Analyzing Latency Performance of Different Cache Methods for Microservice Architecture

Nur Ayuni Nor Sobri[1], Mohamad Aqib Haqmi Abas[1], Ihsan Mohd Yassin[2*], Megat Syahirul Amin Megat Ali[2], Nooritawati Md Tahir[3], Azlee Zabidi[4], Zairi Ismael Rizman[5]

[1]School of Electrical Engineering, College of Engineering, Universiti Teknologi MARA, 40450 Shah Alam, Selangor, Malaysia

[2]Microwave Research Institute, Universiti Teknologi MARA, 40450 Shah Alam, Selangor, Malaysia

[3]Research Nexus UiTM, Universiti Teknologi MARA, 40450 Shah Alam, Selangor, Malaysia

[4]Faculty of Systems & Software Engineering, College of Computing & Applied Sciences, Universiti Malaysia Pahang, 26600 Pekan, Pahang, Malaysia

[5]School of Electrical Engineering, College of Engineering, Universiti Teknologi MARA, 23000 Dungun, Terengganu, Malaysia

Corresponding author: ihsan.yassin@gmail.com

Article Info
Page Number: 915 – 927
Publication Issue:
Vol. 71 No. 3s2 (2022)

Abstract

Nowadays, due to popularization of the internet, the demands for a robust and scalable application system are increasing rapidly. Underlying backend of the application system are required to be scalable to allow thousands of users to be served concurrently. With the rise of microservice architecture, load balancing strategy can be used efficiently to distribute load evenly between service instances for service that runs on high load. Moreover, cache technology is also used heavily to reduce the load impact on databases. This paper investigates the use and performance result of cache layer as opposed to database only for storing data in a microservice setting where we propose on running multiple instances of our services for load balancing purposes in production. The result we obtained for a normal architecture that runs a single service instance in-process memory cache would be the most beneficial, while for microservice architecture with service that runs on multiple instances, Redis would give the best performance and data integrity.

Article History
Article Received: 28 April 2022
Revised: 15 May 2022
Accepted: 20 June 2022
Publication: 21 July 2022

Keywords: Backend application system; microservice; load balance; cache.

Introduction:

Research background

In the past decade, the software architecture paradigms have started to shift from a monolithic architecture that contains tightly coupled logic with a more modular, loose-coupled microservice architecture. Microservice architecture has been found by many to have

tremendous positive impacts on software architecture development and deployment. In development, it gives the developers agility and independent development capability as each service only concerns a small logic of the whole system [1-3]. As for the system that is created with microservice architecture, it results in a more efficient, flexible, robust, modular and scalable system [4-11].

In microservice architecture, the services are small in size where they consist of small loosely-coupled business logic that are bounded by small and similar domains. This has a huge positive impact as it allows for independence between services. In the development and maintenance phase of the software system, it promotes agility and flexibility for developers as it would be easier to debug for issues and also to add newer features to the service as the domain logic is small and easier to reason about [5]. When deploying the service, since it is independent from other services, unlike in monolith architecture, the services can be deployed independently and the configuration can be very flexible, where services that are most frequently used in the system can be tweak to increase the number of instance for load balance purpose or we can increase its resource of cpu cycle and memory. This promotes a highly scalable system and increases high availability and fault tolerance of the system [5, 12-15].

Since the services have their own bounded domain logic, each service would have their own state [16]. In other words, they would have their own database and cache store separate from other systems. Microservice architecture supports heterogeneity of language and technology to be used when developing for each service, this includes the programming language used to develop the system and the data store technology [7, 16, 17].

Cache allows for storing and retrieval of data at a faster speed compared to normal database store technology. Many applications nowadays require the system to serve requests with low latency. Having cache to store and serve the data plays an important role in this situation as end users typically consume more data that they generate [18, 19]. Redis (Remote Dictionary Server) is an in-memory key-value NoSQL datastore and one of the most popular technologies that is used as cache for applications nowadays [18, 20-24]. Redis can also be used as database, queue and message broker apart from the standard use as cache [25-27].

The general flow for a web server that has both cache and disk-system data stores to retrieve data is by first looking in the cache, if the data is not present, then it will initiate a second step, to look in the main data store. The reason being, normally cache system gives a faster retrieval and in most cases lives in-memory instead of in a disk-based database. Hence, this gives a significant latency boost and would easily allow the web server to serve at scale [18]. Even though cache is good, in some cases, bad implementation of cache could cause problems such as cache stampede that could severely limits the performance of the web server and database [28]; and incorrect data output which is usually caused by miss update when any mutation (update or delete) happens to the data in the system or failure on handling concurrent requests correctly.

This paper investigates the use and performance result of cache layer as opposed to database only for storing data in a microservice setting where we propose on running multiple instances of our services for load balancing purposes in production.

