Database Scaling on Kubernetes

H.C.S. Perera
Department of Computer Science and Software Engineering, Faculty of Computing, Sri Lanka Institute of Information Technology
B263, Malabe 10115, Sri Lanka
hcsperera@gmail.com

T.S.D. De Silva
Department of Computer Science and Software Engineering, Faculty of Computing, Sri Lanka Institute of Information Technology
B263, Malabe 10115, Sri Lanka
samanthi20258@gmail.com

W.M.D.C. Wasala
Department of Computer Science and Software Engineering, Faculty of Computing, Sri Lanka Institute of Information Technology
B263, Malabe 10115, Sri Lanka
durekshawasala@gmail.com

R.M.P.R.L. Rajapakshe
Department of Computer Science and Software Engineering, Faculty of Computing, Sri Lanka Institute of Information Technology
B263, Malabe 10115, Sri Lanka
randu612@gmail.com

N. Kodagoda
Department of Computer Science and Software Engineering, Faculty of Computing, Sri Lanka Institute of Information Technology
B263, Malabe 10115, Sri Lanka
nuwan.k@sliit.lk

Udara Srimath S. Samaratunge Arachchilage
Department of Computer Science and Software Engineering, Faculty of Computing, Sri Lanka Institute of Information Technology
B263, Malabe 10115, Sri Lanka
udara.s@sliit.lk

H.H.N.C. Jayanandana
Department of Computer Science and Software Engineering, Faculty of Computing, Sri Lanka Institute of Information Technology
B263, Malabe 10115, Sri Lanka
nilesh@wso2.com

2021 3rd International Conference on Advancements in Computing (ICAC) | 978-1-6654-0862-2/21/$31.00 ©2021 IEEE | DOI: 10.1109/ICAC54203.2021.9671185

***Abstract*— Kubernetes is a hot topic in the field of Software Engineering and Distributed Computing. When compared to previous methods, the principle underlying Kubernetes, which is containerization, has altered how applications are created and delivered. However, when considering the state, particularly the databases, with Kubernetes, there is a scalability and data synchronization barrier. The most frequently used approach is to host the database outside of Kubernetes and maintain connectivity with the cluster. Kubernetes inherent capabilities are sufficient for hosting databases. But that requires high domain knowledge to do the configurations and maintain the databases on Kubernetes. The purpose of this research is to fulfil that gap by introducing a solution for managing highly available databases on Kubernetes. The solution is limited to managing PostgreSQL databases on Kubernetes using auto-scaling. A novel algorithm is proposed for auto-scaling, as previous algorithms do not take database requests into account when determining the scaling need. The drawbacks of data synchronization and auto-scaling will be solved in this research, and the end user will be able to access the service without interruption for the majority of the time, as the final solution makes the database cluster highly available for the service layer.**

Keywords—Kubernetes, PostgreSQL, Autoscaling, High Availability, Data Synchronization, Time Series

I. INTRODUCTION

Containerization is a technology that enables operating system-level virtualization and enables the development of lightweight portable applications. Docker is a well-known platform for package source code and dependencies together as a container image. Kubernetes is an open-source container orchestration platform that was introduced in 2014 to manage containerized software deployments [1]. Kubernetes can be configured to run on any cloud, whether private, public, or hybrid. Initially, container orchestration was used by IT companies such as Google to deploy and expand highland services. Then, with the hype around Microservice architecture, this trend of containerized software gained widespread adoption. Cloud computing's meteoric rise over the last few years has elevated Microservices to the status of a major topic of discussion in the area of Software Engineering. Prior to the advent of microservices, monolithic programs were the preferred architectural type.

However, the community has made the transition to Microservice architecture in order to take use of the growing benefits of cloud computing resources [2]. In addition to contributing significantly to the expansion of streaming services such as Netflix, Kubernetes has made substantial contributions to the expansion of financial services in the banking sector as a fast-evolving platform. However, it has not been attempted with database technologies, even in test environments, because database systems involve complicated operations and data persistency and consistency play a critical part in a database system, which was initially lacking in the Kubernetes environment [3].

