Germany*
olexandra.dmytrenko@gmail.com, mskulysh@gmail.com

Keywords: Microservices, Cloud Energy Efficiency, Fault Tolerance, Shared Instance Group, Cluster.

Abstract: This paper contains ideas on how to optimize the costs of running a microservice system. Currently, there is much done to provide high fault tolerance of a microservice and a system as a whole. Cloud providers come up with new ways to guarantee the high speed of newly launched instances. This leads to a ubiquitous run of redundant servers with possible cold or hot standby mode. This is often crucial because the ability to use some applications quickly and on time can be important to many users, potentially saving lives. At the same time, it's important to prioritize ecological preservation and minimize overuse of the Earth's resources. In the context of cloud, and specifically, server computing, that would involve using resources in a way that extends their lifespan, minimizing the creation of slowly decomposing waste, and avoiding excessive energy consumption. Cloud providers, such as Amazon, Google, and Azure, discard millions of underused hardware units due to the necessity of ensuring service guarantees to their customers. In the article, method to optimize the usage of servers by organizing microservices in complementary sets are described. As a result, server resources will be used most efficiently. The method of grouping the microservices can be likened to the principles of lattice theory. The ideas in the article could be useful for the systems like Kubernetes scheduler in the stage of picking the right set of instances to run a new microservice, or to cloud providers. As a result, less energy and hardware resources will be used to provide the same quality of fault tolerance.

1 INTRODUCTION

These days, many businesses try to make their program products scalable in order to handle high loads efficiently. That's why designing a program as a group of microservices that can be replicated across multiple instances is becoming increasingly popular. To support this kind of structure, a microservice manager platform is needed. Docker, Kubernetes, and different cloud platforms offer assistance in managing load and scalability by deploying new microservice instances that could help overcome the limitations of a single instance.

In most cases, application loads are predictable. The impact of sudden load surges has a significant influence on system stability. The main idea is to distribute microservices among server groups first taking into account prioritizes balancing the overall load of the group, then followed by predictability and ensuring that a certain bearable maximum that the server group can provide is not exceeded.

2 GROUPING COMPLEMENTARY MICROSERVICES ACCORDING TO PRINCIPLES OF FUZZY LATTICE THEORY

2.1 Fuzzy Lattice Theory in Relation to Microservices

In this paper, the complementarity of microservices and demonstrate how their coordinated utilization can significantly reduce wasteful resource usage will be discussed. Let's consider a definition of complementary objects from the lattice theory. A lattice b is complemented to lattice a is a bounded lattice that satisfies

$$a \vee b = 1 \text{ and } a \wedge b = 0 \quad (1)$$

In linguistic terms, the initial formula can be interpreted to signify the absence of overlapping resource utilization, wherein concurrent usage of a single resource by both microservices does not occur. Simultaneously, both microservices collectively leverage all available resources during operation. A complementary lattice need not necessarily be unique. A bounded lattice means that the minimal element is 0 and the maximal element is 1 [1]. The lattice theory contains definitions of a relatively complemented lattice and orthocomplementation, which could also be useful in showing the parallelism with the idea of organizing microservices into groups.

Let us expand this idea to microservices running on a single resource group, or simplifying, on one server. It means that the microservices are complementary when, if running together, they fully utilize the server's capacity, and if they are off, the server remains idle.

To generalize, taking in consideration that it might be difficult to find an ideal complementary type of microservice, let's assume that *b* can be recursively replaced with a set of two other microservices, *c* and *d,* that follow the same rule as in (1) by sharing the total load in proportions that will never exceed 1 and that won't fully interact with *a*:

$$a \vee (c \vee d) = 1,\ c \wedge d = 0,\ a \wedge d = 0,\ a \wedge c = 0 \quad (2)$$

This kind of recursion can occur multiple times, meaning there is no limit to how many microservices a resource group can contain. The main principle is to achieve the efficient usage of the group.

2.1.1 Adding Fuzziness

When dealing with resource load, incorporating fuzziness can be more appropriate. To that end, we modify the definition of the microservices to allow partial resource load. It can be proportional or not, but the total load at any moment won't exceed the full possible load.

