Analysis of Infrastructure Cost and Reliability of Architecture Patterns for Deploying a Timetabling Application on Cloud

Sommer Harris, Jaelyn Ma, Meet Patel, Kshitij Nair, Niranjan Velraj, Michal Aibin *Member, IEEE*
Khoury College of Computer Sciences
Northeastern University, Vancouver, Canada
{harris.som | ma.du | patel.meet2 | nair.ks | velraj.n | m.aibin}@northeastern.edu

Abstract—There is no standardized way for businesses to deploy their application on Cloud. The suitable deployment strategy depends on the type of application and the specific requirements of the business. In building a timetable application, we test five deployment architectures and perform an evaluation based on metrics such as reliability factor and cost per month to determine which one best suits our business requiring relatively high system reliability and a low-performance cost model. The result of our study shows that out of our different architectures, the one most suitable for deploying the business app on AWS cloud deployment will be the serverless architecture. This study may act as an example of how researchers may develop their own tests and metrics in other business cases.

Index Terms—cloud deployment, microservices, timetabling.

1 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 few standalone educational timetabling software products like aSc Timetables [3] and Lantiv [4] [5]. Over decades, these applications have attempted accommodating the growing constraints, but in [6], it was found that there is a gap with timetabling software being accessible or usable only by experts.

The long term vision of this project is to further bridge the gap between research and industry by developing a cloud based timetable application based on the novel timetabling solution developed in Hoshino 2022 [1]. 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 in the most cost-effective manner. To address accessibility and a larger number of consumers, we also explore in this paper various scalability strategies in cloud computing and use metrics (discussed in Section 4) to determine a cost-effective solution.

In this paper, we compare different microservice architecture patterns on cloud provider AWS by monitoring and measuring specific metrics that are important for our timetabling software. We compare various architectures using our own cloud based timetabling application. We aim to find the best deployment pattern using evaluation metrics that suits the business needs of a timetabling application.

In the remainder of this paper, we will first review works related to assessing cloud deployment architectures, then provide a formulated problem statement that includes the architectures we plan to test, as well as our metric of assessment. Next, we will show the preliminary system design for our application, and explore how it was modified for deployment for each of the architectures. Lastly, we will outline our simulation setup, and share our results, analysis, and concluding remarks.

2 Related Works

This section reviews research on advantages and assessment of the cloud based solution as compared to standalone applications. We also review the significance of architecture in cloud deployment and the use of microservices. We discuss AWS services for deployment, and open ended nature of metrics for evaluating architectures. We end this section by further clarifying the gap we cover in our approach.

In building an end-to-end pipeline application for a scheduling software, we considered two different approaches pertaining to the environment of the application. First approach would be a desktop-based application running on the client-side that would be locally installed and second would be a cloud-based approach where the software would run on remote servers accessible to users via a web browser. We draw comparisons between the two approaches in Appendix A [1] [7]–[11].

1. Appendix A displays architecture feature analysis in a table.

Through a comparative study, Rajendran and Swamynathan 2016 generated a list of attributes that impact cloud service performance. They also state that choosing a provider is a multidimensional problem and users must find an intelligent way to select the best service based on their app's unique needs. They outline functional and nonfunctional metrics that can be used to consider providers [12].

One of the most important factors in cloud service performance is Architecture. It is crucial to availability, consistency, and scalability of cloud applications [13]. A monolithic architecture has vertical scale benefits, but a single change in the system will impact all users [14]. In a large on-demand application, the system must be able to handle the load of multiple users without downtime [14]. With Microservices, the services each fulfill a single function in the application [15]. Microservices scale well because they can horizontally scale and are loosely coupled so are fault-tolerant and isolated [14]. Although microservices are often recommended for large-scale applications, incorporating microservices to smaller applications will bring ease to future scaling [14].

There are several patterns to deploy a microservice where each pattern comes with its own tradeoffs and cost structure. [16] Traditionally, When deploying microservices on servers, the responsibility to manage and scale is on developers. Serverless framework allows deployment without managing infrastructure, where resources are allocated on demand by cloud providers without any user overhead.