A. Objective of The Project

The primary goal of this solution is to provide developers with a highly accessible database on Kubernetes that they can use with any application, regardless of their previous knowledge or experience with Kubernetes, distributed architecture, or deployment management. A web application on the operator's front-end server lets any user interested in integrating apps with a Kubernetes-based highly accessible database to set up the database cluster, which allows them to quickly get their applications up and running. Developers may utilize the administrative dashboards in the frontend web application to keep track of the application's health and performance after it has been configured to their specifications. As a result of this, developers will not be required to do any configuration or to be concerned with the time and resources required to construct a highly available PostgreSQL database on Kubernetes from the ground up.

When a cloud database is used, the service provider is responsible for the majority of the operating responsibilities. By contrast, users have no control over database versions, updates, or scalability. As illustrated in this study, running a database on Kubernetes, as advocated in this paper, enables better control over database maintenance while still using the benefits of cloud computing.The proposed approach automates the majority of operations related to database management.

978-1-6654-0862-2/21/$31.00 ©2021 IEEE

II. Background of The Project

A. Database Evolution

Having accurate data is crucial to the operation of any company. The SQL language and relational databases have historically been the most widely used database technology [4]. Later, as the need for unstructured data became more pressing, NoSQL emerged as the next big thing. Database technologies have evolved into big data platforms as a result of the exponential growth in data volume and velocity over the previous decade. Cloud databases, regardless of the database type (SQL, NoSQL, or big data), are becoming increasingly popular for application development in recent years. For some industries, such as the banking industry, cloud databases are more concerned with security and data protection than managed databases, which is a distinction from managed on-premises databases. In such industries, it was typical to maintain a hybrid cloud while also retaining data on-premises that was subject to tight privacy restrictions. In order to reclaim control over data, which is currently lost while employing cloud databases, this was accomplished. When working in a cloud-native environment, users have the ability to design the server to their specifications and manage many layers of security independently.

B. PostgreSQL

Open-source object relational database management system PostgreSQL was first introduced in 1986 as a project named Postgres95 and is currently the fourth most popular database in the world. The project was renamed PostgreSQL in 1996, and it has since garnered widespread support from a large number of people. Postquel is the query language used with PostgreSQL to interact with data that supports inheritance, time travel, transitive closure, nested queries, and path expressions [5]. Postgres is a database management system that is currently used in commercial systems in conjunction with a storage system and write-ahead logs (WAL). Postgres is gaining popularity as a result of the addition of key-value storage, which makes it a viable alternative to NoSQL. PostgreSQL is a suitable database for the majority of applications due to its speed and security.

C. Kubernetes and Databases

Kubernetes is a container orchestration system founded in 2014 with a focus on stateless applications. Despite the fact that it has since been upgraded with stateful sets, it is still incapable of supporting a high-availability service such as a database. There are certain drawbacks to utilizing Kubernetes to deploy databases, including worries about data synchronization and an inability to maximize availability [6]. One of the most common approaches is to use a separate persistent disk as the datastore for the application, as explained below. However, due to the database's lack of scalability, this creates another single point of failure. Maintaining a database on Kubernetes and providing features comparable to those available in fully managed and cloud databases is a time-consuming process [3]. Scalability is a vital factor to consider if you are concerned about high availability and avoiding single point failure. When it comes to database scalability, Kubernetes' native features come in handy, but the key concerns are data synchronization and identifying the right scaling requirements. As a result, it's challenging to maintain a scalable database solution in a cloud-native environment. Numerous cloud-native PostgreSQL operators are available to aid users in controlling the database's lifecycle, from deployment through operations. Manual Postgres deployment on Kubernetes is highly time-consuming throughout both the development and deployment phases. Throughout the study, the PostgreSQL operator is utilized to initialize the Postgres database.

