Architecture for the 15-Minute City

Conference Paper · September 2021

CITATIONS
0

READS
18

3 authors, including:

Nuno Cruz
Instituto Politécnico de Lisboa
16 PUBLICATIONS **27** CITATIONS
SEE PROFILE

Nuno Datia
Instituto Politécnico de Lisboa
26 PUBLICATIONS **44** CITATIONS
SEE PROFILE

Architecture for the 15-Minute City *

Leonardo Melo[1], Nuno Cruz[1,2], and Nuno Datia[1,3]

[1] FIT - Future Internet Technologies, ISEL - Instituto Superior de Engenharia de Lisboa, Instituto Politécnico de Lisboa
{leonardo.melo, nuno.cruz, nuno.datia}@isel.pt
[2] LASIGE, Faculdade de Ciências, Universidade de Lisboa
[3] NOVALINCS, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa

Abstract. Finding a new home in a large city has increasingly become more complex, as people are concerned with not only the estate itself but also with its surroundings. The **15-Minute City** concept, thinks of cities as a way to ensure that their residents within a 15-minute radius, can fulfill six essential functions: home, work, commerce, health care, education, and entertainment; which varies according to a chosen travel mode. To help people find properties that would fit them according to this concept, we have built an application that aims to provide an intuitive user interface that guides the user through the important decision of buying a house. To present relevant information to the user, we gathered information from three relevant sources, each with its unique challenges. The first source is estate data, extracted from the websites of local real estate agencies; second, city data, with points of interest relevant to the six essential functions mentioned previously; and at last, the user data provided by the user itself through our profile system which lets us understand his needs. System-wise, we built a reliable application following the microservices architecture guidelines, which future proofs our solution by segregating each part of the project and allowing it to scale easily, development and feature wise. The application scrapes, ingests, transforms and stores data regularly. The complete infrastructure is containerized using Docker and orchestrated by Kubernetes. With this application, we provided a scalable solution that allows users to select the best real estate taking into consideration the surrounding environment, tailored to their day-to-day needs, something that, as far as we know, is a novelty on real estate agency websites.

Keywords: 15-Minute Cities · Cloud computing · Geographical systems · Distributed systems · Microservices.

1 Introduction

Looking for a new place to live increasingly became more difficult not only because the home of choice matters, but also everything that surrounds it as

* We would like to thank the CGIUL team for its support under the "Data at the service of Lisbon" Protocol. This work is supported by LASIGE (UIDB/00408/2020) and by NOVA LINCS (UIDB/04516/2020) with the financial support of FCT-Fundação para a Ciência e a Tecnologia, through national funds.

well [7]. Each family has its own needs; factors such as school districts, commerce and green spaces, end up being a major factor of decision. Our project aims to help the future owner (the user of the application), by understanding his needs and translating them into the most appropriate list of available houses. All of this merges well with the concept of the **15-minute city**, which "*may be defined as an ideal environment where most human needs and most desires are located within a travel distance of 15 minutes*" [3], with its radius varying depending on the transportation used (walk or bike). Even though that in most of the definitions available, the concept doesn't consider cars as a travel option, due to the range extended by several kilometers, we have decided to give the user that decision while presenting the disadvantages of such choices.

Given the broad aspect of our project, this paper focus on the architecture created to support the project – how we made it scale automatically based on the necessity and how the data is presented to the user with the help of our system. Since our priority was to make the solution scalable and highly available, one of the first decisions made was to opt for a microservices based architecture, which allows for independently scalable and deployable services. Having everything isolated, also contributes to reduced downtime through fault isolation. To deploy this solution, we set as a requirement that each of our services should be containerized with Docker and deployed in Kubernetes. With Kubernetes, we had at our disposal tools that allowed each service to be deployed as pod that could automatically be scaled up or down in a matter of seconds, which allows us to target specific microservices that may require an extra performance boost. As the application requires real estate information and points of interest (POI) surrounding such estates, we had to come up with a solution to gather all this information. We designed a scraping solution to collect data from multiple real estate websites. For relevant POI, most of the information was publicly available, so we gathered the data needed from Lisbon's open data platform, **Lisboa Aberta** [4].