Existing comparisons of the cloud providers generally tend to favor AWS for its broader offerings and faster response times [17] [18]. As the biggest cloud service provider, AWS provides multiple solutions for microservices deployment, such as EC2, Amazon Elastic Container Service (Amazon ECS), Amazon Elastic Kubernetes Services (Amazon EKS) and AWS Lambda. [19] Amazon EC2 as a Infrastructure-as-a-Service(IaaS) provider deploys microservices on virtual machines [20]; Amazon ECS provides Docker container management services [21]; EKS deploys containerized application while provisioning Kubernete management services [22]; AWS Lambda deploys microservices in serverless computing technology [23].

There are no standard metrics that can be used by all applications to evaluate a cloud architectural pattern [24] [25]. The evaluation parameters vary depending on the application being deployed and the way it is hosted and identifying the most suitable cloud provider requires a good understanding of the application architecture. Leitner et. al, 2016 present an approach to developer tooling that models cloud deployment costs for microservices, and also claim that more studies are needed that explore what cost related questions developers are trying to answer [24]. Saraswat and Tripathi, 2020 compare and analyze parameters of cloud service providers AWS, Microsoft and Google. Ultimately, they claim the choice on how to deploy to the cloud will depend on technical and business needs of each particular company. Rajendran and Swamynathan, 2022 analyze parameters for cloud service providers also state that there is an absence of standards and frameworks for assessing cloud services and that a key step is that clients must identify and evaluate what is necessary to their application.

Timetabling software is different from most other hosted applications because generating a timetable is processor intensive. The scale and load are seasonal as there may be a greater number of users during summer compared to other periods and microservices should scale up/scale down accordingly. Considering the unique demands of timetabling applications, we felt that comparing the cost-to-performance ratio of deploying it using different architectural patterns will be necessary to identify the best hosting solution.

To further clarify the gap we are exploring, we hope to share how we modeled our particular application needs to develop our own cloud deployment tests, as a step toward contributing to the knowledge gap on how to select the best way of deploying microservice models and developing metrics that meet our particular business needs. We focus on exploring the best architecture pattern for hosting microservices for our timetable scheduling application based on proper metrics to performance.

3 Problem Statement

Timetabling applications to date 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 [26]. The development speeds become slower as any change to one component would require testing of the whole application as they are in a unified codebase. Also, if there is an error in one module, it might affect the availability of the entire application [26].

As referenced in the related works section, microservice architecture solves these issues when the application has to scale. It is an architectural method that emphasizes modularity by using independently deployable modules rather than a single unified application. This reduces coupling between different modules because each microservice becomes an independent unit of development, deployment and versioning. This is particularly crucial for timetabling applications as it gives developers the freedom to use different programming languages for the front end of the application and the actual algorithm.

Even though microservices have become the standard practice in most software architectures, there are multiple ways they can be deployed. We discuss monolithic architecture, as well as different deployment patterns of microservices below:

3.1 Pattern 1 : Monolithic

This monolithic architecture is a traditional model of software program which 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 installations, more straightforward configuration, and less cross-service debugging [27].

3.2 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 [16].