D. Algorithm Evolution

For containerized apps, in previous approaches, researchers created a master-slave architecture. Incoming requests are routed to the running containers. Components are deployed on slaves. Master has a self-adapter that is in charge of scaling the number of running containers. A new container is created for each threshold. [1], [12] A new replica is built if the load is too high. While this design appears to work, the disadvantage is that the threshold is chosen manually and not optimally [10].Queuing theory is a recent trend in scaling containerized systems. They created request queues for containers, application tiers, and virtual machines. Scaling begins when the queues are full or empty. This is a large-scale cloud stream processing experiment that uses STORM and Heron to scale with demand [7]. Another approach to containerized application autoscaling is time series analysis [13]. This study used CPU utilization to forecast future values. They used fuzzy logic to auto-scale cloud or open-stack infrastructure based on HTTP traffic load. It is a model-driven solution that strives for optimal resource utilization. This strategy can be used to predict future requests and traffic.

III. Methodology

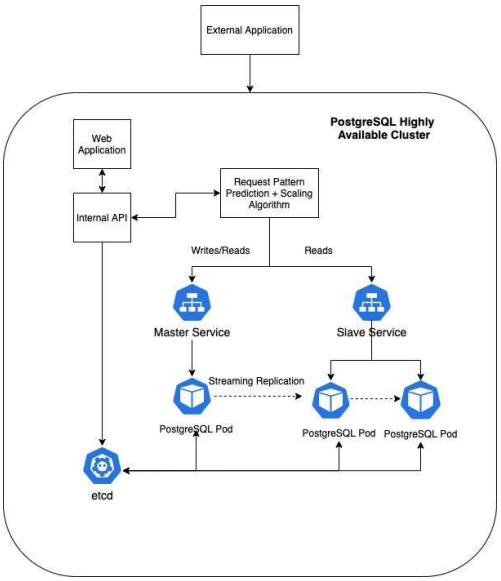

Fig. 1. High-level diagram

As illustrated in Figure 1 the module for future request prediction will forecast future request requirements separately for read and write requests and will save the forecast in the database. Once a new prediction is made, the mechanism for identifying scaling requirements is triggered, and the master and slave nodes are scaled in accordance with the prediction. After registering a new node, it will be

synchronized with the master node's write ahead log timeline. Even when a change occurs in any of the nodes, the synchronisation module will synchronize the changes using timeline comparison, and the validation module will validate all of the synchronizations.

A. Future Request Pattern Identification

For containers or microservices applications, Prometheus is a free and open-source event monitoring tool [7] and it's originally built at SoundCloud. Kubernetes and Prometheus were the first and second open-source projects to join the Cloud Native Computing Foundation, which had just been formed (CNCF) [1]. Both systems were built from the ground up to be cloud-native tools. While Kubernetes was created to manage large-scale microservices-based applications, Prometheus was created expressly to monitor and notify such systems. Long Short-Term Memory (LSTM) networks are well-suited to categorizing, processing, and making predictions based on time series data [8]. Because there might be lags of undetermined duration between critical occurrences in a time series. LSTMs were created to remove the problem of vanishing gradients that can occur when training traditional Recurrent Neural Network (RNN)s [8].

The proposed solution will make use of the Prometheus to gather incoming database requests. Incoming database requests are routed through the ingress controller and Prometheus gathers them. Based on those gathered metrics, a time series-based prediction on incoming database requests count, In which the expected number of inbound requests for a given period is anticipated. Moreover, the incoming database request prediction process is carried out using a Long short-term memory (LSTM) network, which uses a specific number of time steps of utilization measures to forecast future incoming database requests. The model predicts that in the next fifteen. In comparison to other LSTM models, the chosen LSTM model is capable of forecasting with greater accuracy. The algorithm will calculate the percentage of probable database requests after the predictions have been made and notify the scaling algorithm. Every minute, this process repeats again.