Let us define the following terms:

- *C*: a certain processor characteristic, such as processor load, channel throughput, user number, or memory usage;
- T_j: a certain time interval, e.g. an hour or "morning", during which C is measured. The cycles are repeated every 24 hours. Certain differences can be also made for holidays and weekends;
- *j*: a time unit. We consider that time is divided into reasonable units. For example, what interests us are the hourly measurements during the day or fuzzy definitions of "morning", "day", "evening" and "night" time.
- *i*: an index of a microprocessor that runs from 1 to some positive integer *N*.
- $\mu T(C_i)$: the degree of a certain characteristic C of the microprocessor *i* during time *T*.

If microservices 1 and 2 are the two complementary microservices, then $\mu_T(C_1)$ and $\mu_T(C_2)$ are the degrees of presence of a given characteristics C during a selected time *T*. Together they should form a load of a server resource approaching to a full possible load. Based on given definitions above, we can rewrite a formula of fuzzy complementation presented in [2] in the following way:

$$\mu_{Tj}(C_2) = 1 - \mu_{Tj}(C_1) \quad (3)$$

Every microservice has its usual, average, minimal, and maximal loads. When referring to (2) it means that disregarding the exact load of one microprocessor, the load of the second one should be adopted. The complementation should be based using the definition: maximal intersecting load at any same time should not exceed full load. In the words of formulas, that is:

$$\max(\mu_{Tj}(C_1), \mu_{Tj}(C_2)) < 1 \quad (4)$$

Till now, we have spoken about having 2 complementary microservices. With the same success, we can extrapolate the formulas to more elements. To bound *N* - the maximal number of the microservice instances that could run on one server group, let us take into consideration a number of threads *K* on the group processor(s). Then $i \in [1, K]$. Making generalization, the formulas (3) and (4) would be the following:

$$\mu_{Tj}(C_N) = 1 - \mu_{Tj}(C_1) - \mu_{Tj}(C_2) - \ldots - \mu_{Tj}(C_{N-1}) \quad (5)$$

$$\max(\mu_{Tj}(C_1), \mu_{Tj}(C_2), \ldots, \mu_{Tj}(C_{N-1}), \mu_{Tj}(C_N)) < 1 \quad (6)$$

2.1.2 Illustration of an Idea

To illustrate the idea of creating balanced resource groups, consider a simple example. Suppose there are 2 companies with private hostings: a ticket company, which experiences a high load during the daytime, and an online casino, whose main activity takes place at night. To optimize the number of hardware resources spent for hosting these 2 enterprises, a common hosting could be considered. The same servers would be reused by an application that has to handle more load at a particular moment. The total common load remains the same, but fewer resources for running both programs are used. The transition in resource usage by a specific application would be relatively gradual.

Potential spikes in load for each application during non-standard times could occur due to sales and special offers, as well as unpredictable sudden events, such as heavy weather conditions and disasters. If the applications provide services in the same region, evidently, in the second case people would be more likely to prioritize buying tickets and saving their lives over playing in casinos, both during the day and at night.

On the other hand, the first case is more difficult, necessitating either a localized or a comprehensive resolution. A potential local resolution involves negotiating between companies regarding staggered timing for special offers. This possibility is unlikely, as it would require the companies to be managed by friends or the same person. This approach also entails considering the risk of one or both companies having their income undermined. A general resolution implies providing additional resources that are rarely used, but which are crucial for safe and stable working conditions. Such resources would also be useful when components of the hardware degrade. Greater hardware durability can be anticipated when operating below maximum capacity. Additional resources would also serve to provide longer life expectancy for the servers.

Increased resource allocation and simultaneous higher loads for both applications could raise concerns about profitability of shared hardware. This is where the concept of dividing an application into microservices would act as a resource optimizer. Every microservice has different tasks and different general loads in the same period compared to other microservices of the same application. In online stores, people spend significantly more time on product selection than on ordering and payment procedures. As a result, much fewer login activities are to be expected than filtering goods activities. Hence, combining low-load microservices from one application with high-load microservices from another application is essential for achieving a balanced group.

As a result, when having high activities in both applications at the same moment, a balanced group of microservices will not require a substantial amount of resources. As a consequence, a general margin of safety for a shared space among several microservice applications will be reduced compared to using private spaces. At the same time, general fault tolerance will be higher [3].

