Infrastructure Cost Analysis for Running Timetabling Software Using Various Deployment Architectures in the Cloud

Sommer Harris, Jaelyn Ma, Meet Patel, Kshitij Nair, Niranjan Velraj, Michal Aibin, *Senior Member, IEEE*
Khoury College of Computer Science, *Northeastern University*
Vancouver, Canada
vancouver@northeastern.edu

***Abstract*—There is no standardized way to deploy applications in the cloud. The suitable deployment strategy depends on the application type and the business's specific requirements. We tested five deployment architectures and evaluated metrics such as reliability factor and cost per month to determine which best suits school timetabling business, requiring relatively high system reliability and a low-performance cost model.**

***Index Terms*—cloud deployment, deployment architectures**

I. Problem Introduction

The timetable is such an important part of school functioning that some school administrators spend months or weeks trying to timetable the curriculum using Post-it notes [1]. This directly affects institutions with multiple and varied constraints in their timetabling. Since the onset of COVID-19, educational leadership roles have changed dramatically, with one of the largest challenges being time management given increasing demands [2]. Providing effective timetabling solutions could return to these administrators weeks or months of their valuable time. There are a few standalone educational timetabling software products like aSc Timetables [3] and Lantiv [4]. Over decades, these applications have attempted to accommodate the growing constraints. However, there is still a gap with timetabling being usable only by experts [5].

The long-term vision of this project is to further bridge the gap between research and industry by developing a cloud-based timetable application for schools. Cloud-based solutions can offer a range of features with varying costs, which makes it imperative to explore and find a balance between features that meet our use case most cost-effectively. To address accessibility and a larger number of consumers, we also explore in this paper various scalability strategies in cloud computing and use metrics to determine a cost-effective solution.

In this paper, we compare different microservice architecture patterns using Amazon Web Services (AWS) by monitoring and measuring specific metrics that are important for our timetabling software. The overall goal is to find the best deployment pattern.

II. Related Works

In building an end-to-end pipeline application for scheduling software, we considered two different approaches about the application's environment. The first approach would be a desktop-based application running on the client side that would be locally installed. The second would be a cloud-based approach where the software would run on remote servers accessible to users via a web browser.

One of the most important factors in cloud service performance is architecture. It is crucial to cloud applications' availability, consistency, scalability and security [6]–[8]. A monolithic architecture has vertical scale benefits, but a single change in the system will impact all users [9]. In a large on-demand application, the system must handle multiple users' load without downtime. With microservices, the services each fulfill a single function in the application [10]. Microservices scale well because they can horizontally scale and are loosely coupled, so they are fault-tolerant and isolated [9]. Although they are recommended for large-scale applications, incorporating them into smaller ones will bring ease to future scaling.

Moreover, there are several patterns to deploy a microservice where each pattern comes with its tradeoffs and cost structure [11]. Traditionally, developers are responsible for managing and scaling when deploying microservices on servers. A serverless framework allows deployment without managing infrastructure, where cloud providers allocate resources on demand without any user overhead.

To further clarify the gap we are exploring, in the next sections, we discuss how to select the best way of deploying microservice models and developing metrics that meet our particular, school-based, business needs.

III. Problem Statement

Timetabling applications have been mostly algorithm-centric and developed as a single unified unit, described as monolithic architecture. When applications with monolithic architectures grow too large, scaling becomes a challenge because individual services cannot be scaled in isolation [12]. The development speeds become slower as any change to one component would require testing the whole application as they are in a unified codebase. Also, an error in one module might affect the availability of the entire application [12].

As referenced in the related works section, microservice architecture solves these issues when the application has to scale. Even though microservices have become the standard practice in most software architectures, they can be deployed

979-8-3503-7099-7/24/$31.00 ©2024 IEEE

in multiple ways. We discuss monolithic architecture as a baseline solution, as well as different deployment patterns of microservices below.