B. Scaling Requirement Identification

When considering about the horizontal Pod auto-scaling in Kubernetes, Kubernetes enables you to create and deploy Pods [9]. In Kubernetes, a pod is a collection of one or a small number of containers that are tightly coupled together through a common IP address and port space [10]. A Pod contains an application container (or, in some instances, several containers), storage resources, a unique network IP address, and parameters governing how the container(s) should operate. This process wraps an application container (or many containers), storage resources, a network IP, and parameters for how the container(s) should operate. A Pod is a deployment unit in Kubernetes, consisting of a single container or a small number of closely linked containers that share resources [11]. Here, we concentrate on single-container pods Duplicate and deploy the Pod that contains the application container to replicate an app's instance [9].

The "Escala" algorithm is based on several metrics which will be used to calculate the number of Postgres nodes which will be needed to deploy to cluster according to the future requests increment or decrement. Most of the metrics can be gathered by the Kubernetes itself with Prometheus or some other monitoring tools. The novelty in this algorithm is with the calculation made with results of the metrics gathered with the future request pattern prediction model.

Algorithm 1 Escala algorithm: Returns the number of pods to be deployed

---

Inputs:

CPU_{Target} – *Target CPU resource usage*
CPU_{Total} – *Total CPU usage of pods*
$Req_{Current}$ – *Current incoming request rate*
$Req_{Predicted}$ – *Predicted incoming request rate*
$N_{Current}$ – *Current Nodes*
N_1 – *Number of pods to keep the average CPU utilization a target value such as 80%.*
N_2 – *Number of pods need to be added or removed from the cluster*

Output: $N_{(pods)}$ – *Number of pods to be running*

do {

$$N_1 = N_{Current} \times \left\lceil \frac{CPU\ Total}{CPU\ Target} \right\rceil \quad (1)$$

$$Predicted_{NoReq} = Req_{Current} + (Req_{Current} \times Req_{Predicted})$$

$$N_2 = N_{Current} \times \left\lceil \frac{Predicted_{NoReq}}{Req_{Current}} \right\rceil \quad (2)$$

$$N_{(Pods)} = \lceil\ AVG\ (N_1, N_2)\ \rceil \quad (3)$$

If $N_{Current} < N_{(Pods)}$

Execute ($N_{(pods)}$ - $N_{Current}$);

} white(true);

Equation (1), the total CPU usage (CPU_{Total}) will be divided by the target CPU usage (CPU_{Target}) and the result will be multiplied by the current number of pods ($N_{Current}$). Target CPU utilization is a term that refers to the average CPU utilization that will be maintained throughout the cluster. The number of pods required to maintain a target CPU utilization (for example, 80 percent) will be determined as a result of this equation.

Equation (2), $Req_{Predicted}$ is the value of an increment or decrement which is received from the future request patterns prediction model and this value will be multiplied by the current request rate ($Req_{Current}$) and added the resulted value to $Req_{Current}$ to obtain $Predicted_{NoReq}$. This $Predicted_{NoReq}$ will be divided by ReqCurrent and the value obtained will be multiplied by the current number of pods, to arrive at a final figure. The total number of pods that must be added or removed from the cluster will be determined as a result of this entire equation.

Both (1) and (2) generate an output that specifies how many pods should be deployed in the cluster according to the future request increment or decrement of the cluster. Because precision is important, the maximum value of results (1) and (2) will be used, and this value will be the output of the algorithm (3).

Pods are periodically queried to get information about their metrics. That time has been specified as one minute, and

the forecast will occur 15 minutes ahead of time. Here, specified two values for the number of pods that should be deployed and the maximum number of pods that will be deployed to the cluster using the collected data.

CPU_{Total} refers to the total sum of the CPU usage throughout the cluster. This requires the gathering of CPU usage data from each pod. The calculation for (1) would be utilized to ensure that the mean CPU use of the cluster remained at the value specified in the CPU_{Target} parameter. In order to maintain the target CPU utilization close to, if not at, the target, pods will be added and removed as required.

$Req_{Predicted}$ metric is relying on the result of the future request pattern prediction model when evaluating (2). It is possible to collect both $N_{Current}$ and $Req_{Current}$ using monitoring tools. The final number of nodes to deploy would be calculated by comparing the current number of nodes.