2.1.3 Comparison of the Example with the Fuzzy Lattice Theory

In the context of lattice theory, a complementary microservice that achieves the same maximum state of processor usage should be identified, ensuring that only one microservice is active at any given moment.

Let us denote by *a* a microservice of the ticket company that works in the daytime, and by b, a microservice of the casino that works at night. Let us also denote by 1 a capacity of the server/cluster/resource group that runs both microservices and at the same time is a potential maximal safe capacity of one of the microservices. Likewise, let 0 denote a state when everything is idle or which should not happen.

Following (1) and its fuzzy counterpart (3), we can say that we want to achieve state 1 when either *a* or *b* is working. The state of partial load is also acceptable: if it’s 80% of *a*, then it’s no more than 20% of *b*, and similarly for (4) and (6). States where one is 60% loaded and the other one is 10% loaded are also acceptable, as they comply with the formulas for fuzzy sets. In that case, a server could run something else in addition during this time.

The maximum resource capacity is designed only to support partial load. In case when both microservices need more resources, then replication should be activated and a state of 0 must be prevented.

Taking into consideration that there may be several complementary paths, it is worth mentioning that a microservice that could fit as a dual pair to the first one is not unique. Not only ticket companies work at night. A bank microservice could similarly fit the casino one in this regard.

2.2 Grouping Strategies

The example provided is reasonable, however, having a standard set of rules or considerations for grouping microservices would be beneficial. A group by itself implies that the physical servers are located as close as possible and are connected in a local group via wires, or this is a single supercomputer with plenty of resources. Below, we list criteria that may influence decision-making.

2.2.1 Security or Multi-Tenancy

Microservices can be grouped by security reasons or user access rights. This is the most secure way of grouping. However, the challenges posed by multi-tenancy can be solved in several ways, depending on specific needs and risks.

1) Strong division of common space among several running applications can be performed by virtualization.
2) Containerization can be used in addition to an already running operating system.

3) One can launch applications from different users in terms of one operating system, where every user has its predefined space.
4) In a public cloud environment, it is possible to segregate the same physical server into secure logical spaces for every user or tenant, thereby ensuring a high level of security.
5) Single tenancy, which requires separate microservices for separate user groups or users, can also be provided in terms of one group.

At the same time, extremely stringent security requirements are relatively uncommon. To provide an adequate level of security, it is not necessary to launch microservices that require the same user access rights in the same group. Unless it is required, it would be more effective to base the division on alternative guidelines, from a resource efficiency perspective.

In Kubernetes, multi-tenancy often involves many teams sharing the same cluster. It also has a so-called SaaS tenancy, where multiple clusters with different applications are provided to a single user team [4].

2.2.2 Shared Resources

Most applications are not standalone but have separate resources such as databases. In a microservice architecture, it is common to have a shared schema registry, DTO classes, domains, libraries, and protocols that are stored separately, exclusively for use by two or more microservices. Grouping can also be based on using the same shared resource.

If several microservices communicate with each other often, it is beneficial to have a system design with a focus on maximal failure prevention, particularly during data transmission. Data inconsistencies caused by partially sent and processed data can be challenging to roll back. Addressing such inconsistencies may necessitate actions from each microservice that processed the information. This is described in the SAGA pattern. Alternatively, creating specialized clean-up procedures could be an option. These procedures would know all potential failure scenarios and would be capable of making manual rollbacks, fetching information from the backup files, or recalculating updated values and writing previous consistent values.

Rollback procedures in a microservice architecture are risky. There is no guarantee that some part of the application won’t read temporary incorrect values. Both strategies are time-consuming and include multiple operations, rather than completing everything in a single step. The human factor is also present because when the code is updated, a developer might forget to accordingly modify the rollback procedure. The whole process is complicated by communication problems. If the communication issue happens on the forward path, it may also happen in the reverse direction. Therefore, eliminating the problem is more beneficial than solving it.

In a microservice architecture, the following problems are possible: latencies, outages in one part of the system, failure to establish the connection because of the connecting hardware or its settings updates, limitations of queues and their operational peculiarities, and other physical or logical issues. It is always better to eliminate as many potential failures as possible and increase fault tolerance, as long as it does not affect the quality of work and does not increase the downsides of a software product dramatically. Grouping by shared resources will reduce many possible issues related to connectivity problems. In case of outages, a whole group will fail, which will keep the information in the database consistent with a higher probability. At the same time, a downside of such grouping is not being able to respect the load of microservices and to provide effective usage of the resources.