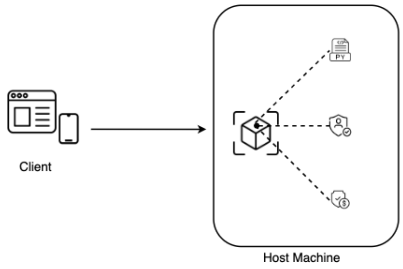

Fig. 1. Pattern 1: Monolithic Architecture

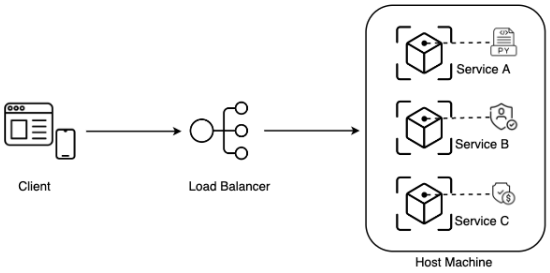

Fig. 2. Architecture Pattern 2: One Host, Multiple Services

This approach has certain benefits and drawbacks. Scaling up would just require us to copy the service to another host and start it [28] and it is also relatively fast to startup as it has very little overhead. The resource utilization is also fairly efficient as all the services share the server and its OS [28]. One drawback of this approach is that there is no isolation of the service instances as we cannot limit the resources each instance uses [16].

3.3 Pattern 3 : One Host, One Service (Virtual Images)

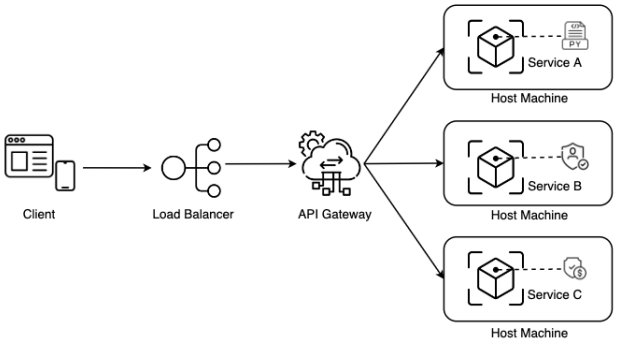

Fig. 3. Architecture Pattern 3: One Host, One Service

In this pattern, instead of having all the services in a single host, we package each service in its own host [29]. As shown in Fig. 3, each host machine will run a single service packaged in 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. [16]

3.4 Pattern 4 : One Host, One Service (Containers)

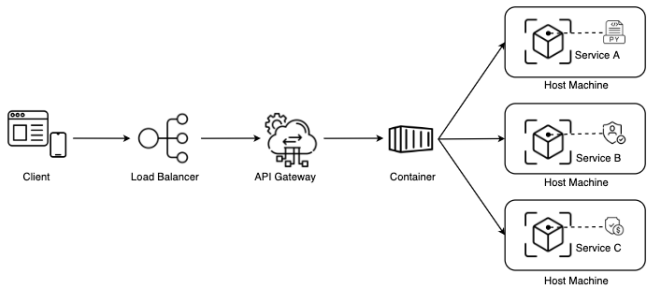

Fig. 4. Architecture Pattern 4: One Host, One Service, with Container

In this pattern, similar to Pattern 4, we have one service running on one host. The service environment inside the host is completely isolated by running the service inside a container [30]. While running services packaged as virtual images works, they are heavy to deploy as they contain operating system along with the code. As shown in Fig. 4, a container wraps the service and all its dependencies but does not contain operating system. It shares the kernel with the host machine which makes deployment extremely fast [31].

3.5 Pattern 5 : Serverless Deployment

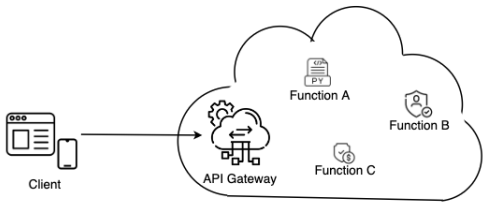

Fig. 5. Architecture Pattern 5: Serverless Deployment

All the patterns discussed above require manual cloud infrastructure management after deployment [32]. For example, if there is a sudden spike in the number of users for the timetable service, then we need to manually scale up the number of servers it is deployed in to accommodate the increased demand. Scaling up and scaling down servers is needed to manage costs and also to ensure that the functions are accessible. This is particularly important for our timetabling application because of the seasonal load and there will be a lot of schools using the timetable service in the summer before the school term starts and very less load in other periods. To manage the cloud infrastructure for scaling up and scaling down, companies have a team of developers who need to take time off from developing new features to ensure that the application is always accessible in the cloud. To avoid this, the Serverless architecture pattern provides a self-maintaining, scalable, and reliable approach to deployment that does not require any manual involvement in infrastructure management [32]. Once the application is deployed in serverless architecture, the responsibility of managing the cloud infrastructure falls on the cloud provider and not on the developers as shown in Fig. 5. One drawback of this approach is limited runtime of each service and cold starts.