C. Database Principals

The ACID principle (Atomicity, Consistency, Isolation, and Durability) [12] is a significant concept in the context of database management systems (DBMSs). The ACID is a collection of guiding principles that ensure the reliability and correctness of database transactions. A database transaction is any activity that occurs inside a database, such as the creation of a new record or the update of existing data. Changes to a database must be handled with care in order to avoid corrupting the data contained within it. The application of the ACID properties to each modification of a database is the most effective approach of maintaining the accuracy and reliability of the database [13]. Atomicity refers to the concept that either all of the transaction's tasks (or all of its tasks) are performed or none of them are finished at the end of the transaction. Essentially, this is the concept of "all or nothing." A failure in one of the transaction's components results in the failure of the entire transaction. Consistency is the second concept in the ACID principle; it ensures that the database is in a consistent state at the start and end of each transaction; no half-completed transactions are permitted.

Isolation refers to the fact that no transaction has access to an interim or partial state of another. As a result, each transaction is completely self-contained. This is essential for database transaction performance and consistency. Durability is the last ACID concept to be discussed. It implies that once a transaction is complete, it will remain as complete and will not be able to be undone [12]. Therefore, a more appropriate and effective method for ensuring consistency in the database on Kubernetes should be developed. To maintain the ACID principle across PostgreSQL database instances running on Kubernetes, this developed Kubernetes operator employs a variety of techniques, including data synchronization, data validation, rollback, crash recovery, and corruption handling.

D. PostgreSQL Replication

PostgreSQL replication is capable of supporting two different replication models: single master and multi-master replication [14]. These are referred to as unidirectional replication and bidirectional replication, respectively. Any changes that occur in a master database are synced with or replicated to replica databases when using single-master replication. Multi-master replication, on the other hand, ensures that changes are synced across many master databases, boosting both write availability and scalability. These synchronizations occur in two modes: asynchronous and synchronous. Synchronous mode marks a transaction complete in the master database after it has been replicated in all replication databases. However, in asynchronous mode, it is not necessary to wait for a transaction to complete in the master database until it has been replicated in all replication databases.

E. PostgreSQL Timelines

A write ahead log (WAL) is a commonly used technique for ensuring data integrity, and such write ahead logs are sometimes referred to as redo logs [15]. They include all database changes and may be used for replication, recovery, backup, and point-in-time recovery. PostgreSQL replication was introduced in 2006, however, it was limited to complete WAL file synchronization. Then, in 2009, streaming replication was developed, which is a record-based log shipping mechanism, rather to a file-based log shipping mechanism. When compared to a complete database backup, these WAL files include a chronology of a database's changes and may stream data from one database to another. These write ahead logs and record-based log shipping are utilized in this research project to create a highly available PostgreSQL cluster. This procedure consists of two components: a WAL sender and a WAL receiver [15]. In the suggested method, the settings of the WAL sender and WAL receiver are updated dynamically based on the cluster's current state. These inherent PostgreSQL functionalities, as well as native Kubernetes features like as fault recovery, are utilized during the data synchronization and validation processes as shown in Figure 2.

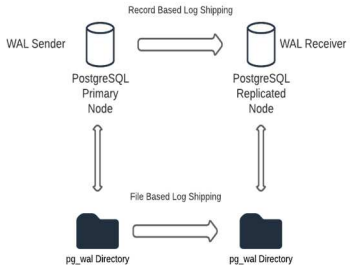

Fig. 2. PostgreSQL streaming methods

F. Security Aspects of The Solution

To prevent unauthorized access to the database, this operator consists of a feature called Role-based access control. Role-based access control (RBAC) is a technique for limiting network access to certain individuals within an organization based on their responsibilities. Role-based access control restricts workers' access to information to that is necessary to do their tasks and prevents them from obtaining information that is irrelevant to them. To operate databases in the Kubernetes environment, the operator needs wide permissions. Due to the critical importance of security, we factored in a separate operator-specific service account and used Kubernetes' RBAC capabilities to precisely specify the operator's necessary permissions while sticking to the concept of least privilege.