2.2.3 Channel Throughput

There are cases when the most busy resource is the channel resource, when a microservice needs to handle massive and/or very frequent data chunks. This may not produce a high processor load but requires many threads to be able to pick up the incoming information. If the activity for such a microservice increases, scaling becomes necessary.

Supporting such a microservice requires a significant amount of resources. The best option would be to combine it with other microservices that would either accept rare requests and possibly have high processor load, or function similarly, but during different hours. Needless to say, such microservices are risky, and if possible, it is advisable to avoid heavy traffic between application components by redesigning the system and the purpose of microservices. There are techniques to prevent it. Introducing an additional stage of data preprocessing or storing intermediary results in the database could be beneficial. Data could undergo primary filtering and be routed in several directions.

2.2.4 Time and State of Processor Load

There are microservices designed specifically for data processing, such as machine learning routines, which require many computational resources [5]. Such tasks may be time-consuming and require high load on the processor. It might be possible to split high processor-load microservices into several ones to reduce the amount of operations they have to perform. Consequently, this will reduce the time of processing a single request and accordingly increase the throughput of a single instance of such a microservice. In many cases, though, researchers and engineers prefer having all logic in one place and reusing its components in various places, rather than dispersing it across multiple microservices, thus duplicating the code or increasing the traffic between the application components [6].

This kind of microservice would normally be the heart of the application and would receive fewer requests compared to the other components. This stems from preceding procedures of filtering erroneous data and data aggregating activities, which reduce the number of initial requests. To create a resource optimizing group, such microservices could be paired with others that process their tasks quickly and do not experience heavy traffic concurrently with the active phase of the first set. Complementary microservices should not tie up threads for extended periods, allowing them to quickly become available for new tasks. In other words, we should combine the microservices when one needs 2 seconds to process one request, while its counterpart processes 10 requests in 1 second.

2.2.5 Geographical and Active Time Reasoning

The speed of user response decreases as the distance to the processing server increases. Hence, globally distributed applications would prefer to run separate instances of their applications on regional servers or cloud centres to provide the best user experience [7]. For example, Amazon servers are currently located in 32 geographical regions [8]. Consider, on the other hand, a company that is present on one continent, e.g. North America, and also has a presence on another continent where working hours do not significantly overlap, e.g. Europe. Depending on the goals of the company and the most likely sources of its profit, there might be no compelling reasons to establish costly hosting on several continents. People using the services in unpopular locations would agree to wait for a response from a distant server and would stay loyal to the service provider they use.

As an example, consider cell phone providers working in a certain county. When their customers travel, the calling services are provided by local providers instead, but the mobile application should work regardless of the user's current location. When on a tourist trip to another country and waiting for information updates in a mobile app, a user may be more ready to receive a delayed response than usual. This response time will not influence his choice of a cell phone provider when he comes back. Awareness of the existence of companies and applications that operate in this manner gives us reasons to plan for the more optimal utilization of shared space.

In terms of grouping, this implies that we could combine microservices to provide services not only at different times for the same region but also for different geographical regions, whose time zones can be considered opposed due to their lack of overlap in active usage times. In the above example, it could be convenient to make one logical and possibly physical group of servers as a set of instances hosting the same microservices, with each one providing services to the opposite geographical regions with low user intensity.

Table 1 and Figure 1 presents an outline of all reasonable ways of combining microservices, allowing them to form a complementary group and be collocated within a single server group.

2.3 Existing Solutions

To understand the existing research in this area, we performed a basic review of the most popular designs of microservice resource usage optimization.

2.3.1 What are the Kubernetes Scheduling Solutions?

In Kubernetes documentation, it is mentioned that there are many factors to determine the server for running a new instance and that is the most feasible. The major ones include individual and collective resource requirements, hardware/software/policy constraints, affinity and anti-affinity specifications, data locality, and inter-workload interference [9]. These approaches are based on physical abilities and the best possible speed of intercommunication. That doesn’t include the optimal solution from the economic or ecological points of view, which are of high importance these days.