4 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 timetabling app. 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 reliability of the system. All cloud providers charge based on the number of hits received by the server. 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 they need it and there is no downtime. 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. For example, if the single server architecture is only able to respond to 5 out of 10 requests within the stipulated time, the reliability factor will be 0.5 (5/10).

$$reliability factor = \frac{responded\ requests}{sent\ requests}$$

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:

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

In the future sections, we will try to identify the architectural pattern and the cloud provider that yields the lowest metric value.

5 System Design

To test the deployments, we developed a cloud-based Automated timetabling application that generates a master timetable, as shown in Fig. 6, The front end of the application is built with React. There are UI libraries to build custom components in React with ease and has a Virtual DOM that makes the rendering faster by updating the content in the DOM [33]. The communication to the Backend happens through REST APIs.

The Node.js server in the backend acts as the API gateway. This is to conform with the facade design pattern where the client communicates with a single server. Node.js uses non-blocking and event-driven architecture which makes it efficient and suitable for microservices [34]. This Node.js server interacts with both the timetable server and authentication server 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 where it is displayed to the user. We used MongoDB for our database. We used NoSQL, which better suited our needs due to its horizontal scalability, absence of SQL normalization, and dynamic schema [35]–[37].

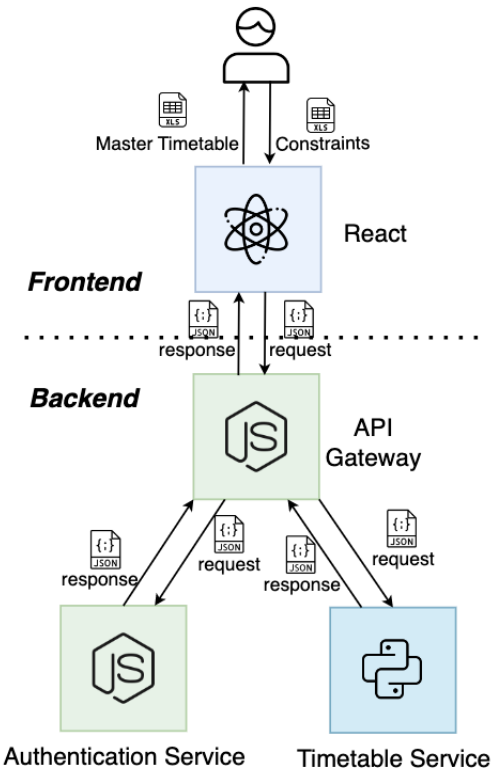

Fig. 6. Basic System Design

5.1 Pattern 1 Deployment

In our monolithic pattern, we run a single server on an EC2 instance. 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, as shown in Fig. 7. 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, and not technology agnostic, so there may be a better approach with other architectures.

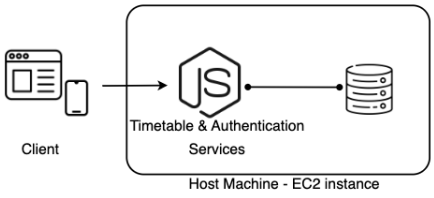

Fig. 7. Deployment Pattern 1: Monolithic

5.2 Pattern 2 Deployment

In a one-host-multiple-services pattern, 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 of 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.