G. Alert and Monitoring

The suggested solution would utilize Grafana to do system monitoring. Grafana's open-source nature also enables us to create custom plugins for interaction with a variety of other data sources. The technology enables us to examine, analyze, and monitor data over time, a process is known as time series analytics [16]. The system retrieves data from the Prometheus endpoint and visualizes it using Grafana. Additionally, the system features a separate dashboard for visualizing database logs and nodes.

IV. System Overview

For developers, configuring a database in a Kubernetes environment is a time-consuming task. While it may look easy to maintain such a database environment as a highly accessible database, it is not. However, the newly developed operator is ideal for such situations. The solution as a whole is composed of four key components: forecasting future request patterns, determining scaling needs, database synchronization, and database validation as shown in Figure 3. For initial database cluster configurations and monitoring, the operator deploys a frontend application integrated with the same API which is connected to other components. This frontend application serves as a graphical user interface for the operator.

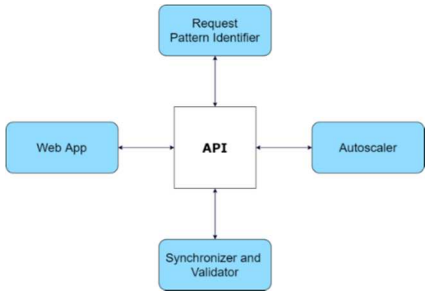

Fig. 3. System components

V. Results and Discussion

When considering previous approaches to implementing auto-scaling on clusters, several very successful methods were used. However, the majority of them are irrelevant to this operator. When it comes to scaling containerized applications, some examples include queueing theory [17], master-slave auto-scaling architecture [18], and time series analysis [19] can be considered. They analyzed a variety of factors, including CPU usage, conservative constant (α), adaptation interval or Control Loop Time Period (CLTP) [11], relative and absolute CPU intensive workload, correlation model linear fitting coefficients, and the number of active pods [9]. When these factors are considered, it is clear that not any approaches are applicable to this operator and that a new algorithm is required to achieve a better solution.

When configuring the database for testing, it is necessary to have data in the database. To accomplish this, we utilized the database seed script to populate the data tables with data. The database requests traffic for the load testing, which is produced using the JMeter load testing tool [20]. That is a scriptable performance testing tool capable of simulating high traffic to a particular server.

A. Evaluation of Escala Algorithm

In order to verify the Escala algorithm, several tests were conducted. To ensure the integrity, dependability, and efficiency of this algorithm in a time-consuming manner, it is needed to evaluate it. Actual data sets were used as input. Data were collected and analyzed findings using a scatter diagram to ensure that the algorithm was functioning properly and that the results were accurate. A sample data sets that we have used is to test the algorithm is given below in Table I.

TABLE I
Input Dataset For Testcases

Metric	CPU_{Target}	CPU_{Total}	$Req_{Current}$	$Req_{Predicted}$	$N_{Current}$
1	60 %	70 %	2500	0.07	6
2	60 %	55 %	4500	-0.25	10
3	60 %	90 %	2700	2.00	7
4	60 %	70 %	5300	-0.90	11
5	60 %	70 %	8150	0.15	20

TABLE II
Testcase Results Evaluation

	N_1	N_2	Actual $N_{(pods)}$	Expected $N_{(pods)}$
1	7	7	7	7
2	9	8	8	7
3	12	14	13	13
4	3	2	3	2
5	24	23	24	24

The expected value in Table II is obtained by calculating the number of pods that will be needed to be managed in a pure Postgres database instance by providing sample requests in each case.

B. Gathering metrics for Escala Scaling Algorithm

The primary goal of this section is to validate the algorithm as well as the metrics that we use to assess the efficiency of the proposed scaling technique. Multiple validation tests were carried out in order to ensure the integrity of the formula. In order to evaluate this algorithm, under the following environment, the tests were performed.

Database Volume: 10GB
Incoming Request Rate: 5000
Maximum number of pods: 10

This section is discussed under the above condition. All the values can be changed according to the change in the environment. In the process of gathering time data to be used in the algorithm, the research team only conducted a few tests that should have been done manually. Information was analyzed after a number of iterations were carried out. A sample calculation for T_{Total} is shown the Table III.