A. Pattern 1: Monolithic

The monolithic architecture is a traditional model of software program that generally has one large codebase that couples all parts of the application together, as shown in Fig. 1. Sometimes this architecture is preferred due to ease of installation, more straightforward configuration, and less cross-service debugging [13].

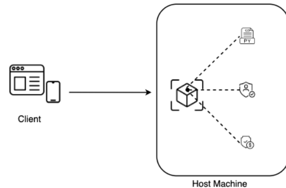

Fig. 1. Pattern 1: Monolithic Architecture

B. Pattern 2: One Host, Multiple Services

In this pattern, as shown in Fig. 2, all the service instances are deployed on a single host on multiple ports. The host can either be a Virtual Machine or a physical server [11]. This approach has certain benefits and drawbacks. Scaling up would require us to copy the service to another host and start it [14], and it is also relatively fast to start as it has very little overhead. The resource utilization is also fairly efficient as all the services share the server and its OS [14]. One drawback of this approach is that there is no isolation of the service instances, as we cannot limit the resources each instance uses.

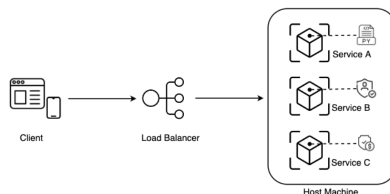

Fig. 2. Architecture Pattern 2: One Host, Multiple Services

C. Pattern 3: One Host, One Service (Virtual Images)

In this pattern, as shown in Fig. 3, instead of having all the services in a single host, we package each service in its host [15]. Each host machine will run a single service packaged in the form of virtual images. This allows greater isolation between services and overcomes the drawback of services competing for common resources. Deployment in this pattern is reliable and robust, as each service is interoperable, immutable and easy to monitor. One drawback of this approach is that deployment is slow, as virtual images contain operating system and is slower to deploy.

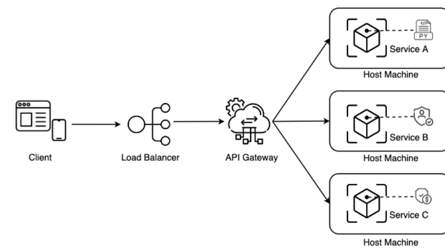

Fig. 3. Architecture Pattern 3: One Host, One Service

D. Pattern 4: One Host, One Service (Containers)

In this pattern, we have one service running on one host, as shown in Fig. 4. The service environment inside the host is completely isolated by running the service inside a container [16]. While running services packaged as virtual images works, they are heavy to deploy as they contain an operating system along with the code. A container wraps the service and all its dependencies but does not contain an operating system. It shares the kernel with the host machine, which makes deployment extremely fast [17].

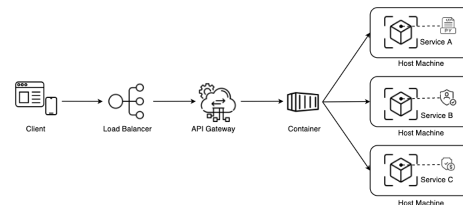

Fig. 4. Architecture Pattern 4: One Host, One Service, with Container

E. Pattern 5: Serverless Deployment

All the abovementioned patterns require manual cloud infrastructure management after deployment [18], [19]. For example, suppose there is a sudden spike in the number of users for the timetable service. This is particularly visible for timetabling applications because of the seasonal load. There are a lot of schools using the timetable service in the summer before the school term starts and very little load in other periods. In that case, we need to manually scale up the number of servers to accommodate the increased demand. To avoid this, the serverless architecture pattern provides a scalable and reliable approach to deployment that does not require any manual involvement in infrastructure management [18]. As shown in Fig. 5, the services are deployed as functions.