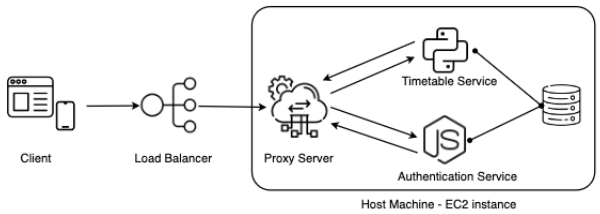

Fig. 8. Deployment Pattern 2: One Host, Multiple Services

In the backend, we keep the Flask server and Node.js server separate to generate timetable service and perform authentication service respectively. We installed a Node.js server that acts as an API proxy. Any incoming requests would be intercepted by API proxy server and based on the type of requests, it will forward them to the respective services. Timetable service and authentication services listen to a different port and communicate over the API proxy server. Master timetable generated from timetabling service and login token generated from authentication service will be both stored in MongoDB database that is deployed to the same EC2 instance.

5.3 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 EC2 instances use the t2.micro machines running Amazon's own distribution of Linux, AWS Linux. 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, which will make sure incoming requests are tokenized, thereby making the entire architecture a black-box. The proxy server will also make sure 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 in also illustrated in Fig. 9.

5.4 Pattern 4 Deployment

In this pattern, microservices will be deployed using containers that provides perfect environment for running small independent services. Containers has the code, runtime, system tools, system libraries and settings to run microservices. For this pattern, each virtual host machine will contain a single container running a single microservice, as shown

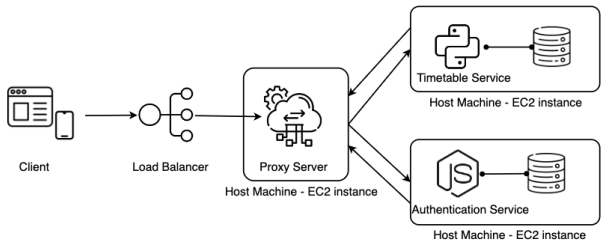

Fig. 9. Deployment Pattern 3: One Host, One Service

in Fig. 10. We will use Docker for building and managing containers. As containers are independent unit of software and does not contain overheads of operative system, they can be deployed to any number of server relatively fast. As the number of microservices grow, so will the containers. It will become harder to manually manage the number of container as the project grows.

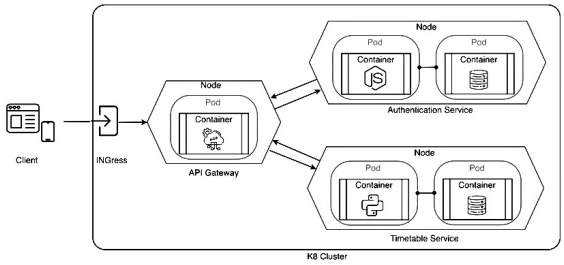

Fig. 10. Deployment Pattern 4: One Host, One Services (Containers)

We use Kubernetes - a open source container orchestration tool developed by Google to manage containerized application. A Kubernetes cluster offers high availability of containers, provides provisions for scalility and fault tolerance. The most important component inside a cluster is Nodes and Pods. Nodes are either the physical machine or virtual machines. Pods are the smallest units of kubernetes that provides a layer of abstraction over container and reside inside a Node. In our architecture, each service will be deployed inside a Node which will have Pods for application code and database. Ingress component in Kubernetes acts as 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.

5.5 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, as shown in Fig. 11.

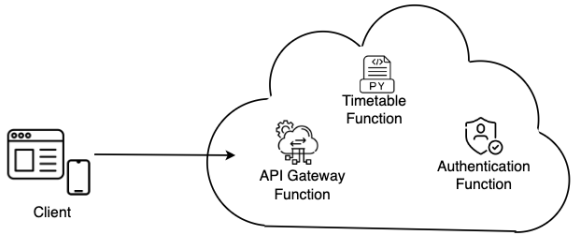

Fig. 11. Deployment Pattern 5: Serverless

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. 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.

6 Simulation

6.1 Setup

We used JMeter tool to run our simulations and compute the reliability factor based on the results for each of our architectures. A test plan consisting of two end points is used to simulate 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 ran in multithreaded environment allowing to simulate load on both the endpoints parallelly. For the simulation, we are sending a total of 1000 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 schedule timetable and login operations with the parameters of 1 thread (users) and a 100 iterations and documented the results in TABLE 1.