2.3.2 How Amazon Cloud Manages the Load Balancing?

Amazon Cloud has its Elastic Load Balancing (ELB) which includes three elements. The first one is the Application Load Balancer. It supports host-based and path-based routing, meaning that the traffic is routed based on its content or its headers, the domain name. This idea is not unlike geographical grouping,

Table 1: The grouping possibilities to optimize resources usage.

Group Name	Primary (problematic) microservice	Complementary microservice	When to use	Problems
Security	A microservice with certain security standards.	Other microservices with the same security standards.	To manage multi-tenant access to a shared resource.	Might not respect the load of microprocessors and the group won't optimally use resources.
Shared resource	A microservice that uses a certain external resource that is launched on the same subcluster.	Other microservices that use the same shared resource.	When there is a need to have maximal speed of communication between external resources and microservice.	Might not respect the balance of microservice group load, and might not provide effective usage of the resources.
Heavy channel throughput	Massive and/or very frequent chunks of data should pass through a microservice.	Microservices that would either accept rare requests and possibly have high processor load, or work in a similar way, but in the other daytime.	When many threads are needed to pick up the incoming information.	Heavy traffic microservices are risky because of possible lack of space in the queues, more often failing to send the information and delays in processing.
Heavy processor load	Tasks that are implemented during a long time and/or heavily load the processor.	Complementary microservices should not hold threads for a long time and so let them be ready for new tasks quickly.	When not many threads are needed, but high processor capacity is called for.	Time of returning response can be a problem in real-time applications.
Geographically separated	Applications handling a large number of concurrent connections resulting in high load by channel throughput and/or processor load.	Microservice that has a high load of a similar type but which working hours are not intersecting.	When the time of heavy load is stable and limited to a certain part of the day,	Problems may arise if the complementary microservices have sudden unexpected picks of activity in the intersecting time.

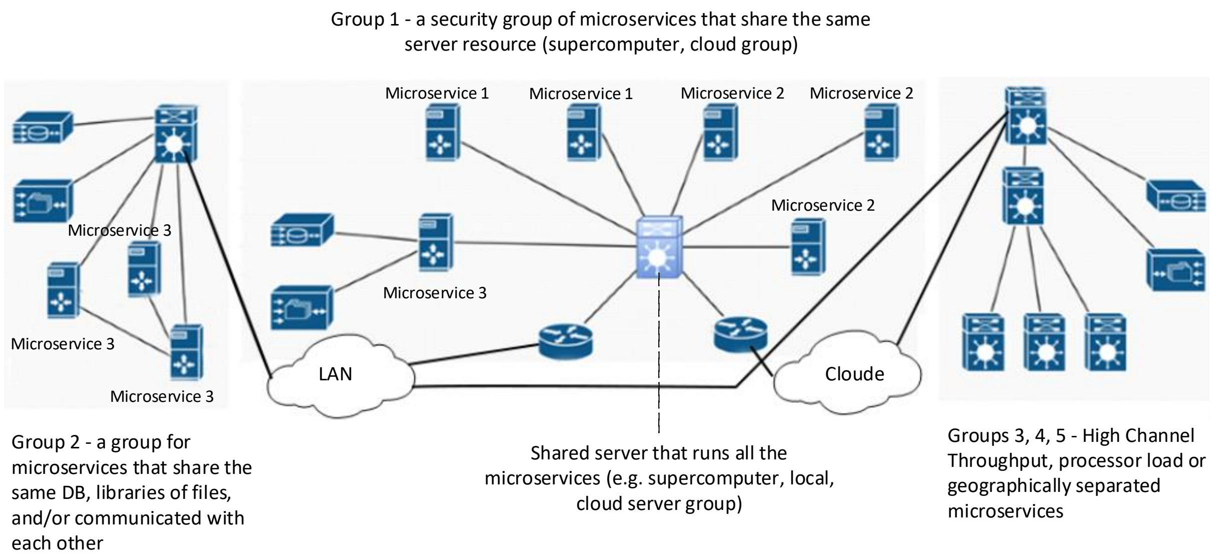
Group 1 - a security group of microservices that share the same server resource (supercomputer, cloud group)
Microservice 1
Microservice 1
Microservice 2
Microservice 2
Microservice 2
Microservice 3
Microservice 3
Microservice 3
Microservice 3
LAN
Cloude
Group 2 - a group for microservices that share the same DB, libraries of files, and/or communicated with each other
Shared server that runs all the microservices (e.g. supercomputer, local, cloud server group)
Groups 3, 4, 5 - High Channel Throughput, processor load or geographically separated microservices

Figure 1: Possible strategies of organizing servers into groups mentioned in the Table 1.

but in fact is opposite in spirit. Groupings are done by close locations rather than the opposite ones.