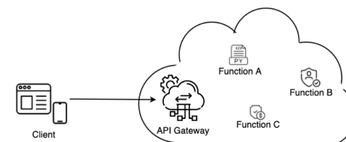

Fig. 5. Architecture Pattern 5: Serverless Deployment

Once the application is deployed, the responsibility of managing the cloud infrastructure falls on the cloud provider and not on the developers. One drawback of this approach is the limited runtime of each service and cold starts.

IV. Evaluation

Each microservice architecture pattern is different and has its own set of drawbacks. We will compare them and see which is suitable and meets the criteria for the timetabling application. We need a common evaluation metric to compare these different architectural patterns. We believe the important parameters to compare are the cost per user and the system's reliability. All cloud providers charge based on the number of hits the server receives. More hits imply that we have more users of our application. To compute the cost per user, we divide the total cost incurred by the number of users of our application. The system is said to be reliable if it is always available to perform the services it is designed for. Reliability is an important factor to consider because it ensures that the application is available to the users when needed and there is no downtime [20]. Reliability in cloud computing is measured by comparing the failure rate of all the components in the architectural pattern. To compute the reliability factor, we will send n requests to the servers and check how many of those requests are responded to efficiently by the system. The value for the reliability factor is computed using (1). For example, if the architecture can only respond to 5 out of 10 requests within the stipulated time, the reliability factor will be 0.5.

$$reliability factor = \frac{responded\, requests}{sent\, requests} \tag{1}$$

The reliability factor should be high for serverless architecture and low for single-server architecture. Considering these two factors, the evaluation metric we wanted to compare will be measured using (2).

$$Metric = \frac{total\ cost}{number\ of\ users} * \frac{1}{reliability\ factor} \tag{2}$$

In the next sections, we identify the architectural pattern and the cloud provider that yields the lowest metric value.

V. System Design

To test the deployments, we developed a cloud-based automated timetabling application that generates a master school timetable. The system design of the application is shown in Fig. 6. The frontend of the application is built with React. The communication to the backend happens through REST APIs.

The Node.js server in the backend acts as the API gateway. This conforms with the facade design pattern where the client communicates with a single server. Node.js uses non-blocking and event-driven architecture, making it efficient and suitable for microservices [21]. This Node.js server interacts with the timetable and authentication servers for the required operations.

The frontend input is an Excel file with timetable structure and constraints. This is converted to a JSON object and sent to the Python server that generates the timetable and returns it as a JSON object. The server sends this response to the frontend, displaying it to the user. We used MongoDB for our database, which better suited our needs due to its horizontal scalability, absence of SQL normalization, and dynamic schema [22].

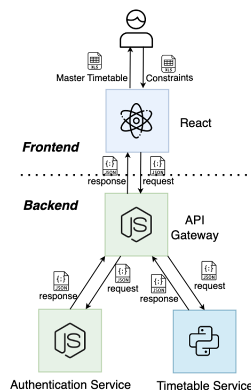

Fig. 6. Basic System Design

A. Pattern 1 Deployment

As a baseline, we run a single server on an EC2 instance in our monolithic pattern, as shown in Fig. 7. We adapt our basic system design to a monolithic pattern by combining the functionalities from the authentication server (runs on Node.js) and the Flask server to our main Node.js server. We ran our Python timetabling code (previously on the Flask server) by spawning a child process from the main Node.js server and then called the Python code in the child process. This approach was less straightforward than our basic system design and not technology agnostic, so there may be a better approach with other architectures.

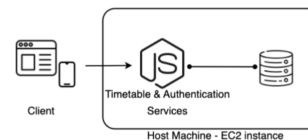

Fig. 7. Deployment Pattern 1: Monolithic

B. Pattern 2 Deployment

In a one-host-multiple-services pattern, the system design of this deployment pattern is very similar to our basic system design. As shown in Fig. 8, we create an EC2 instance as the host machine, and all services will be deployed and run on a single host machine. There will be no overhead communication since they are all inside the same host machine. In this deployment pattern, we also adopt an Application Load Balancer (ALB) in case of future scaling. The ALB will route requests to the desired server.

