two gaps: timetabling application (on the cloud!), how can we best meet scalability goals?

Analysis of Microservices Architecture for Automated Timetabling Platform

Research on infrastructure cost in terms of performance and reliability for running microservice architectures across various cloud providers

Sommer Harris
harris.som@northeastern.edu

Jaelyn Ma
ma.du@northeastern.edu

Meet Patel
patel.meet2@northeastern.edu

Kshitij Nair
nair.ks@northeastern.edu

Niranjan Velraj
velraj.n@northeastern.edu

what is the best architecture pattern given scalability goals?

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[30]. 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[6]. Providing effective timetabling solutions could return to these administrators weeks or months of their valuable time. There are few educational timetabling software products [5]. Of those that exist, timetabling applications use algorithms that date back to 1980s [25]. Over decades, multiple applications with various algorithms have attempted accommodating the growing constraints, but do not always produce results that fit all constraints [9], [7].

why not? how are we fitting constraints?

Depending on these constraints, the timetabling problem can be classified as an NP-Hard or NP-complete optimization problem [26]. Some applications, like aSc Timetables [2] and Lantiv [17], use backtracking with certain heuristic constraints to make an optimal schedule. Our application proposes to use an arguably better approach: Constraint Satisfaction Problem [8]. In Padilla et al., 2014, it was also found that there exists a gap with this software not being accessible or usable by non-experts[27] .

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 will also address in this paper various scalability strategies provided by different Cloud Service Providers and use metrics (discussed in Section X) to determine a cost-effective solution.

In this paper, we propose a cloud based solution that approaches timetable scheduling as a Constraint-Satisfaction Problem that is scalable, accessible and cost-effective adhering to the unique requirements in this field, built on the microservices architecture. We will be comparing different microservices architecture pattern across cloud providers AWS and Azure by monitoring and measuring specific metrics pertaining to cloud software. Our evaluation formula to compare these services will mathematically incorporate costs for number of users with the reliability of service.

2. Related Works

feature table: CSP, cloud-based, what constraints not met that we would meet?

This section reviews research on advantages of the cloud based solution and microservices, how microservices are currently hosted, and how we can use this information to select appropriate metrics when we test which provider would be best for this case.

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 Figure 5[1] [12][1][15][20][3].

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 [29].

One of the most important factors in cloud service performance is Architecture. It is crucial to availability, consistency, and scalability of cloud applications [35]. A monolithic architecture has vertical scale benefits, but a single change in the system will impact all users [4]. In a large

[1] Appendix displays architecture feature analysis in a table.

what have people done in the past for modeling balance between desired features and cost?

on-demand application, the system must be able to handle the load of multiple users without downtime [4]. With Microservices, the services each fulfill a single function in the application [28]. Microservices scale well because they can horizontally scale and are loosely coupled so are fault-tolerant and isolated [4]. Although microservices are often recommended for large-scale applications, incorporating microservices to smaller applications will bring ease to future scaling [4].

There are several patterns to deploy a microservice where each pattern comes with its own tradeoffs and cost structure.[31] 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.

As two of the biggest cloud service providers, both Azure and AWS provide multiple solutions for microservices deployment. Azure has three core services that are often discussed for the purpose of being used to deploy microservices applications, each comes with its own advantages and disadvantages.[21] Services Fabric allows service life cycle management; Azure Kubernetes Services (AKS) deploys and manages containerized microservices within fully managed Kubernetes clusters; and Azure Functions allows serverless approach.[21] AWS offers various solutions to deploy microservices, such as EC2, Amazon Elastic Container Service (Amazon ECS), Amazon Elastic Kubernetes Services (Amazon EKS) and AWS Lambda.[32] Amazon EC2 as a Infrastructure-as-a-Service(IaaS) provider deploys microservices on virtual machines [16]; Amazon ECS provides Docker container management services [19]; EKS deploys containerized application while provisioning Kubernete management services [13]; AWS Lambda deploys microservices in serverless computing technology [14].

To identify the cost-effective solution for hosting, we can compare the offerings from major cloud service providers. Existing comparisons of the cloud providers generally tend to favor AWS for its broader offerings and faster response times [33] [24]. But when we consider deploying microservice applications, the results are mostly inconclusive. With lots of factors to take into consideration, there are no standard metrics that can be used to compare all applications [18] [11]. The parameters to compare should 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.

In this paper, we will focus on exploring the best architecture pattern for hosting microservices across multiple Cloud Service Providers for our timetable scheduling application based on proper metrics to performance. 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 hosting it in different cloud providers will be necessary to identify the best hosting solution.

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 [34]. 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 [34].

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 Figure 1. Sometimes this architecture is preferred due to ease of installations, more straightforward configuration, and less cross-service debugging [23].