We then multiplied this latency by two and set this as our duration assertion (timeout) for the "Schedulett" and "Login" services respectively. We then ran simulations with 5 threads (users) and 100 iterations each. We then counted the number of successful responses out of the total requests for both timetable and authentication service, which are documented in TABLE 2. This information was used to determine the reliability factor, using our formula mentioned in the evaluation section.

TABLE 1
Average latency

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

TABLE 2
Successful Responses

Pattern	Out of 500 Requests		Total
	Login	Schedulett	
Pattern 1	4	331	335
Pattern 2	106	28	134
Pattern 3	500	21	521
Pattern 4	456	18	474
Pattern 5	496	495	991

We computed the cost using AWS pricing calculator. Pattern 1, 2 and 3 are running on t2.micro EC2 Instance with 1GB of memory, 1 vCPU and 8GB of storage and 1GB data transferred both inbound and outbound. We get the cost for these patterns by feeding the above mentioned configuration to AWS pricing calculator, the results for which are shown in TABLE 3. Pattern 4 uses t2.small instance as that's the minimum configuration required for running Kubernetes cluster that allows deploying more than 4 Pods in an instance and each Node reserves 80GB of storage. This significantly increases the cost for this pattern. The cost for Pattern 1-4 are dependent on the running time of instances and the bandwidth transferred from them. However, for the serverless architecture, the cost is computed by the total number of requests and the time taken to process each of these requests. We used the average latency previously computed and a 1000 requests for the computation. The costs computed using the AWS Pricing calculator takes into account the number of users accessing the services as well.

TABLE 3
Successful Responses
(out of 500 requests each endpoint)

Pattern	Login	Schedulett	Total(1000)
Pattern 1	4	331	335
Pattern 2	106	28	134
Pattern 3	500	21	521
Pattern 4	456	18	474
Pattern 5	496	495	991

TABLE 4
Evaluation Metric (lower 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

TABLE 5
Normalized metric value (lower is better)

Pattern		Normalized	
	Reliability Factor Factor	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

6.2 Results

Once we have the reliability factor and the cost, we computed the metric value (in TABLE 3). 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 reliability factor.

The initial results in TABLE 3 emphasizes 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 converges to a small number, but the higher cost values are exponentially high. These results are specified in TABLE 4.

6.3 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 become higher, Monolith becomes less reliable. The Pattern 1 is still more reliable than Pattern 2 (Multiple servers in 1 Host) because Pattern 1 does not require any inter services 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 for Pattern 3 and Pattern 4 as the services are not clashing for the same system resources. But this improved reliability does not compensate for the increased cost. For our use case, deploying using containers for Pattern 4 seems like an over kill which can be inferred from the high metric value for this pattern. Even though this pattern seems like a bad option for our business need in smaller companies, it can still be used by larger companies where cost does not matter much. Using containers 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 4.

Pattern 5 (Serverless architecture) seems to be the most reliable as the deployments are scaled automatically to handle the incoming requests. The cost for serverless is not as high as unlike 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 an intermediate cost makes it the most suitable option for our deployments. The only drawback of using this architecture is that it has a lower timeout and high latency than EC2 instances. As the higher latency is not a factor of consideration for our business needs, based on our results, Pattern 5 seems to be the most preferred deployment pattern.

7 Conclusion

In this paper we explored five different architectures for deploying a 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. To yield the metric values, we ran simulations on each deployment pattern with 5 threads and 100 iterations on both of the endpoints. As the results of our simulation plan 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 performed on the AWS platform, so the cost and reliability factors in our metrics were limited to AWS infrastructure. 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 considered in the metric formula and simulation plans can be redesigned accordingly.

References

[1] Jameson Albers Richard Hoshino. Automating school timetabling with constraint programming. 2022.