Related works

We have found some relevant studies that are related to our work in using cache. In [29], the author proposed a novel Redis-based Web server cluster session maintaining technology. It is to ensure the web cluster can scale more easily and provide a highly available service to the end user. The author compares between multiple session maintaining methods such as cookie-based, session replication, session sticky and cache server. The experiment done by the author shows the cache server using Redis strikes the best balance of performance and stability for the system compared to other session maintaining methods.

The study in [20] compares Redis as a database store with MariaDB relational database. The author designed a few sets of experiments with large amounts of data and compared the efficiency of operations like insert, update, delete and select from various aspects on the same dataset. The study shows that Redis gives a better run time performance compared to MariaDB. However, Redis has disadvantages with regular querying and updating that apply to an extensive dataset. MariaDB is also good when involving a smaller amount of data. The author suggested that Redis is to be used in the form of a temporary database to enhance performance of the database system.

In [25], the author reviewed Redis and its various module extension such as RedisSearch, RedisGear, RedisJSON, RedisAI, RedisGraph, RedisTimeSeries and few more which makes Redis to be capable not only as a cache and database but also supporting to build power applications with functionalities like search, real-time monitoring, analytics, gaming and many more.

Methodology:

For our application system, we want to ensure the system would be reliable, scalable and highly available. From Section I-B, based on our study from the literature on microservices architecture, we propose migrating the monolithic application system of our application to microservice architecture as it provides the benefits that align with our goal for the application.

Furthermore, to ensure a better scalability and high availability of the system, we propose running multiple instances of each service in our production environment. This is to ensure we can load balance the HTTP request that each of our services will be receiving from the API gateway. Fig. 1 shows the example of our proposed architecture between API gateway service and the role and permission service. The API gateway service acts as an intermediary service that routes the HTTP request from external to specific service that is being requested. In Fig. 1, we only use one service to display as example, which is the role and permission service, but in the whole system context, there could be hundreds or thousands of services that serves logic based on their specific domain and API gateway is going to be the main interface with the all

ISSN: 2094-0343
2326-9865

requests and redirect them to the domain specific service.

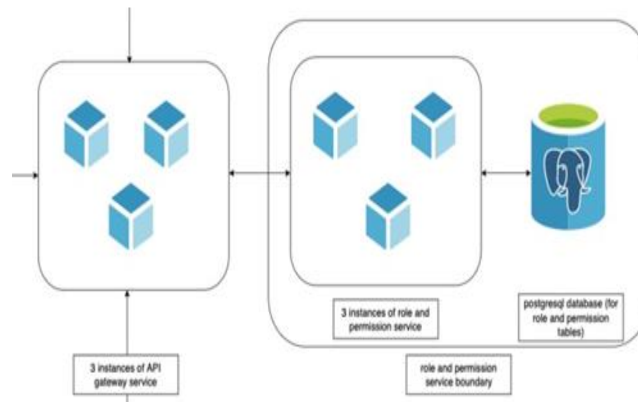

Fig. 1 Requests flow to and from API gateway service to role and permission service

Each of the services that we run on the system can be independently deployed and configured. We can tweak the number of resources such as the CPU cycle, memory allocation, storage space and number of instances to run based on how often the service is being hit by the end user. Microservice architecture is flexible and does not impose any hard requirements on the database as we can also replicate the database to provide load balance and backup capabilities between the master and its replica. It would help to reduce the load of only a single database being hit. However, it does not affect the latency when the database executes the query as it would have to retrieve the data from its disk store.

We propose on using cache data store as it would help tremendously in reducing the load of the Postgres database being hit from the application, improve application performance by having significantly better latency and increase the read throughput by having to access the data only in-memory instead of in the disk.

In this experiment we will be experimenting on the performance difference in latency when we retrieve the data from Postgres database, Redis cache and in-process cache when having a single instance service and three instances service. We perform the experiment using two different scenarios (single and three instances service) to check the difference of performance between normal method and load balanced method and the effect of using different cache mechanisms.

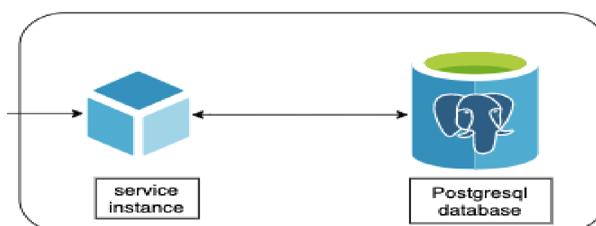

Fig. 2 HTTP request flow for single service instance calls to Postgres database directly

Fig. 2 shows the first situation where the service instance makes a direct query to the Postgres database.