In Figure 1, we can see an overview of our project architecture and its most important components. The first component is the data source, representing the real estate websites that are scraped by our solution and an the corresponding data inserted into the databases of one of our services. The microservices section is composed by all the required services, namely: *User*, used to access the users' personal information, *Metrics*, responsible for all the index logic, *Estate*, access to all the property data, *Search* allows us to perform text queries and easily find locations by name, *Parameter* gives us access to the POI information, and *Auth*, provides authentication and authorization, enabling user access to the website and to our services. All services have their own databases, designed to support service specific use cases, and message brokers to allow inter-service communication. All services are funneled in through the API gateway.

This article presents a resilient and scalable solution for a real estate application, and a scrapping solution that extracts and integrates data from multiple public resources. Moreover, it is shown how the data can be presented to entice

[4] See lisboaaberta.cm-lisboa.pt

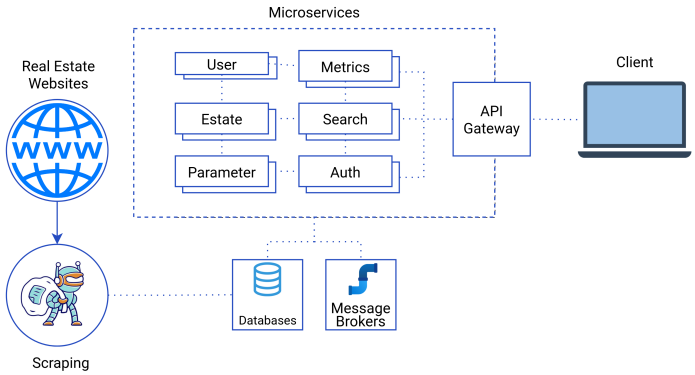


Fig. 1: Project architecture

the user during his estate selection. Finally, we propose metrics which could be used as a standard for system scaling adjusted to different thresholds.

The article is structured in the following way: Section 2 describes similar existing projects and how they compare to our solution; Section 3 describes the approach taken to the create the backend and an insight into the design of the scrapping solution; Section 4 details some of the struggles with data gathering and how we have approached monitoring in our system; After detailing the behind the scenes, section 5 describes how all of this work is then shown to the user; Section 6 features the results of the stress testing we have done to mimic our architecture in a real world scenario; and at last, Section 7 concludes our article while providing some insight into possible future work.

2 Related Work

There are multiple projects in the literature that employ a microservice architecture using kubernetes [1]. The following paragraphs describe some of the work we found representative and relevant for our work.

A. Alexandrescu [2] shows the development of a similar platform based on microservices with data gathering as a basis. One of the caveats of the solution is the dependency on obtaining their source URLs from a database. Our solution requires us to take a step back and deal with the extraction itself by crawling through the necessary websites, allowing us to add any required source in the future. K. Milkovich et al [6] are responsible for ZenDen, a personalized house searching application, which tries to modernize the way we search for houses in today's market. In their approach, they focused the application around a swipe based interface which displays estates based on the results of their deep learning based recommendation system. This recommendation system learns by getting fed with user view history and comparing house images from this ads. Our solution tries to revamp the market through the use of the 15-Minute City concept as a basis for our recommendation system, where as input we introduce the users