The second one is the Network Load Balancer. Its purpose is to prevent connection draining before a target is considered unhealthy and evenly distribute the traffic in a cross-zone mode which makes the optimal resource utilization. The flow-based distribution algorithm is used to make the network load balancing particularly suitable for applications that benefit from predictable and consistent connection handling, making it well-suited for a variety of use cases, including those with stateful and connection-oriented requirements

The third element is the combination of the two mentioned ones, called a Classic Load Balancer [10], [11].

The methods of load balancing are very well-thought-out and are meticulously designed and organized to serve users in the most effective manner possible. The techniques are also formulated to ensure equitable sharing of resources, thereby extending the longevity of the last serve [12]. The resources are turned off as soon as they are not needed, which prevents wasteful energy consumption. Nevertheless, in Cloud management theory, there is not information on how to reduce the number of needed servers in order to optimize the cloud activity. The methods described in Table 1 could help to achieve this.

Google Cloud [11] offers similar functionality.

3 CONCLUSIONS

Running multiple microservices and scaling them to ensure fault tolerance for the entire application, as well as its ability to effectively handle any number of users, is a critically important task. When choosing the right node, cluster, or instance group to run a certain instance of the microservice at cloud and Kubernetes, one of the many criteria to be taken into account is the ecological component. This means fullutilization of resources of one server, reducing their number to an absolute minimum. Such an approach also helps minimize electricity bills, which can be huge for constantly operating machine loads.

The article proposes forming instance groups of complementary microservices using the rules of a complemented lattice in fuzzy logic. The most efficient can be the groupings based on

- Difference in the time zones.
- Difference in the style of work of the applications.
- Balance between high or long processor load and amount of fast requests that the application handles.

At present, such criteria are not included in the list of factors that Kubernetes and popular cloud services use to decide on which server to run an instance.

REFERENCES

[1] G. A. Gratzer, "General lattice theory." Pure and Applied Mathematics: A Series of Monographs and Textbooks, no. 75, Academic Press, New York, 1978.

[2] N. Ajmal and K. V. Thomas, "Fuzzy lattices," Information Sciences, vol. 79, no. 3-4, pp. 271-291, Jul. 1994, doi: 10.1016/0020-0255(94)90124-4.

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

[4] Kubernetes Team, "Multi-tenancy in Kubernetes," Kubernetes, [Online]. Available: https://kubernetes.io/docs/concepts/security/multi-tenancy/, [Accessed: 23 Dec 2023].

[5] "A review of in-memory computing for machine learning: architectures, options," International Journal of Web Information Systems, Dec. 2023, doi: 10.1108/IJWIS-08-2023-0131.

[6] F. Wilhelmi, D. Salami, G. Fontanesi, L. Galati Giordano, and M. Kasslin, "AI/ML-based Load Prediction in IEEE 802.11 Enterprise Networks," 2023.

[7] A. Meir, "Does Location Matter In Cloud Computing?," Ridge Cloud, [Online]. Available: https://www.ridge.co/blog/location-in-cloud-computing/, [Accessed: 23 Dec 2023].

[8] Amazon Team, "AWS Global Infrastructure". [Online]. Available: https://aws.amazon.com/about-aws/global-infrastructure/?nc1=h_ls, [Accessed: 22 Dec 2023].

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

[10] 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.

[11] S. Al-Raheym, S. C. Açan, and Ö. T. Pusatli, "Investigation Of Amazon And Google For Fault Tolerance Strategies In Cloud Computing Services," AJIT-E Online Acad. J. Inf. Technol., vol. 7, no. 23, pp. 7-22, Nov. 2016, doi: 10.5824/1309-1581.2016.4.001.x.

[12] L. Globa, M. Skulysh, and A. Zastavenko, "The method of resources allocation for processing requests in online charging system," in The Experience of Designing and Application of CAD Systems in Microelectronics, 2015, pp. 211-213, doi: 10.1109/CADSM.2015.7230838.