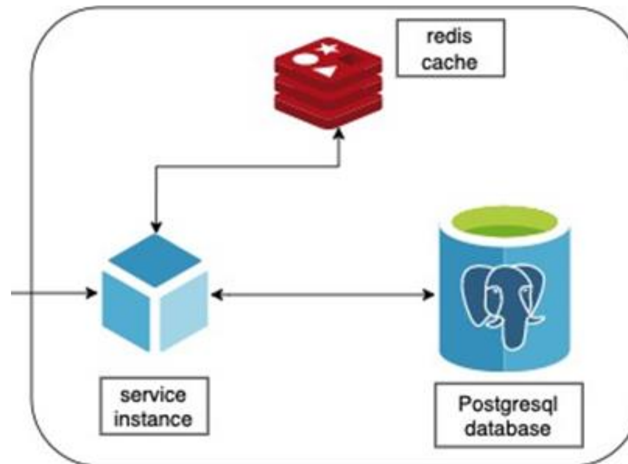

Fig. 3 HTTP request flow for single service instance calls to Redis cache, then Postgres to retrieve data

Fig. 3 shows the second situation where the service instance makes a call to Redis cache to retrieve the data, if the data is absent in Redis, then it will make the second call to Postgres database.

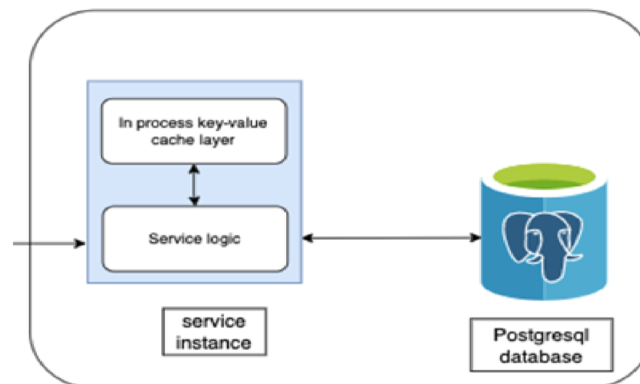

Fig. 4 HTTP request flow for single service instances call to its in- process cache, then Postgres to retrieve data

Fig. 4 shows the third situation where the service instance will first try to find the data from its own in-process memory, and if it's absent, it will then make a query to Postgres database to retrieve the data. Fig. 2 to 4 shows the experiment for the first scenario where we handle the requests with only a single instance of the service.

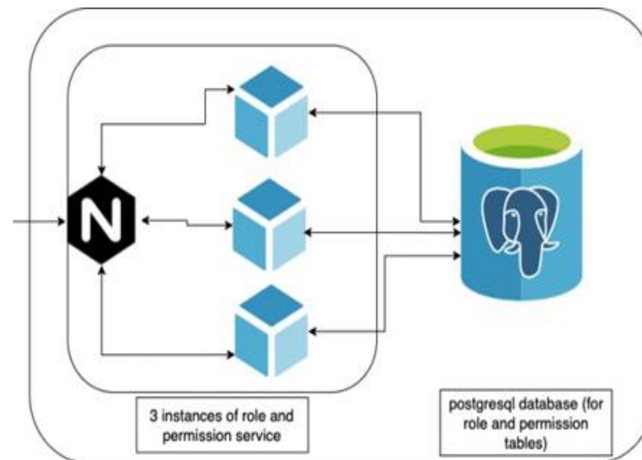

Fig. 5 HTTP request flow for three service instance directly call to Postgres database

Fig. 5 shows the fourth situation where there will be 3 service instances to handle the HTTP requests and all three of them directly calls to Postgres database without having any cache in between.

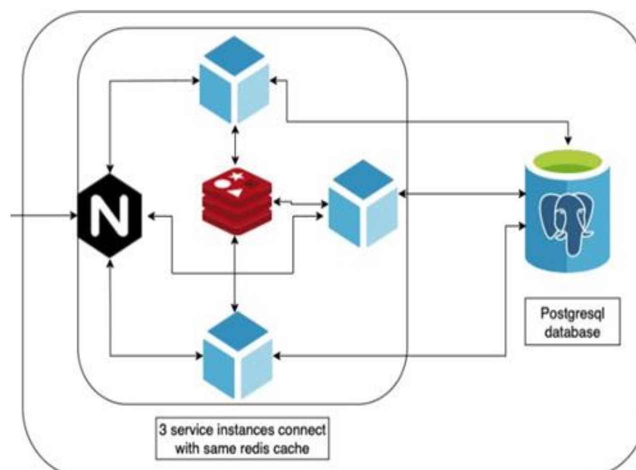

Fig. 6 HTTP request flow for three service instance calls to Redis cache, then Postgres to retrieve data

Fig. 6 shows the fifth situation where there will be 3 service instances to handle the HTTP requests and all three of them will first call the Redis cache to retrieve the data, if the data is absent, they will then call to Postgres database.

The sixth situation will be similar as shown in Fig. 5, but instead of calling directly to Postgres, the service instances will first find the data in their own in- process memory, if the data is absent, it will call the Postgres database.