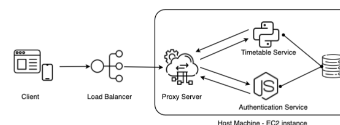

Fig. 8. Deployment Pattern 2: One Host, Multiple Services

In the backend, we keep the Flask server and Node.js server separate to generate timetable service and perform authentication service. We installed a Node.js server that acts

as an API proxy. The API proxy server would intercept any incoming requests, and based on the type of requests, it will forward them to the respective services. Timetable and authentication services listen to a different port and communicate over the API proxy server. The master timetable generated from the timetabling service and login token generated from the authentication service will be stored in the MongoDB database deployed to the same EC2 instance.

C. Pattern 3 Deployment

In this pattern, each microservice will be deployed on Amazon EC2 by creating a virtual instance for each of the microservices. The routing server, which is written in Node.js, will act as the API Gateway or the proxy server that communicates between the Authentication server and the Timetabling server. For this deployment pattern, each host machine will have its own image of the microservice that responds to a designated feature required by the application. The client accessing the tool will only communicate to the API-Gateway (Node.js) server, ensuring incoming requests are tokenized, thereby making the entire architecture a black-box. The proxy server will also ensure each incoming request is tokenized after properly being authenticated using a MongoDB database before granting access to the time-tabling service. In this way, each microservice is independently deployed and remains scalable and responsive. This pattern architecture is illustrated in Fig. 9.

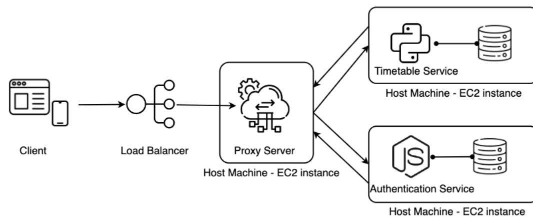

Fig. 9. Deployment Pattern 3: One Host, One Service

D. Pattern 4 Deployment

In this pattern, microservices will be deployed using containers that provide the perfect environment for running small independent services. Containers have the code, runtime, system tools, libraries, and settings to run microservices. As shown in Fig. 10, each virtual host machine will contain a single container running a single microservice. We will use Docker for building and managing containers. As containers are an independent unit of software and do not contain overheads of operative systems, they can be deployed to any number of servers relatively fast. As the number of microservices grows, so will the containers. Managing the number of containers as the project grows manually will become harder.

We use Kubernetes - an open-source container orchestration tool developed by Google to manage containerized applications. A Kubernetes cluster offers a high availability of containers and provides provisions for scalability and fault tolerance. The most important component inside a cluster

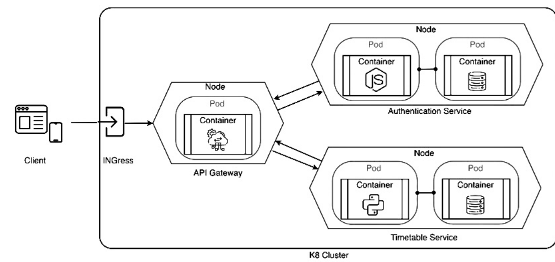

Fig. 10. Deployment Pattern 4: One Host, One Service (Containers)

is Nodes and Pods. Nodes are either physical machines or virtual machines. Pods are the smallest units of Kubernetes that provide a layer of abstraction over the container and reside inside a node. In our architecture, each service will be deployed inside a node with pods for application code and database. The ingress component in Kubernetes acts as a load balancer. A client will send the request to ingress, which forwards the request to respective pods. Kubernetes cluster is deployed on Amazon Elastic Kubernetes Service with nodes running on EC2 instances.