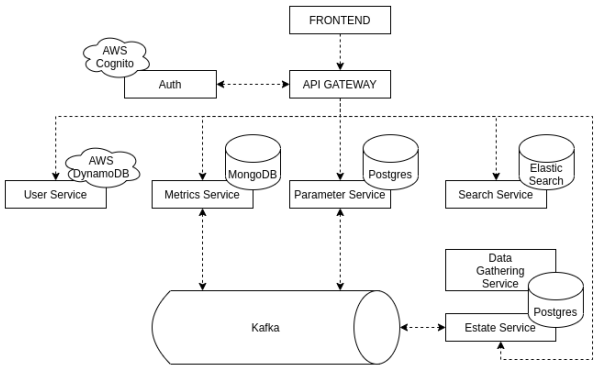


Fig. 2: System architecture

preferred characteristics. F. Rossi et al [9] discusses the use of kubernetes as support for a microservice architecture, and discusses the multiple techniques available for pod scaling, exploring approaches based in reinforcement learning-based policies. Concluding that the default scaling solution, Horizontal Pod Autoscaler (HPA) works based on Central processing unit (CPU) which may not be ideal metric to scale latency sensitive applications. There's also been some real case studies of microservices implementations in companies. **Trulia** [4], a real estate website after implementing a microservice architecture discovered that they had no unified approach to observability. Each team was using different sets of tools, increasing development cost, license cost and making every other team request authorization access to each one of these tools, making it extremely difficult to track and diagnose problems. In our approach, we have unified all our observability through the use of prometheus, every single microservice can be monitored with the same tool and its access is as simple as a click.

3 Architecture

To satisfy the requirements of the project, a cloud application was created that follows a microservices architecture [5], depicted in Fig. 2. The goal was to design a highly modular system, with every service being small and focused, so each can be an independent application. Since the system is broken up into smaller chunks, it has the added benefit of easier failure isolation, as it allows for faster recovery by finding problems faster. With the increase in the number of microservices, it becomes a challenge to route traffic to the services. It was solved by implementing an API Gateway [12], a special server which acts as the only entry point to the microservices located on the system boundary, encapsulating the internal aspects of the system. However, being the only entry point into the system, it can be overloaded with requests, prone to bottlenecks and becoming the single point of failure in our system.

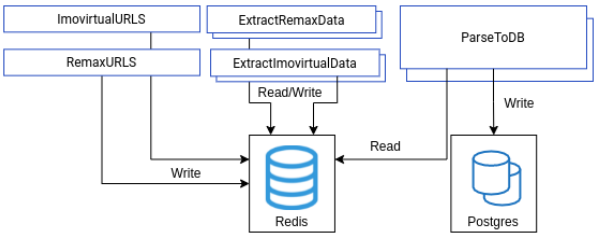


Fig. 3: Scraping Architecture

One of the characteristics of the microservices architecture [8], is that services are loosely coupled, and one way we have achieved loose coupling is by giving each service his own database, being the exceptions the Data Gathering Service (write-only) and Estate Service (read-only), which share the same database. Since microservices are language-neutral, there must be also a language-neutral communication form, as such the services communicate with the outside through the gateway with HTTP via REST. For inter-service communication we used Kafka [10], an event streaming platform that was used as a message broker. A major advantage of this architecture is its adaptability; the way it was designed allows our system to be scaled and changed to support different numbers of users.

Due to the nature of the problem, it became apparent that data was the most important asset in this application, and as the data scraped was structured and consistent in format, we opted for a relational model. Since there was no publicly available estate information, we had to design an appropriate solution that could be easily scalable and would allow the addition of new websites to be scraped. Depicted in Fig. 3, we can see the architecture behind the solution. The scraping solution is split into three sections, represented by three columns in Fig. 3 where each has a different objective:

1. Each pod is responsible of gathering estate listing URLs for his specific website and writing them to an website specific named queue (e.g. remax-queue). The scraping takes place every 3 to 7 days, and there's rarely more than 30 pages to extract data from within this interval of time.
2. During this step the processes are actively waiting for data in the queue, so they can start gathering information from each estate listing. Since each website has special requirements, we decided to split them by their respective websites, this way they could be developed in parallel. Each of this services follow the same model, output wise, and as such the results are all written into the same queue.
3. During the last phase, pods wait for data to arrive to the queue. After reading it, it is processed, cleaned and if valid, stored in the database;

Since the estate data is the backbone of this project, it was important to ensure that it would always be ready to use and backed up. For that reason,

the cluster is composed by two Postgres[5] instances running in a master-slave composition, maintaining the data replicated, and if one ever goes down, one new pod will start up and be assigned as a slave to the new master (the instance that survived). To backup data, a CRON job periodically runs and backups up data remotely. For the queue, a Redis[6] instance was used and since losing one scraping job was not critical, we decided that it was not worth to implement replication as seen with the Postgres cluster.

4 Backend

This section details the implementation of our backend solution, starting with data gathering in Sub-section 4.1, followed by an overview of monitoring and observability in Sub-section 4.2 where we explain how we made it possible to monitor the system efficiently.