From Fig. 2 to Fig. 6, the call made to Postgres and Redis requires the application service instance to call to different port on the local machine (default to port 5432 and 6379 respectively), while the in-process key-value cache would be from the same application and same process memory itself.

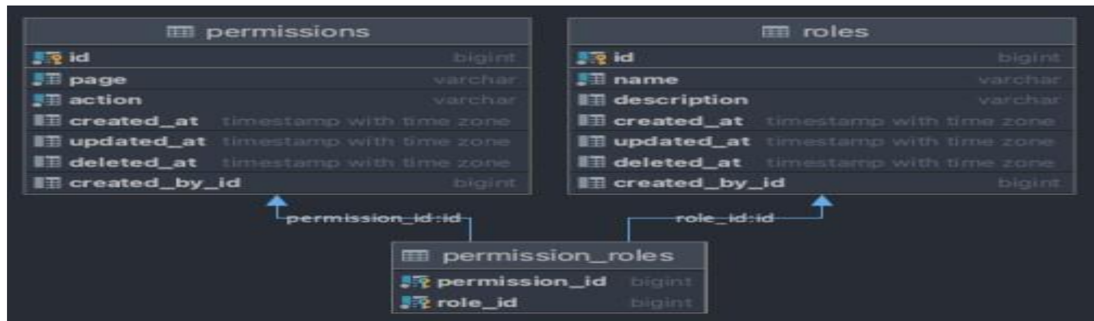

Fig. 7 Roles and Permission table

Fig. 7 shows the roles and permissions table with its intermediary many-to-many table. We are using a role based access control (RBAC) authorization model for our system. In RBAC, users have access to an object/page/module based on their respective assigned role in the system [30]. Roles are commonly assigned based on job function and permissions are defined based on job authority and responsibility of the job. The database tables from Fig. 7 is from the example we previously used in Fig. 1, where the service instance state (data) is being stored on. The reason that role/permission service are chosen for this paper is due to this service being one of the most used in the system. Multiple endpoints in the system require authorization checks on whether a specific user has the necessary role and permission needed to access the endpoint.

The load testing is going to be performed with Arm64v8 CPU architecture. The limitation of the platform applies to this project. We also limit the go runtime to use a single CPU (with GOMAXPROCS = 1) and 256MB of memory (ulimit), however neither limiting the CPU and RAM gives any effect in our experiment as none of the tests would even hit the limit of bottleneck. However, the situation will be different in production servers when we deploy the services where we have a more limited amount of CPU cycles and RAM configuration for our machine. In this test, we are using the default setting of Postgres database and Redis cache as will be in production without tuning any configuration, if needed Postgres and Redis are very flexible and can be tweaked easily in postgresql.conf file and command-line tools respectively.

Only one API endpoint will be tested for this experiment, which is the "/roles" endpoint that will give all the roles in the database table, including its permissions relation. We will test the endpoint using a HTTP load tester, vegeta, which is written in Go for 10 requests per second and get the latency reading for each experiment step shown in Fig. 2, 3 and 4. The reason we make 10 requests per second was to inspect the latency of calling the database and subsequent calls to cache (where applicable).

Results and Discussion:

The result of the experiment are as follows:

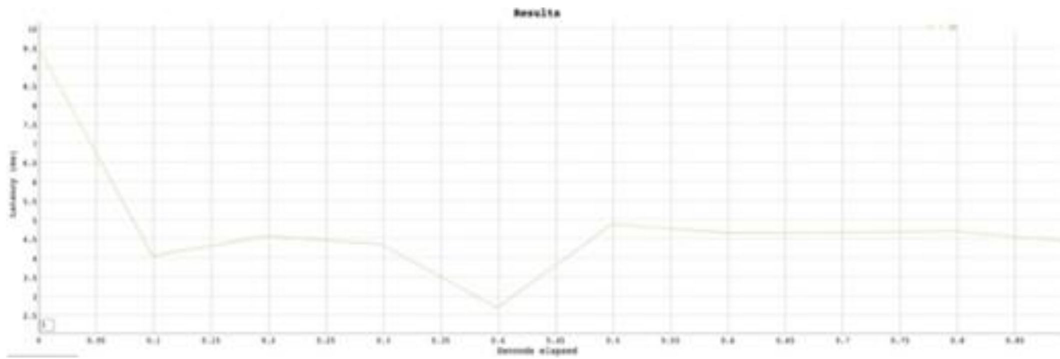

Fig. 8a Latency when a single instance directly calls Postgres database

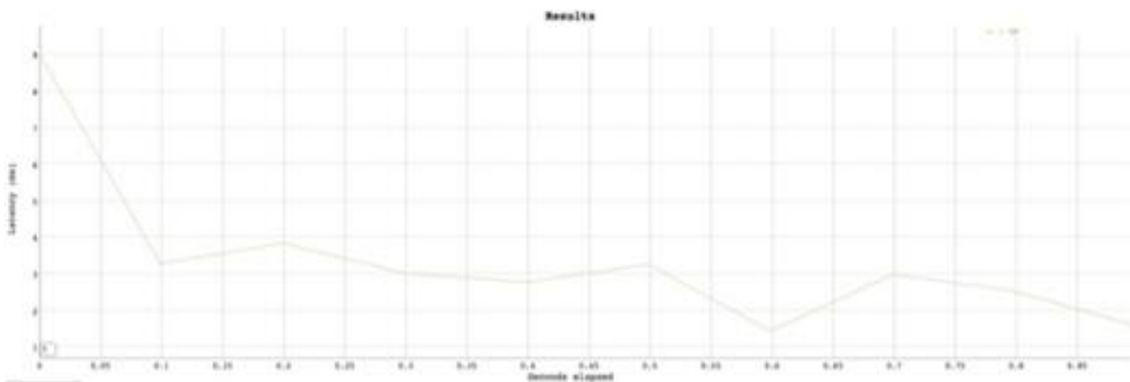

Fig. 9b Latency when a single instance calls to Redis cache before Postgres

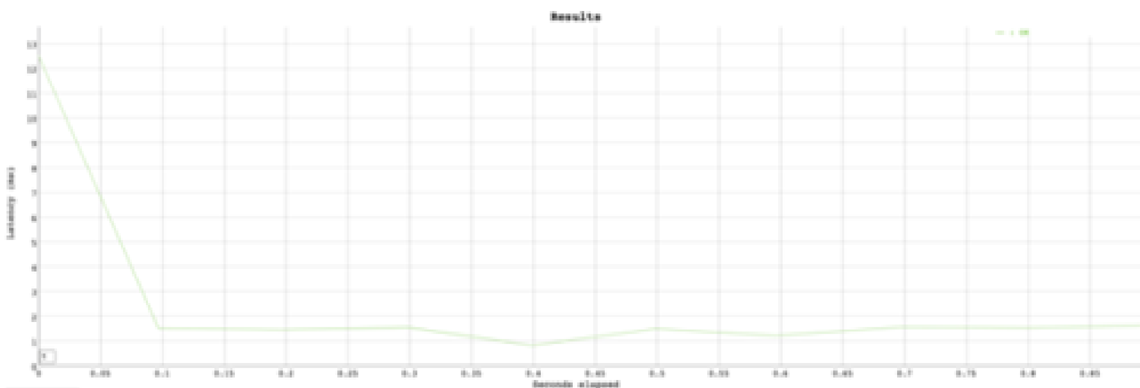

Fig. 10c Latency when a single instance calls to its own in-process memory cache before Postgres

Fig. 8a, 8b and 8c show the result of the experiment for the first scenario where there is only a single service instance to handle the load and HTTP requests. As we can see, the latency result from Fig. 8c shows the best stability and lowest reading. This is as expected due to the data being retrieved directly in its own process memory instead of having to call to an external application in a different host port. In Fig. 8a, the calls to Postgres directly also gives a better subsequent performance after the first call, this is expected due to optimization done by Postgres internally by default, as we discussed in Section II, we did not tweak any configuration of the shared buffer or operating system cache as we want to have a similar configuration as in production environment.

The second and subsequent calls for Fig. 8a shows the latency to be below 5ms with 4.88ms the highest and 2.69ms the lowest. While for Fig. 8b, the second and subsequent calls give latency below 4ms with 3.84ms the highest and 1.44ms the lowest. The best latency result is shown in Fig. 8c as expected, where second and subsequent calls give a constant and stable latency below 2ms, with 1.64ms being the highest and 0.83ms the lowest.V.