E. Pattern 5 Deployment

In this pattern, the services are packaged and deployed on a serverless platform as serverless functions. The responsibility of configuring and managing the host machines falls on the cloud provider as they automatically assign and scale the required number of machines to handle the demand. Deploying in a serverless architecture requires us to convert each of the servers to serverless functions that are compatible to be deployed. There are two approaches to deploying serverless functions in AWS. The first approach is to create and deploy the serverless functions directly. This can be done by exposing the services as modules which are then hosted on the serverless platform. The other approach is to convert these functions to container images and deploy these container images as serverless functions. Deploying the application as container images incur additional costs as they have to be pushed in a private ECR (Elastic Container Registry) repository for it to be accessible by the serverless platform. For our analysis, we wanted to pick the route that incurs a lower cost, so we chose the former.

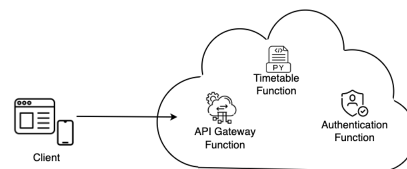

Fig. 11. Deployment Pattern 5: Serverless

For deploying the application as a serverless function, we first converted the auth service and the timetable service into serverless functions by exporting the server component as a module (see Fig.!11. To access these functions, we wrapped these modules using an API gateway so that we can send get or post requests to the functions to perform the required

operations. In addition to that, we also converted the Node.js server, which acts as the proxy server, to a serverless function so that the client has just a single server URL to keep track of. This proxy server, along with the API gateway, forms the API gateway function. This function communicates with the other functions based on requests from the client.

VI. Simulations

We used the JMeter tool to run our simulations and compute the reliability factor based on our architectures' results. A test plan consisting of two endpoints simulates load on all architectural patterns. In this test plan, the endpoint "Schedulett" is used for timetable generation by timetable service and the endpoint "Login" is used for signing a user in the platform by authentication service. The simulation is run in a multithreaded environment, allowing the simulation load on both endpoints parallelly. For the simulation, we are sending a total of 100,000 requests to both the authentication service and the timetable service to see how many of these requests are responded to in a reasonable time. To account for the instability of the network in access to these cloud deployments, we consider it a successful response if the server is able to respond within twice the average latency the server takes with less load. We computed the average latency in ms by performing both the scheduled timetable and login operations with the parameters of 1 thread (users) and 1000 iterations. This average latency is presented in Table I.

TABLE I
Average latency (ms)

Pattern	Login	Schedulett
Pattern 1	302	750
Pattern 2	257	512
Pattern 3	253	544
Pattern 4	151	410
Pattern 5	1556	2740

We then multiplied this latency by two and set this as our duration assertion (timeout) for the "Schedulett" and "Login" services respectively. We ran simulations with 50 threads (users) and 1000 iterations each. We then counted the number of successful responses out of the total requests for both timetable and authentication service. The number of successful responses is specified in Table II. This information was used to determine the reliability factor, using our formula mentioned in the evaluation section.

We computed the cost using the AWS pricing calculator. All patterns 1, 2 and 3 are running on t2.small EC2 instances. We get the cost for these patterns by feeding the above-mentioned configuration to the AWS pricing calculator, the results for which are shown in Table III. The costs computed using the AWS Pricing calculator take into account the number of users accessing the services as well.

Once we have the reliability factor and the cost, we compute the metric value (Table III). As the computed cost already takes into account the number of users accessing the service, we need not divide the cost again by the number of users. Therefore, the metric value will be the product of cost and the inverse of the reliability factor.

TABLE II
Successful Responses (out of 50,000 requests each)

Pattern	Login	Schedulett	Total (100,000)
Pattern 1	400	33100	33500
Pattern 2	10600	2800	13400
Pattern 3	50000	2100	52100
Pattern 4	45600	1800	47400
Pattern 5	49600	49500	99100

TABLE III
Evaluation results (lower evaluation metric is better)

Pattern	Reliability factor	Cost/month (USD)	Evaluation Metric
Pattern 1	0.335	9.27	27.67
Pattern 2	0.076	9.27	69.18
Pattern 3	0.521	42.86	82.27
Pattern 4	0.474	147.37	310.91
Pattern 5	0.991	22.52	22.73

The initial results in Table III emphasize too much on the cost as the cost matters a lot for the smaller companies. For much larger companies, the reliability of service would take a higher priority than the cost. To reduce the impact of cost, we normalized it using an inverse logarithmic function so that all the lower values of cost converge to a small number, but the higher cost values are exponentially high. These results are specified in Table IV.