[2] Charalampous Constantia, Papademetriou Christos, Reppa Glykeria, Athanasoula-Reppa Anastasia, and Voulgari Aikaterini. The impact of covid-19 on the educational process: The role of the school principal. *Journal of Education*, page 00220574211032588, 2021.

[3] ASC. Applied software consultants. *https://www.asctimetables.com//home/features*.

[4] Lantiv. Lantiv. *[online]. Dostupno na: https://lantiv.com/*, 2021.

[5] Capterra. Helping businesses choose better software since 1999.

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

[7] Lewis Golightly, Victor Chang, Qianwen Ariel Xu, Xianghua Gao, and Ben SC Liu. Adoption of cloud computing as innovation in the organization. *International Journal of Engineering Business Management*, 14:184797902210939, 2022.

[8] The importance of scalability in software design. *Award-winning App Development Company*.

[9] Lance Keene. The top 8 reasons why you should convert your desktop app to a web app. *ASP.NET, .NET Developer, C Programmer, FileMaker, React, SQL Server, DotNetNuke*, Apr 2021.

[10] Xenia Labis. Cloud based software vs desktop software. *Syntactics Inc.*, Jun 2021.

[11] Rashmi Bhardwaj. Cloud app vs web app : Detailed comparison " network interview. *Network Interview*, Jul 2022.

[12] V Viji Rajendran and S Swamynathan. Parameters for comparing cloud service providers: a comprehensive analysis. In *2016 International Conference on Communication and Electronics Systems (ICCES)*, pages 1–5. IEEE, 2016.

[13] Hulya Vural, Murat Koyuncu, and Sinem Guney. A systematic literature review on microservices. In Osvaldo Gervasi, Beniamino Murgante, Sanjay Misra, Giuseppe Borruso, Carmelo M. Torre, Ana Maria A.C. Rocha, David Taniar, Bernady O. Apduhan, Elena Stankova, and Alfredo Cuzzocrea, editors, *Computational Science and Its Applications – ICCSA 2017*, pages 203–217, Cham, 2017. Springer International Publishing.

[14] Grzegorz Blinowski, Anna Ojdowska, and Adam Przybyłek. Monolithic vs. microservice architecture: A performance and scalability evaluation. *IEEE Access*, 10:20357–20374, 2022.

[15] K. Brown R. Barcia and R. Osowski. Ibm : Microservices point of view. *https://www.ibm.com/downloads/cas/ORNMYNRW [2022-09-21]*, 2018.

[16] Chris Richardson. Choosing a microservices deployment strategy, 2016.

[17] Manish Saraswat and RC Tripathi. Cloud computing: Comparison and analysis of cloud service providers-aws, microsoft and google. In *2020 9th International Conference System Modeling and Advancement in Research Trends (SMART)*, pages 281–285. IEEE, 2020.

[18] Erina Fika Noviani, Bayu Kembara, Bakti Anugrah Yudha Pratama, Dyah Ayu Permata Sari, Ary Mazharuddin Shiddiqi, and Bagus Jati Santoso. Performance analysis of aws and gcp cloud providers. In *2022 IEEE International Conference on Cybernetics and Computational Intelligence (CyberneticsCom)*, pages 236–241. IEEE, 2022.

[19] François Rouxel. A new way of deploying a microservice in AWS — linkedin.com. https://www.linkedin.com/pulse/new-way-deploying-microservice-aws-fran2021. [Accessed 29-Sep-2022].

[20] Hamzeh Khazaei, Cornel Barna, Nasim Beigi-Mohammadi, and Marin Litoiu. Efficiency analysis of provisioning microservices. In *2016 IEEE International conference on cloud computing technology and science (CloudCom)*, pages 261–268. IEEE, 2016.

[21] Desheng Liu, Hong Zhu, Chengzhi Xu, Ian Bayley, David Lightfoot, Mark Green, and Peter Marshall. Cide: An integrated development environment for microservices. In *2016 IEEE International Conference on Services Computing (SCC)*, pages 808–812, 2016.