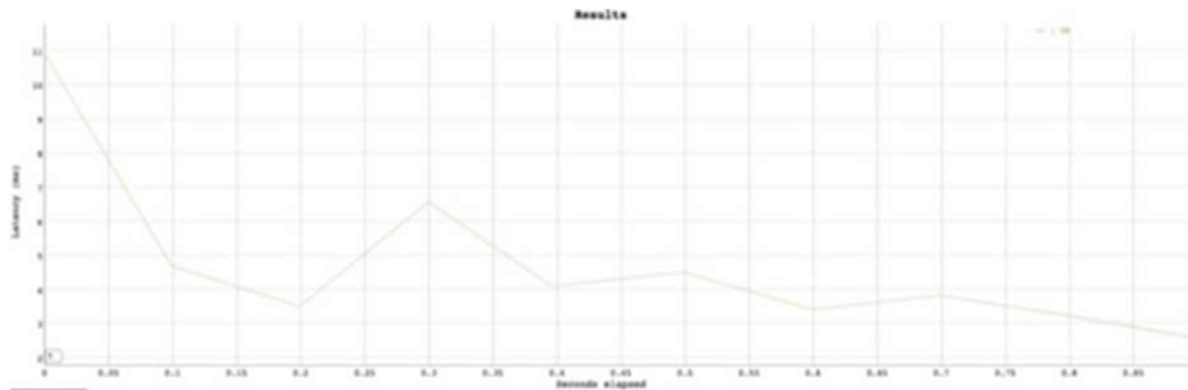

Fig. 9a Latency when three service instances directly calls Postgres database

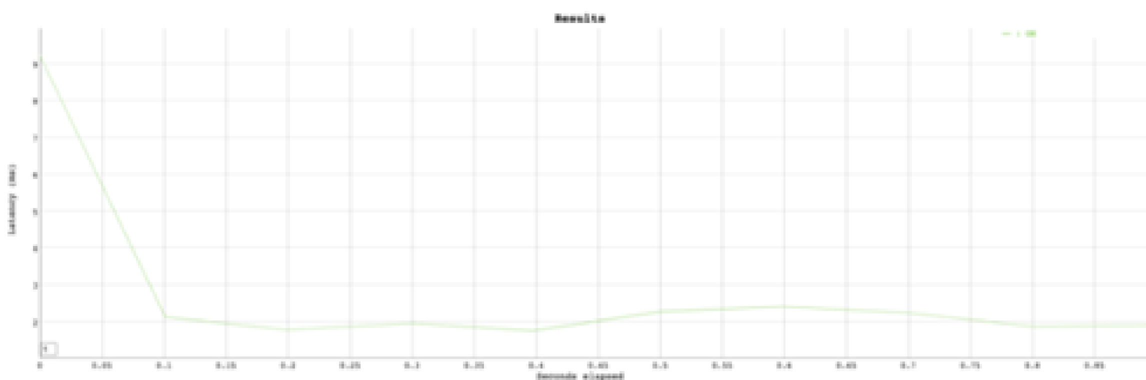

Fig. 9b Latency when three service instances calls to Redis cache before Postgres

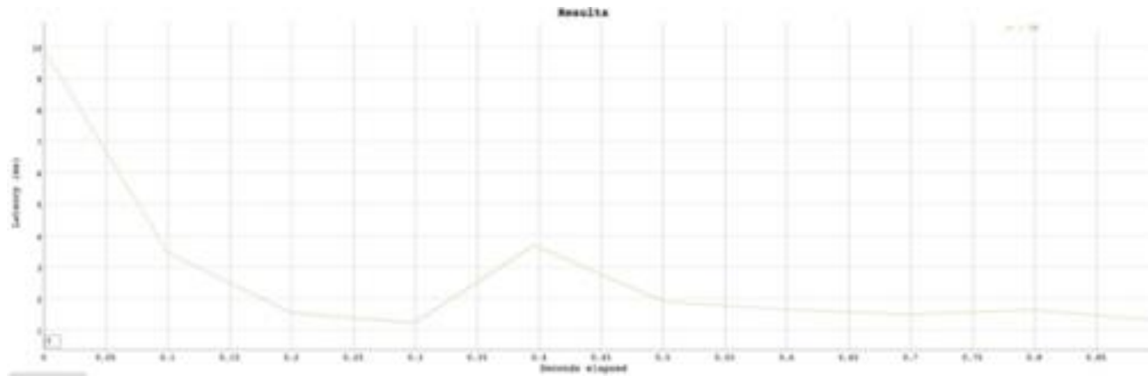

Fig. 9c Latency when three service instances calls to its own in-process memory cache before Postgres

Fig. 9a, 9b and 9c shows the result of the experiment for the second scenario where there are three service instances to handle the load and HTTP requests. As seen from the result, Fig. 9b gives the best and stable latency for second and subsequent requests. This is expected as each service instance calls to the same Redis cache for subsequent calls, compared to internal cache where each would have to call its own in-process memory cache first before being able to show improvement.

As shown in Fig. 9a, the second and subsequent calls give a latency of below 7ms with 6.57ms being the highest and 2.6ms being the lowest. In Fig. 9b, we see Redis shows its advantage with multi instance set up, where it gives a stable and constant latency of below 3ms, with 2.41ms being the highest and 1.75ms being the lowest. From Fig. 9c, we see a slightly worse performance for internal in-process memory cache for multi instance set up compared to single instance as shown in Fig. 8c. It gives below 4ms for its second and subsequent calls with 3.71ms being the highest and 1.23ms for the lowest latency call.

Postgres database default optimization helps tremendously with reducing the latency as shown in the result in Fig. 9a and 9c. For Fig. 9c, since the load is being distributed to three instances, the subsequent instances might not have the data in their own in-process cache and requires them to make its own call to Postgres database. As for Fig. 9a, even though other new instances are making the direct call to Postgres itself, Postgres optimized by not having to access disk for the data if the calls refer to the same data by using its shared buffer cache and operating system cache. The result for the experiment run in this section can be further summarized in Table 1 and Table 2.