TABLE IV
Normalized metric value (lower evaluation metric is better)

Pattern	Reliability factor	Normalized Cost/month (USD)	Evaluation Metric
Pattern 1	0.335	1.018	3.041
Pattern 2	0.076	1.018	7.602
Pattern 3	0.521	1.089	2.091
Pattern 4	0.473	1.343	2.833
Pattern 5	0.991	1.046	1.055

A. Analysis

Pattern 1 (Monolith Architecture) is easy to maintain and is extremely useful in the early stages of application development when the number of users is not high. When the number of users becomes higher, the monolith becomes less reliable. Pattern 1 is still more reliable than Pattern 2 (Multiple servers in 1 Host) because Pattern 1 does not require any inter-service communications. This additional time, along with the fact that all services share the same processing resources, makes Pattern 2 the least reliable.

Having just a single service running in a host improves the reliability of Pattern 3 and Pattern 4 as the services are not clashing for the same system resources. However, this improved reliability does not compensate for the increased cost. For our use case, deploying using containers for Pattern 4 seems like an overkill which can be inferred from the high metric value for this pattern. Even though this pattern seems like a bad option for our business needs in smaller companies, it can still be used by larger companies where cost does not matter much. Using container images for the deployment makes it the easiest to scale. The higher reliability of these patterns makes it a much better option than Pattern 1,2 for larger companies as the metric value indicates in Table IV.

Pattern 5 (serverless architecture) seems to be the most reliable as the deployments are scaled automatically to handle incoming requests. The cost for serverless is not as high as the other patterns; AWS Lambda charges just for the time the function is actually running and not for the entire duration the server is active. This high reliability and intermediate cost make it the most suitable option for our deployments. The only drawback of this architecture is that it has a lower timeout and higher latency than EC2 instances. Based on our results, Pattern 5 seems to be the most preferred deployment pattern as the higher latency is not a large factor for our business needs.

VII. Conclusion

This paper explored five different architectures for deploying timetabling software on AWS to find the one that best suits our business needs. For this purpose, we developed a metric formula based on cost-per-performance and reliability factors, and our goal was to find the architecture that produced the lowest metric value in the simulation plan. As the results of our simulations show, Pattern 5, the serverless mode, has clear advantages due to its high reliability and elastic cost per performance. We believe this pattern is the best choice for timetabling application businesses.

Our deployment simulations were conducted exclusively on the Amazon Web Services (AWS) cloud platform, which means that the performance metrics we measured—such as cost, uptime, and speed—are inherently tailored to the characteristics and limitations of AWS's infrastructure. This includes, for instance, data on EC2 instance performance, S3 storage reliability, and the latency of AWS's global content delivery network. If interested, this approach can be applied to other cloud platforms to explore the differences. On the other hand, our metrics and simulation plans are designed to meet the business needs of start-up timetabling companies that have smaller customer bases. For companies with larger customer bases and functional endpoints, factors such as scalability, maintainability, and fault tolerance can be added to the metric formula and simulation plans can be adjusted accordingly to the business needs.

References

[1] R. Hoshino, J. Albers. Automating school timetabling with constraint programming, Northeastern University 2022, unpublished.

[2] Constantia, Charalampous and Christos, Papademetriou and Glykeria, Reppa and Anastasia, Athanasoula-Reppa and Aikaterini, Voulgari. The impact of covid-19 on the educational process: the role of the school principal, Journal of Education, 2021, SAGE Publications Sage CA: Los Angeles, CA.