T_1 – Time to create a new Pod
T_2 – Time to complete the data synchronization
T_3 – Time to complete the data validation

TABLE III
Sample T_{TOTAL} Calculation

	T_1 Value	T_2 Value	T_3 Value	T_{Total} Value
1	4.53	26.57	1.01	32.11
2	5.21	23.17	0.47	28.85
3	5.12	25.48	1.02	31.62
4	5.26	24.59	0.5	30.35
5	4.59	23.48	0.58	28.65
Total Average				30.32

$T_{Total} = 30.32 \cong 30\ seconds$

Apart from the attributes we are considering, there are other attributes that contribute indirectly to this algorithm, such as server performance, cloud service provider service, and so on. With these attributes, some errors may occur. These steps have been taken to ensure that the errors can be ignored.

VI. Future Work

Currently, this solution focuses exclusively on maintaining and deploying highly accessible PostgreSQL databases on Kubernetes, as PostgreSQL is one of the most widely used databases in the industry. After conducting the further study, this approach may be expanded to accommodate additional databases. While this operator is now effectively maintaining the Postgres service on Kubernetes, there is a guaranteed possibilities to enhance performance, security, and reliability in the future with the addition of more features.

VII. Conclusion

This paper discusses a viable method for deploying databases on Kubernetes without the need of hazel by utilizing a built operator and frontend application. The presented approach automates the initialization and management of an autoscaling PostgreSQL cluster on Kubernetes. Additionally, the auto-scaling process, which comprises of a method for identifying future request patterns and a mechanism for identifying scaling requirements. These two components can be utilized in conjunction with other research on database scaling, as they were designed after researching database traffic and how databases react under conditions of fluctuating database request traffic. The data synchronization and validation techniques rely on native PostgreSQL functionality and Kubernetes capabilities. This article discusses the technologies that were utilized in the study and implementation process. Apart from that, this publication details the future work that may be done to enhance the solution.

Acknowledgment

This paper was written to satisfy the requirements for the Bachelor of Science Special (Honors) in Information Technology degree offered by the Sri Lanka Institute of Information Technology.

References

[1] A. Zhao, Q. Huang, Y. Huang, L. Zou, Z. Chen, and J. Song, "Research on Resource Prediction Model Based on Kubernetes Container Auto-scaling Technology," *IOP Conference Series: Materials Science and Engineering*, vol. 569, no. 5, 2019, doi: 10.1088/1757-899X/569/5/052092.

[2] L. A. Vayghan, M. A. Saied, M. Toeroe, and F. Khendek, "Kubernetes as an Availability Manager for Microservice Applications," 2019, [Online]. Available: http://arxiv.org/abs/1901.04946

[3] J. T. Olle Larsson, Johan Tordsson, "Running Databases in a Kubernetes Cluster," p. 43, 2019, [Online]. Available: https://www.diva-portal.org/smash/get/diva2:1369598/FULLTEXT01.pdf

[4] J. Hainaut, V. Englebert, and J. Henrard, "Database Evolution : the DB-MAIN Approach Facultes Universitaires Notre-Dame de la Paix Institut d ' Informatique Evolution of database Applications : the DB-MAIN Approach RP-94-016," no. June 2014, 1996.

[5] M. Stonebraker, L. a Rowe, and M. Hirohama, "the Implementation of Postgres 2 . the Postgres Data Model and Query Language," *IEEE transactions on knowledge and data engineering, v. 2, n. 1,* pp. 125–142, 1990.

[6] A. Tesliuk, S. Bobkov, V. Ilyin, A. Novikov, A. Poyda, and V. Velikhov, "Kubernetes container orchestration as a framework for flexible and effective scientific data analysis," *Proceedings - 2019 Ivannikov Ispras Open Conference, ISPRAS 2019*, pp. 67–71, 2019, doi: 10.1109/ISPRAS47671.2019.00016.