[22] Shimon Ifrah. Deploy a containerized application with amazon eks. In *Deploy Containers on AWS*, pages 135–173. Springer, 2019.

[23] Pooyan Jamshidi, Claus Pahl, Nabor C. Mendonça, James Lewis, and Stefan Tilkov. Microservices: The journey so far and challenges ahead. *IEEE Software*, 35(3):24–35, 2018.

[24] Philipp Leitner, Jürgen Cito, and Emanuel Stöckli. Modelling and managing deployment costs of microservice-based cloud applications. In *Proceedings of the 9th International Conference on Utility and Cloud Computing*, pages 165–174, 2016.

[25] Maria Fazio, Antonio Celesti, Rajiv Ranjan, Chang Liu, Lydia Chen, and Massimo Villari. Open issues in scheduling microservices in the cloud. *IEEE Cloud Computing*, 3(5):81–88, 2016.

[26] Mario Villamizar, Oscar Garcés, Harold Castro, Mauricio Verano, Lorena Salamanca, Rubby Casallas, and Santiago Gil. Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. In *2015 10th Computing Colombian Conference (10CCC)*, pages 583–590. IEEE, 2015.

[27] Nabor C Mendonça, Craig Box, Costin Manolache, and Louis Ryan. The monolith strikes back: Why istio migrated from microservices to a monolithic architecture. *IEEE Software*, 38(05):17–22, 2021.

[28] Lorenzo De Lauretis. From monolithic architecture to microservices architecture. In *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 93–96. IEEE, 2019.

[29] Joydip Kanjilal. Deployment patterns in microservices architecture. *Developer.com*, Sep 2022.

[30] Satish Narayana Srirama, Mainak Adhikari, and Souvik Paul. Application deployment using containers with auto-scaling for microservices in cloud environment. *Journal of Network and Computer Applications*, 160:102629, 2020.

[31] Docker. Docker container. https://www.docker.com/resources/what-container/. [Accessed 12-Oct-2022].

[32] Theo Lynn, Pierangelo Rosati, Arnaud Lejeune, and Vincent Emeakaroha. 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)*, pages 162–169, 2017.

[33] Facebook. React js. https://reactjs.org/. [Accessed 12-Oct-2022].

[34] Gilad David Mayaan. Reasons to build microservices with node.js. https://bambooagile.eu/insights/microservices-node-js/, 2022. [Accessed 12-Oct-2022].

[35] Benjamin Anderson and Brad Nicholson. Sql vs. nosql databases: What's the difference. *IBM [online]. Dostupno na: https://www. ibm. com/cloud/blog/sql-vsnosql [28. 7. 2021.]*, 2021.

[36] Dongming Guo and Erling Onstein. State-of-the-art geospatial information processing in nosql databases. *ISPRS International Journal of Geo-Information*, 9(5):331, 2020.

[37] Benymol Jose and Sajimon Abraham. Exploring the merits of nosql: A study based on mongodb. In *2017 International Conference on Networks Advances in Computational Technologies (NetACT)*, pages 266–271, 2017.

APPENDIX A
ARCHITECTURE COMPARISON

Feature	Desktop Apps	Cloud Apps
Architecture	Requires development for various platforms; environment-dependent	Runs in user's web browser; environment-independent
Scalability	Has extra development overhead based on what operating system it is being used with	Access to features are consistent and equal for all users resulting in faster scalability for future
Security	Less prone to cyber attacks	More prone to cyber attacks
Ease of Use	Due to inconsistencies, they may behave differently on different operating systems. Multi-tenancy is hard to achieve.	Since they are hosted in a remote server accessed via browsers, all users have the same experience. Multi-tenancy is supported.
Cost	Can be more expensive to develop and maintain in the long-run. Also requires on-site engineers to help install and maintain the software.	Can be cheaper to build and maintain in the long run. Since the software is maintained in remote servers, maintenance costs are reduced.

Fig. 12. Architecture Table