Table 1: Latency for each request with different caching method with single instance service

HTTP Request	Cache method		
	No cache	Redis	In-Process memory
1	9.49ms	9.03ms	12.52ms
2	4.05ms	3.28ms	1.52ms
3	4.57ms	3.84ms	1.47ms
4	4.36ms	3.02ms	1.56ms
5	2.69ms	2.77ms	0.83ms
6	4.88ms	3.26ms	1.51ms
7	4.65ms	1.44ms	1.23ms
8	4.67ms	2.99ms	1.57ms
9	4.69ms	2.55ms	1.54ms
10	4.44ms	1.59ms	1.64ms

Table 2: Latency for each request with different caching method with 3 instances service

HTTP Request	Cache method		
	No cache	Redis	In-Process memory
1	10.93ms	9.22ms	9.93ms
2	4.71ms	2.13ms	3.47ms
3	3.5ms	1.78ms	1.55ms
4	6.57ms	1.96ms	1.23ms
5	4.1ms	1.75ms	3.71ms
6	4.51ms	2.28ms	1.92ms
7	3.42ms	2.41ms	1.65ms
8	3.83ms	2.25ms	1.5ms
9	3.24ms	1.86ms	1.64ms
10	2.6ms	1.91ms	1.32ms

From the results shown in this section, we can see that for three service instance setup as we proposed in Section II, Redis cache would give the best performance and stability compared to other methods. For single service instance, internal in-process memory cache gives the best performance but in multi service instance setup it gives a slightly lower performance. Furthermore, data invalidation would need to be handled in each service instance and this would be tedious to handle when we increase the number of instances and it is prone to error as improper handling of data invalidation can cause wrong data to be retrieved by the end user.

Conclusion:

In this paper, we have presented the experiment to find the best cache method to be used in our proposed microservice architecture. Based on the result, we can conclude that Redis is the best cache method to be used, not only due to performance, but also due to the ease of logic to invalidate any existing cache after any mutation changes happens to the underlying data.

Acknowledgment

The authors would like to thank the Ministry of Higher Education, Malaysia for financial support through the Long-term Research Grant Scheme (LRGS) 600-RMC/LRGS 5/3 (001/2020).

References

1. M. Villamizar et al., "Cost comparison of running web applications in the cloud using monolithic, microservice, and AWS Lambda architectures," Service Oriented Computing and Applications, vol. 11, no. 2, pp. 233-247, 2017, doi: 10.1007/s11761-017-0208-y.
2. D. Taibi, V. Lenarduzzi, C. Pahl, and A. Janes, "Microservices in agile software development: a workshop-based study into issues, advantages, and disadvantages," in Proceedings of the XP2017 Scientific Workshops, 2017, pp. 1-5.
3. M. Viggiato, R. Terra, H. Rocha, M. T. Valente, and E. Figueiredo, "Microservices in practice: A survey study," arXiv preprint arXiv:1808.04836, 2018.
4. I. Asrowardi, S. D. Putra, and E. Subyantoro, "Designing microservice architectures for scalability and reliability in ecommerce," Journal of Physics: Conference Series, 2020.
5. W. Hasselbring and G. Steinacker, "Microservice Architectures for Scalability, Agility and Reliability in E-Commerce," presented at the 2017 IEEE International Conference on Software Architecture Workshops (ICSAW), 2017.