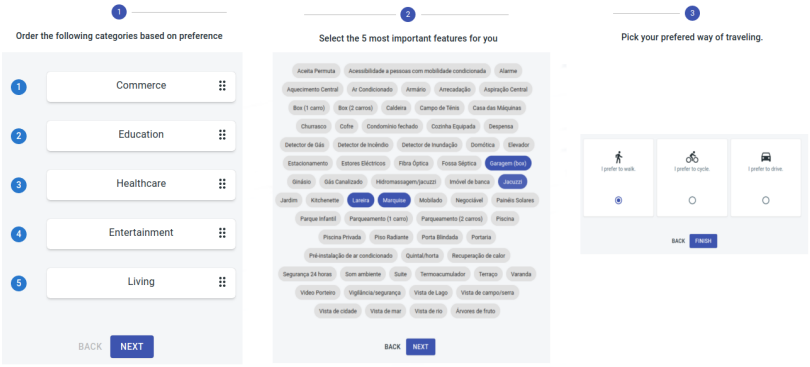
4.1 Data Gathering

As most of the project depends on geographical information to work, including estate and POI coordinates, as well as geometries to represent the boundaries of the zones, both PostgreSQL instances, represented in the architecture in Fig. 2, have been extended with the PostGIS extension. It allows us to perform topological queries and easily obtain which geometries intersect distance-based queries, to name a few. As most scraping solutions [11], to ensure sites are not overloaded with our requests and that no prohibited data is accessed, we have followed the 'robots.txt' rules defined for each website. Through XPATH expressions, we have found and extracted the relevant information necessary to present the estates in our platform. Most pages today try to avoid overloading their own services by sending data to the client only when necessary, as such, while trying to scrape each URL we have noticed missing information. This data was loaded based on user interaction, to overcome this we had to use automation tools such as **Selenium**, allowing us to simulate a real user and load the necessary data by engaging the page with movement.

4.2 Monitoring and observability

One of the disadvantages of the microservices is the amount of information it requires the developer to keep up with, as such we've decided to use **Prometheus**[7] as a way to monitor the entire system. It allowed us to overcome this hurdle and we were able to leverage the metrics collected and use them to make scaling decisions. But prometheus data wasn't directly accessible by the **Horizontal Pod Autoscaller**. We had to introduce to our solution a prometheus-adapter[8],

[5] See www.postgresql.org
[6] See redis.io
[7] See prometheus.io
[8] See helm-charts, prometheus-adapter



(a) (b) (c)

Fig. 4: User preferences stepper

which does it for us. This way, we had only to write the prometheus queries straight into our adapter, which in turn will fetch all the data for us. To make the data available in prometheus, and thus make it available to query, we had to define the instrumented jobs that it uses to scrape the metrics. This job lets us define our scrape interval, the location of our prometheus instance and the location of the metrics we wish to scrape, along with some metadata. Even though all the data was now available, there was no easy way to visualize it. To fix this we decided to incorporate **Grafana** into our solution, an open source visualization and analytics software. With it we were able to query, visualize and easily explore our metrics, which was instrumental to the optimization of our scaling solutions.

5 Frontend

In this section we have highlighted the different visual components of the application, as an implementation of our architecture. Starting with **User Profile** (5.1), where the user is guided to a profile creation and Overview (5.2), where query results are displayed.

5.1 User profile

Since the project relies heavily on a user profile, we have tried to simplify the registration process so it would not overwhelm the user and discourage him from completing the process. After the user creation and authentication, the first page presented to the user is the profile customization, allowing us to have an idea of what the user is looking for when searching for a new home. The user must go through three steps as seen in Fig. 4. In the first one, Fig. 4.*(a)*, the user order the categories of the '15-minute city' concept by preference using a drag and drop list. The second one, Fig. 4.*(b)*, targeted at the estate itself, allows the user