[7] N. Sukhija and E. Bautista, "Towards a framework for monitoring and analyzing high performance computing environments using kubernetes and prometheus," *Proceedings - 2019 IEEE SmartWorld, Ubiquitous Intelligence and Computing, Advanced and Trusted Computing, Scalable Computing and Communications, Internet of People and Smart City Innovation, SmartWorld/UIC/ATC/SCALCOM/IOP/SCI 2019*, pp. 257–262, 2019, doi: 10.1109/SmartWorld-UIC-ATC-SCALCOM-IOP-SCI.2019.00087.

[8] S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997, doi: 10.1162/neco.1997.9.8.1735.

[9] E. Casalicchio and V. Perciballi, "Auto-Scaling of Containers: The Impact of Relative and Absolute Metrics," *Proceedings - 2017 IEEE 2nd International Workshops on Foundations and Applications of Self* Systems, FAS*W 2017*, pp. 207–214, 2017, doi: 10.1109/FAS-W.2017.149.

[10] S. Kho Lin *et al.*, "Auto-Scaling a Defence Application across the Cloud Using Docker and Kubernetes," *Proceedings - 11th IEEE/ACM International Conference on Utility and Cloud Computing Companion, UCC Companion 2018*, pp. 327–334, 2019, doi: 10.1109/UCC-Companion.2018.00076.

[11] S. Taherizadeh and M. Grobelnik, "Key influencing factors of the Kubernetes auto-scaler for computing-intensive microservice-native cloud-based applications," *Advances in Engineering Software*, vol. 140, no. September 2019, p. 102734, 2020, doi: 10.1016/j.advengsoft.2019.102734.

[12] K. Machado, R. Kank, J. Sonawane, and S. Maitra, "A Comparative Study of ACID and BASE in Database Transaction Processing," *International Journal of Scientific & Engineering Research*, vol. 8, no. 5, pp. 116–119, 2017, [Online]. Available: http://www.ijser.org

[13] M. Vieira, A. C. Costa, and H. Madeira, "Towards timely ACID transactions in DBMS," *Proceedings - 12th Pacific Rim International Symposium on Dependable Computing, PRDC 2006*, no. January 2004, pp. 381–382, 2006, doi: 10.1109/PRDC.2006.63.

[14] T. Pohanka and V. Pechanec, "Evaluation of replication mechanisms on selected database systems," *ISPRS International Journal of Geo-Information*, vol. 9, no. 4, 2020, doi: 10.3390/ijgi9040249.

[15] B. Shaik and A. Vallarapu, *Beginning PostgreSQL on the Cloud*. 2018. doi: 10.1007/978-1-4842-3447-1.

[16] E. Karavakis and A. Aimar, "Kibana, Grafana and Zeppelin on Monitoring data," no. August, 2016, doi: 10.5281/zenodo.61079.

[17] T. M. Truong, A. Harwood, and R. O. Sinnott, "Predicting the stability of large-scale distributed stream processing systems on the cloud," *CLOSER 2017 - Proceedings of the 7th International Conference on Cloud Computing and Services Science*, no. Closer, pp. 575–582, 2017, doi: 10.5220/0006357606030610.

[18] P. P. Kukade and G. Kale, "Auto-Scaling of Micro-Services Using Containerization," *International Journal of Science and Research (IJSR)*, vol. 4, no. 9, pp. 1960–1964, 2013.

[19] H. Arabnejad, C. Pahl, P. Jamshidi, and G. Estrada, "A comparison of reinforcement learning techniques for fuzzy cloud auto-scaling," *Proceedings - 2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGRID 2017*, no. May, pp. 64–73, 2017, doi: 10.1109/CCGRID.2017.15.

[20] M. A. Putri, H. N. Hadi, and F. Ramdani, "Performance testing analysis on web application: Study case student admission web system," *Proceedings - 2017 International Conference on Sustainable Information Engineering and Technology, SIET 2017*, vol. 2018-Janua, no. November, pp. 1–5, 2018, doi: 10.1109/SIET.2017.8304099.