[3] Applied Software Consultants. accessed 2022-09-21, https://www.asctimetables.com.

[4] Lantiv. Lantiv, accessed 2022-08-03, https://lantiv.com, 2021.

[5] Padilla, Jose J. and Diallo, Saikou Y. and Barraco, Anthony and Lynch, Christopher J. and Kavak, Hamdi. Cloud-based simulators: making simulations accessible to non-experts and experts alike, Proceedings of the Winter Simulation Conference 2014. Pages 3630-3639.

[6] Vural, Hulya and Koyuncu, Murat and Guney, Sinem. A systematic literature review on microservices, computational science and its applications – ICCSA 2017, Springer International Publishing

[7] Aibin, Michał, Krzysztof Walkowiak, and Arunabha Sen. "Software-defined adaptive survivability for elastic optical networks." Optical Switching and Networking 23 (2017): 85-96.

[8] M. Aibin, R. Goścień and K. Walkowiak, "Multicasting versus anycasting: How to efficiently deliver content in elastic optical networks," 2016 18th International Conference on Transparent Optical Networks (ICTON), Trento, Italy, 2016, pp. 1-4, doi: 10.1109/ICTON.2016.7550657.

[9] Blinowski, Grzegorz and Ojdowska, Anna and Przybyłek, Adam. monolithic vs. microservice architecture: a performance and scalability evaluation," IEEE Access 2022.

[10] R. Barcia, K. Brown, and R. Osowski. Microservices point of view, 2018. accessed 2022, https://www.ibm.com/downloads/cas/ORNMY

[11] Saraswat, Chris. Choosing a microservices deployment strategy, NGINX 2016.

[12] Villamizar, M., Garcés, O., Castro, H., Verano, M., Salamanca, L., Casallas, R., and Gil, S. (2015, September). Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. In 2015 10th Computing Colombian Conference (10CCC) (pp. 583-590). IEEE.

[13] Mendonça, N. C., Box, C., Manolache, C., and Ryan, L. (2021). The monolith strikes back: Why istio migrated from microservices to a monolithic architecture. IEEE software, 38(5), 17-22.

[14] De Lauretis, L. (2019, October). From monolithic architecture to microservices architecture. In 2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW) (pp. 93-96). IEEE.

[15] Kanjilal, Joydip, Deployment patterns in microservices architecture. accessed 2022 https://www.developer.com/design/deployment-patterns-microservices/, Developer.com

[16] Srirama, S. N., Adhikari, M., and Paul, S. (2020). Application deployment using containers with auto-scaling for microservices in cloud environment. Journal of Network and Computer Applications, 160, 102629.

[17] Docker Container, accessed October 2022 https://www.docker.com/resources/what-container/

[18] Lynn, T., Rosati, P., Lejeune, A., and Emeakaroha, V. (2017, December). A preliminary review of enterprise serverless cloud computing (function-as-a-service) platforms. In 2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom) (pp. 162-169). IEEE.

[19] Aibin, Michal, and Krzysztof Walkowiak. "Resource requirements in fixed-grid and flex-grid networks for dynamic provisioning of data center traffic." 2016 IEEE Canadian Conference on Electrical and Computer Engineering (CCECE). IEEE, 2016.

[20] N. K. K. Kit and M. Aibin, "Study on High Availability and Fault Tolerance," 2023 International Conference on Computing, Networking and Communications (ICNC), Honolulu, HI, USA, 2023, pp. 77-82, doi: 10.1109/ICNC57223.2023.10074557.

[21] Gilad David Mayaan, Reasons to build microservices with Node.js, https://bambooagile.eu/insights/microservices-node-js/, accessed October 2022

[22] Anderson, Benjamin and Nicholson, Brad. SQL vs. NoSQL databases: what's the difference, IBM. https://www. ibm. com/cloud/blog/sql-vsnosql 2021.