3.2. Pattern 2 : One Host, Multiple Services

This is the most standard way to deploy a microservice application. 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. This approach has certain benefits and drawbacks. Scaling up would just require us to copy the service to another host and start it 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. One drawback of this approach

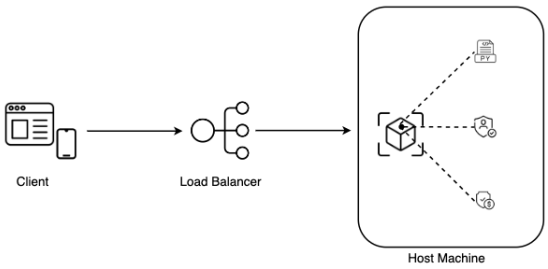

Figure 1. Pattern 1: Monolithic Architecture

is that there is no isolation of the service instances as we cannot limit the resources each instance uses.

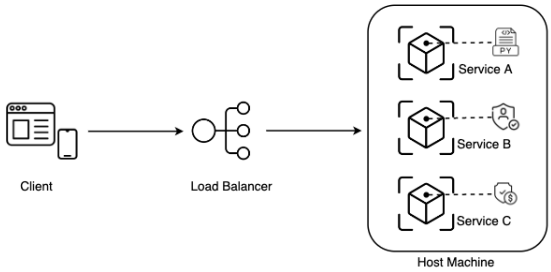

Figure 2. Architecture Pattern 2: One Host, Multiple Services

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

In this pattern, instead of having all the services in a single host, we package each service in its own host. 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.

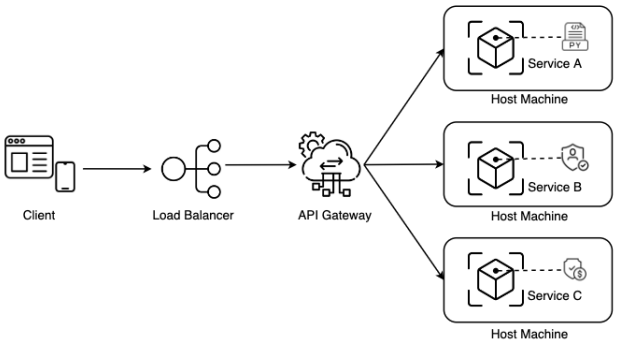

Figure 3. Architecture Pattern 3: One Host, One Service

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

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. While running services packaged as virtual images works, they are heavy to deploy as they contain operating system along with the code. 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.

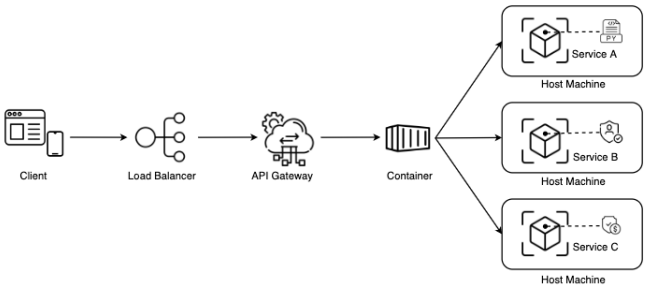

Figure 4. Architecture Pattern 4: One Host, One Service, with Container

3.5. Pattern 5 : Serverless Deployment

In this pattern, each service is packaged and deployed on Serverless platform without worrying about managing host machines. The cloud provider automatically assigns and manages the required number of machines to handle the demand. The developers need not worry about scaling up/down as the cloud providers have provisions to allocate resources as required. Unlike in previous patterns, we can invoke the service in multiple ways like using api request, event, cron jobs or api gateway redirects. One drawback of this approach is limited runtime of each service and cold starts.

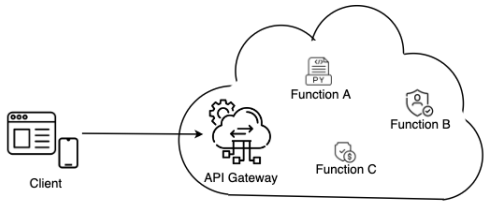

Figure 5. Architecture Pattern 5: Serverless Deployment

3.6. 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

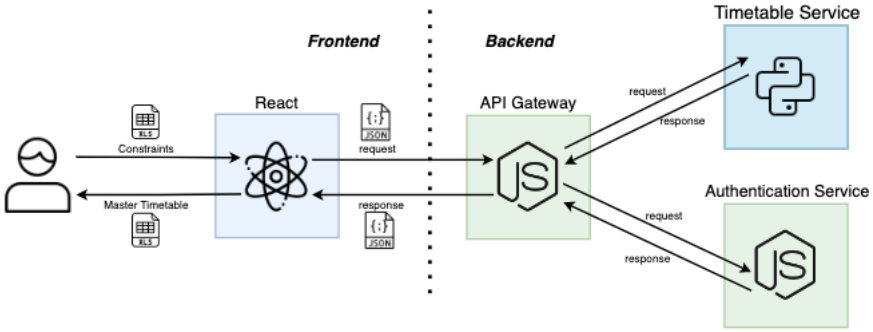

Figure 6. Basic System Design

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:

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

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

4. System Design

Automated timetabling platform is an end to end cloud application that accepts excel files as input which provides the constraints and generates a master timetable. The front end of the application is built with React, a popular JavaScript library that emphasizes on component-based approach to building user interfaces. 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 [10]. 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 desing pattern to have a single server communicating with client, hiding the internal architectural details of the server. Node js is an open-source runtime environment that uses non-blocking and event-driven architecture which makes it efficient and suitable as a server for microservice architectures [22]. This Node js server interacts with the Python server that hosts the algorithm to generate the timetable. Additional microservices for Authentication and data storage can be added in parallel to the Python server and Node js can redirect the requests to the corresponding microservices.

The input to the Frontend will be an excel file that has the information pertaining to timetable structure and the set of constraints that has to be satisfied. This is converted to a more standard JSON object and sent to the backend node server. This JSON file is then forwarded to the python server that generates the timetable and relays back the generated timetable as a response JSON object. The Node js server sends this response to the frontend where it is displayed to the user.

We implemented MongoDB as our database because potentially we want to allow users to be able to retrieve or change a timetable multiple times [30]. Compared to SQL databases, NoSQL databases appear to be better for this use case. NoSQL databases are horizontally scalable due to built-in sharding [**?**]. Queries respond faster in NoSQL databases with large data sets, because they do not have SQL normalization requirements[**?**]. Dynamic schema also makes NoSQL databases flexible in making application changes [**?**].

4.1. Pattern 1 Deployment

In our monolithic patters, 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 (run on node) and the flask server to our main node server. We ran our python timetabling code (previously on the flask server) by spawning a child process from the main node 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.

4.2. Pattern 2 Deployment

In a one-host-multiple-services pattern, system design of this deployment pattern is very similar to our basic system design. Requests from the front end will be routed to the backend via an Application Load Balancer (ALB). The ALB can route the network packet to different backend services based on the content of the network packet. In the backend, we keep the Flask server and Node js server separate in the backend to generate timetable service and perform authentication service respectively. Both servers will be deployed and run on a single host machine, an EC2 instance. Timetable service and authentication services listen to a different port and communicate over an API proxy server, the Node js server.

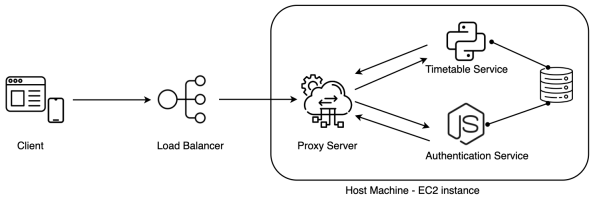

Figure 7. Deployment Pattern 2: One Host, Multiple Services

4.3. Pattern 3 Deployment

In this pattern, each microservice will be deployed in the cloud 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. 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. In this way, each microservice is independently deployed and remains scalable and responsive. The host machines are part of the AWS EC2 systems. Both types of servers in our application- Node.js and Flask will be deployed using Terraform on AWS.

4.4. 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. 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.

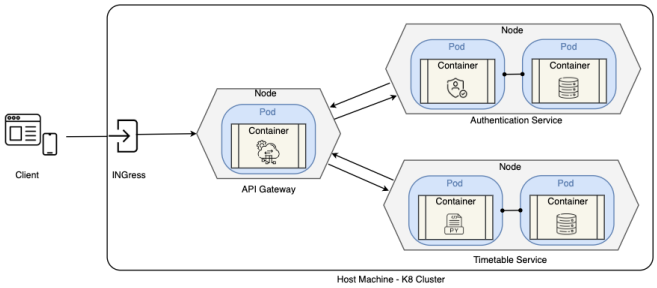

Figure 8. Deployment Pattern 4: One Host, One Services (Containers)

We will 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.

4.5. Pattern 5 Deployment

All of the patterns discussed above require manual cloud infrastructure management after deployment. For example, if there are more users for the authentication microservice, then we need to manually scale up the number of servers it is deployed in to accommodate the increased demand. To manage the cloud infrastructure, most companies have a team of developers who need to take time off from developing new features to ensure that the application is al-

ways accessible in the cloud. The Serverless architecture pattern provides a self-maintaining, scalable, and reliable approach to deployment that does not require any manual involvement in infrastructure management. Once the application is deployed in serverless architecture, the responsibility of managing the cloud infrastructure falls on the cloud provider.