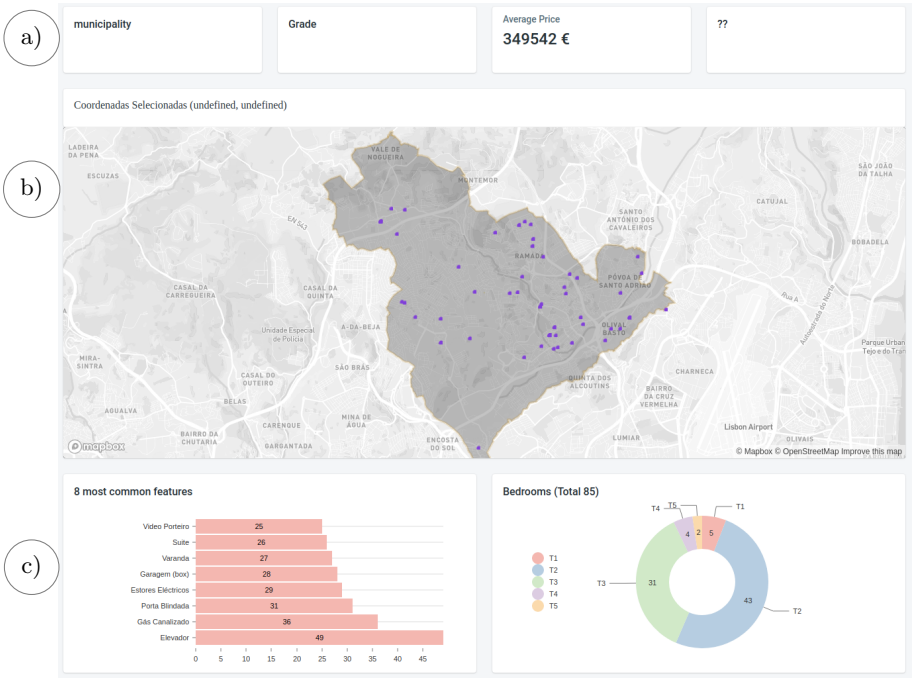


Fig. 5: Current user interface

to pick what is essential when searching for a new home. And, for the last step, Fig. 4.*(c)*, the system asks the transportation means the user intends to use.

5.2 Location overview

After creating his profile, the user can now use the search bar to look for homes in a zone of his preference (ranging from parishes to districts). Consequently, he is presented with the relevant contextualized information available for that specific zone, as shown in Fig. 5:a) General information about the zone such as the name, average price; b) Map, which displays estates that meet the user requirements and relevant POI; c) Estate statistics such as most common features and the distribution of rooms.

6 Evaluation

During the evaluation phase, stress tests were performed with the Parameter microservice, with the goal of finding the limit for the number of users and defining the best thresholds to scale the solution when required. We have chosen a service that has a similar load to the others in the system. This way our findings can be utilized as efficiently as possible. However, in cases such as the User service

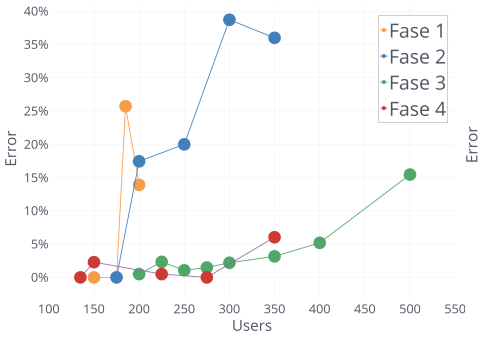


Fig. 6: One Pod

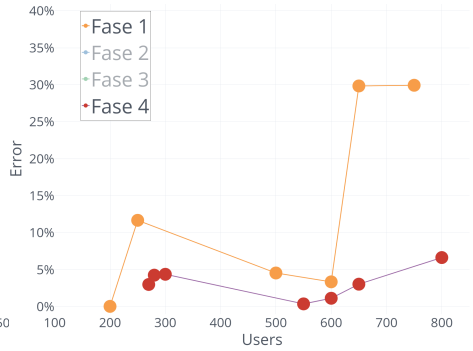
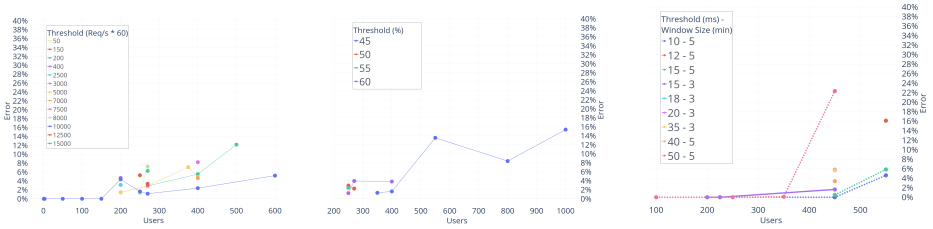