6. H. Khazaei, N. Mahmoudi, C. Barna, and M. Litoiu, "Performance Modeling of Microservice Platforms," IEEE Transactions on Cloud Computing, pp. 1-1, 2020, doi: 10.1109/tcc.2020.3029092.
7. Y. Gan et al., "An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems," presented at the Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, 2019.
8. N. A. Farooqui, A. K. Mishra, and R. Mehra, "IOT based Automated Greenhouse Using Machine Learning Approach", Int J Intell Syst Appl Eng, vol. 10, no. 2, pp. 226–231, May 2022.
9. M. Villamizar et al., "Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud," in 2015 10th Computing Colombian Conference (10CCC), 2015: IEEE, pp. 583-590.
10. D. Taibi, V. Lenarduzzi, and C. Pahl, "Architectural patterns for microservices: A systematic mapping study," in CLOSER 2018: Proceedings of the 8th International Conference on Cloud Computing and Services Science.
11. A. Jindal, V. Podolskiy, and M. Gerndt, "Performance modeling for cloud microservice applications," in Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering, 2019, pp. 25-32.
12. H. Dinh-Tuan, M. Mora-Martinez, F. Beierle, and S. R. Garzon, "Development frameworks for microservice-based applications: Evaluation and comparison," in Proceedings of the 2020 European Symposium on Software Engineering, 2020, pp. 12-20.
13. I. A. S. D. P. E. Subyantoro, "Designing microservice architectures for scalability and reliability in ecommerce," Journal of Physics: Conference Series, 2020.
14. A. Singleton. (2016) The Economics of Microservices. IEEE Cloud Computing.
15. D. S. Linthicum, "Practical use of microservices in moving workloads to the cloud," IEEE Cloud Computing, vol. 3, no. 5, pp. 6-9, 2016.
16. A. B. YILMAZ, Y. S. TASPINAR, and M. Koklu, "Classification of Malicious Android Applications Using Naive Bayes and Support Vector Machine Algorithms", Int J Intell Syst Appl Eng, vol. 10, no. 2, pp. 269–274, May 2022.
17. H. Suryotrisongko, D. P. Jayanto, and A. Tjahyanto, "Design and development of backend application for public complaint systems using microservice spring boot," Procedia Computer Science, vol. 124, pp. 736-743, 2017.
18. S. Newman, "Building Microservices," in Building Microservices, vol. 2, no. 2): O'Reilly Media, 2020, ch. What Are Microservices?
19. M. Mazzara, N. Dragoni, A. Bucchiarone, A. Giaretta, S. T. Larsen, and S. Dustdar, "Microservices: Migration of a Mission Critical System," IEEE Transactions on Services Computing, pp. 1-1, 2018, doi: 10.1109/tsc.2018.2889087.
20. V. Zakhary, D. Agrawal, and A. El Abbadi, "Caching at the Web Scale," presented at the Proceedings of the 26th International Conference on World Wide Web Companion - WWW '17 Companion, 2017.
21. Gill, D. R. . (2022). A Study of Framework of Behavioural Driven Development: Methodologies, Advantages, and Challenges. International Journal on Future Revolution

in Computer Science &Amp; Communication Engineering, 8(2), 09–12. https://doi.org/10.17762/ijfrcsce.v8i2.2068

22. R. K. Singh and H. K. Verma, "Redis-Based Messaging Queue and Cache-Enabled Parallel Processing Social Media Analytics Framework," The Computer Journal, vol. 65, no. 4, pp. 843-857, 2022.
23. W. P. S. Puntheeranurak, "A Comparative Study of Relational Database and Key-Value Database for Big Data Applications," in International Electrical Engineering Congress, 2017, vol. 5.
24. S. L. H. J. M. Shi, "Redis-based Web Server Cluster Session Maintaining Technology," in International Conference on Natural Computation, Fuzzy Systems and Knowledge Discovery, 2017, pp. 3065-3069.
25. Q. Liu and H. Yuan, "A High Performance Memory Key-Value Database Based on Redis," J. Comput., vol. 14, no. 3, pp. 170-183, 2019.
26. G. I. Muradova, "Security of personal medical data for the Redis concept," Problems of information technology, pp. 62-68, 2018.
27. M. Kusuma and R. Ferdiana, "Evaluasi performa web server menggunakan varnish http reserve proxy dan redis database cache," Prosiding SENIATI, pp. 260-B. 264, 2016.
28. R. Patel, "Data + Education. Redis Is a Cache or More?," EasyChair Preprint, 2021.
29. D. Yang and F. Kai, "The Optimization Mechanism Research of Distributed Unified Authentication Based on Cache," in 2017 14th Web Information Systems and Applications Conference (WISA), 2017: IEEE, pp. 297-300.
30. A. Blankstein, S. Sen, and M. J. Freedman, "Hyperbolic caching: Flexible caching for web applications," in 2017 USENIX Annual Technical Conference (USENIX ATC 17), 2017, pp. 499-511.
31. Pepsi M, B. B. ., V. . S, and A. . A. “Tree Based Boosting Algorithm to Tackle the Overfitting in Healthcare Data”. International Journal on Recent and Innovation Trends in Computing and Communication, vol. 10, no. 5, May 2022, pp. 41-47, doi:10.17762/ijritcc.v10i5.5552.
32. A. Vattani, F. Chierichetti, and K. Lowenstein, "Optimal probabilistic cache stampede prevention," in Proceedings of the VLDB Endowment, 2015, vol. 8, pp. 886-897.
33. S. Li, H. Jiang, and M. Shi, "Redis-based Web Server Cluster Session Maintaining Technology," in International Conference on Natural Computation, Fuzzy Systems and Knowledge Discovery, 2017, pp. 3065-3069.
34. N. Meghanathan, "Review of Access Control Models for Cloud Computing," presented at the Computer Science & Information Technology (CS & IT), 2013.
35. Nouby M. Ghazaly, M. M. A. . (2022). A Review on Engine Fault Diagnosis through Vibration Analysis . International Journal on Recent Technologies in Mechanical and Electrical Engineering, 9(2), 01–06. https://doi.org/10.17762/ijrmee.v9i2.364
36. Yargholi M. System Level Simulation of Energy-Detection Based UWB Receivers. sjis. 2020; 2 (4) :10-14, URL: http://sjis.srpub.org/article-5-83-fa.html