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 major approaches to deploying serverless functions in AWS. The first approach is to deploy the serverless functions directly where the cloud provider selects the underlying infrastructure where it is deployed. This approach might not provide the best results when performing comparisons with other architectures as the infrastructure where it is deployed may vary from the other architectural patterns. The other approach to creating serverless functions involves using container images where we have more control over the infrastructure where this will be hosted. Container images are also reusable as we can easily deploy them across multiple servers if needed. To ensure that we are performing valid comparisons, we can reuse the same container images created for Pattern 4 and host these container images as serverless functions as this will provide more reliable results during evaluation.

References

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

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

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

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

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

[6] C. Constantia, P. Christos, R. Glykeria, A.-R. Anastasia, and V. Aikaterini. The impact of covid-19 on the educational process: The role of the school principal. *Journal of Education*, page 00220574211032588, 2021.

[7] C. Cooper. The challenges of timetabling part-time staff, Jun 2018.

[8] V. Ďuriš. Algorithmic verification of constraint satisfaction method on timetable problem. 2020.

[9] H. Eledum. The university timetabling problem: Modeling and solution using binary integer programming with penalty functions. *International Journal of Applied Mathematics and Statistics*, 56:164–178, 01 2017.

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

[11] M. Fazio, A. Celesti, R. Ranjan, C. Liu, L. Chen, and M. Villari. Open issues in scheduling microservices in the cloud. *IEEE Cloud Computing*, 3(5):81–88, 2016.

[12] L. Golightly, V. Chang, Q. A. Xu, X. Gao, and B. S. Liu. Adoption of cloud computing as innovation in the organization. *International Journal of Engineering Business Management*, 14:184797902210939, 2022.

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

[14] P. Jamshidi, C. Pahl, N. C. Mendonça, J. Lewis, and S. Tilkov. Microservices: The journey so far and challenges ahead. *IEEE Software*, 35(3):24–35, 2018.

[15] L. 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.

[16] H. Khazaei, C. Barna, N. Beigi-Mohammadi, and M. Litoiu. Efficiency analysis of provisioning microservices. In *2016 IEEE International conference on cloud computing technology and science (CloudCom)*, pages 261–268. IEEE, 2016.

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

[18] P. Leitner, J. Cito, and E. 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.

[19] D. Liu, H. Zhu, C. Xu, I. Bayley, D. Lightfoot, M. Green, and P. Marshall. Cide: An integrated development environment for microservices. In *2016 IEEE International Conference on Services Computing (SCC)*, pages 808–812, 2016.

[20] A. L. loves to learn, share what she has learned. She gathers all her learnings through reading books, and X. Labis. Cloud based software vs desktop software. *Syntactics Inc.*, Jun 2021.

[21] G. D. Mayaan. Four Steps to Running Microservices in Azure — containerjournal.com. `https://containerjournal.com/features/four-steps-to-running-microservices-in-azure/`, 2021. [Accessed 29-Sep-2022].

[22] G. D. Mayaan. Reasons to build microservices with node.js. `https://bambooagile.eu/insights/microservices-node-js/`, 2022. [Accessed 12-Oct-2022].

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

[24] E. F. Noviani, B. Kembara, B. A. Y. Pratama, D. A. P. Sari, A. M. Shiddiqi, and B. J. 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.

[25] R. Oude Vrielink, E. Jansen, E. W. Hans, and J. van Hillegersberg. Practices in timetabling in higher education institutions: a systematic review. *Annals of operations research*, 275(1):145–160, 2019.

[26] R. A. Oude Vrielink, E. A. Jansen, E. W. Hans, and J. van Hillegersberg. Practices in timetabling in higher education institutions: A systematic review. *Annals of Operations Research*, 275(1):145–160, 2017.

[27] J. J. Padilla, S. Y. Diallo, A. Barraco, C. J. Lynch, and H. 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.

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

[29] V. V. 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.

[30] J. A. Richard Hoshino. Automating school timetabling with constraint programming. 2022.

[31] C. Richardson. Choosing a microservices deployment strategy, 2016.

[32] F. Rouxel. A new way of deploying a microservice in AWS — linkedin.com. `https://www.linkedin.com/pulse/new-way-deploying-microservice-aws-fran%C3%A7ois-rouxel/`, 2021. [Accessed 29-Sep-2022].

[33] M. Saraswat and R. 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.

[34] M. Villamizar, O. Garcés, H. Castro, M. Verano, L. Salamanca, R. Casallas, and S. 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.

[35] H. Vural, M. Koyuncu, and S. Guney. A systematic literature review on microservices. In O. Gervasi, B. Murgante, S. Misra, G. Borruso, C. M. Torre, A. M. A. Rocha, D. Taniar, B. O. Apduhan, E. Stankova, and A. Cuzzocrea, editors, *Computational Science and Its Applications – ICCSA 2017*, pages 203–217, Cham, 2017. Springer International Publishing.

5. Appendix

5.1. Architecture Table

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.

Figure 9. Architecture Table