Fig. 7: Two Pods

that depends on external providers (i.e., AWS), results may vary when faced with the same tests. Hardware wise, the application is currently running in a cluster with 12 cores and 24GB of RAM. To stress test the system, we have used **Apache JMeter**[9], a tool used to load test applications and measure system performance, as it allows us to setup multiple testing scenarios with different numbers of users, ramp up periods and test duration. As stated previously, the goal was to understand how many users could the system handle, as such it was required to first understand the limit of users for a single pod, for two pods and, based on those results, understand the best metric to setup automatic scaling between one and two pods. For consistency purposes, all tests had a ramping period of 60 seconds, where the users are added periodically within that time range and then a 240 seconds period of user simultaneous interaction. Since this is not a critical application, where losing requests has no consequences besides a bad user experience, tests with a minimum of 98% success rate are considered successful. The scaling evaluation was split into three separate tests. During the first one it was observed how the system would react when scaling based on the number of requests received; followed by the classical approach of using CPU resources, the only default tool available in Kubernetes; and finally, based on the work of F. Rossi et al [9], we tried a response time approach.

Results wise, as depicted in the Fig. 6 to 8c, each circle is the result of an average of 5 tests, all done under the same circumstances for the specific situation being tested. The tests were performed over four phases, each iterating on the previous one, as seen in Figures 6 and 7.The first phase (orange) allowed us to understand that abundant resources would not be enough to solve all our problems as the results were subpar, with an error rate above 10% for 200 concurrent users. For this test, we had setup our pod with the initial values of 800 millicores (m) of CPU and 800 MebiBytes (MiB) of RAM, even though it was the phase with the highest memory value. During this phase, we noticed that only 50% of the pod memory was being used, so for Phase 2 (blue), we

[9] See jmeter.apache.org



(a) Increase of HTTP Requests

(b) Percentage of CPU usage

(c) Rate of MaxResponseTime

Fig. 8: Metrics testing

increased Java Virtual Machine (JVM) memory options to access 75% of the available RAM (600Mb out of 800MiB). It ended up not affecting the error rate, but it allowed us to optimize memory usage, saving valuable hardware resources.

From the previous test, we concluded that the ideal value of memory would be 400Mb for JVM and 500MiB for pod resources. After finding the ideal memory values, we decided to increase CPU resources to 1 full core per pod, which decreased the error rate to sub 5% for 200 to 350 users as seen in phase 3 (green). It also greatly improved the boot up speed from 50 seconds to sub 30 seconds on average. During phase 3 we've noticed that after booting up, pods were using 300MiB of RAM and would slowly increase until an eventual crash. As a fix, we have tried a new garbage collector, Garbage First Garbage Collector (G1GC), which actually started releasing memory when it is not needed which increased our up time and error rate, as seen in the results of phase 4 (red). Providing the best results with **275 users with 0% error rate.** Upon finding the optimized pod configurations, we tested them in our 2 pods solution (Fig. 7, represented in red), which improved our results allowing up to **600 users with a sub 2% error rate**. Having discovered the ideal configuration and finding the system limitations for 1 and 2 pods, 275 and 600 users respectively, we can now focus on how to automate the scaling process. The following metrics have been chosen to test as scaling parameters: 1. **http_server_requests_seconds_count** — represents the number of requests received at a certain endpoint at a given time; 2. **CPU usage** — represents the CPU processing power being used at a given time; 3. **http_server_requests_seconds_max** — represents the maximum request duration during in a rolling window, as such the purpose is to measure the worst outlier. And, as a way to optimize our solution, the following rules have been defined: 1. Scaling up to two pods, only when the load is higher or similar to the ceiling found during our one pod tests (e.g.: should not scale with less than 200 users); 2. The pods should be given a grace period to scale up, where they can handle the full load while the second pod is loading up without losing any requests; 3. Handle almost the same load as the experiment that ran with two deployed pods. Over the first metric, Fig. 8a, the Prometheus function **Increase** has been applied with a 1 minute window, which calculates the

per-second average rate of increase of the time series in that interval and multiplies its values by the defined interval in seconds (60). As seen in our figure, most of the tests with more than 300 users had an error rate above 7.5%, independent of the chosen threshold. We also noticed that even when using values below 200 (way below our 1 pod threshold found previously), the pods would scale anyway, which violates one of our requirements. Even though the metric didn't meet the defined requirements, it allowed us to realize that we were losing requests during the scaling period because Kubernetes was routing requests to the newly instantiated pod, as it was declared ready when JVM launched but before Spring finished booting up. We then implemented new readiness and liveness probes, based on the micrometer metrics, which allowed us to declare the pod as ready and alive, only when spring actually launches and is ready to process requests. The second metric, **CPU usage** Fig. 8b, displayed poor results since CPU usage was for the most part, always at max load, which caused our cluster to immediately scale even though it was unnecessary. For the final metric, **http_server_requests_seconds_max**, we applied a prometheus rate which calculates the per-second average rate of increase within a certain interval. A 3 and 5 minute window intervals were tested, as observed in Fig. 8c. In the same figure, we can see that the 5 minute window produces good results, but, as the first metric tested, it would scale below 200 users, and given the window size adjusting the threshold wouldn't help because the test would be over by the time it would stabilize. We decided to reduce the window to 3 minutes, starting with an higher threshold (50) to try and avoid unnecessary scaling which backfire by never scaling, even when above 400 users. After trying different thresholds (40, 35, 20 and 18) while maintaining the number of users at 450, we had interesting results with an error rate below 5%, but still above the desired threshold of 2%. Our final try led us to the 15ms threshold, where it managed to hold a **sub 2% error rate with 450 users** and sustain, at least, 200 users without triggering the scaling process. As a final experiment, the test duration was increased to 10 minutes and the same error rate was maintained.

7 Conclusion and Future Work

In this article, we have presented two resilient and scalable architectures. One responsible for data gathering and another, a microservice architecture for the 15-Minute City application allowing data exploration. Both have been implemented, with latter also being made available through a frontend application. Scalability wise, to deal with the fluctuation in incoming users, we use automatic scaling through the use of Horizontal Pod Scaling, a tool made available by Kubernetes. As it had some limitations, metrics wise, we've had to implement some new metrics as a way to scale our system, which we've done with the help of Prometheus. After stress testing our solution, we were able to find our pod limitations: 275 users for 1 pod and 600 users for 2. And within those parameters, we tried to find the best metric to base our scaling on. During our scaling testing, the best metric found allowed us to handle 200 users without

requiring scaling and up to 550 after scaling. Our tests could be improved by adding more hardware, as given the complexity of the system, it would be hard to scale so many databases and services at the same time without it. As such, we focused our efforts in testing one of our microservices, in a way that could later be easily implemented on to the rest. Http_server_requests_seconds_max was the best performing metric of the ones tested. It measures the worst outlier, which allowed us to understand how consistent the system is performing and at any signal of deterioration, it allows us to scale the system right away.

As future work, we would like to deal with data duplication between multiple sources.We would also like to focus on the development of the index and, by proxy, the recommendation system to be able to launch our project fully functional. As well as to perform user tests to evaluate the system usability wise, trying to understand if it performs as expected.

References

1. Abdollahi Vayghan, L., Saied, M.A., Toeroe, M., Khendek, F.: Microservice based architecture: Towards high-availability for stateful applications with kubernetes. In: 2019 IEEE 19th International Conference on Software Quality, Reliability and Security (QRS). pp. 176–185 (2019). https://doi.org/10.1109/QRS.2019.00034
2. Alexandrescu, A.: A distributed framework for information retrieval, processing and presentation of data. In: 2018 22nd International Conference on System Theory, Control and Computing (ICSTCC). pp. 267–272. IEEE (2018)
3. CNU: Defining 15-minute city. online: `https://www.cnu.org/publicsquare/2021/02/08/defining-15-minute-city`. Last accessed 18 May 2021
4. Inc., T.: Microservice observability with istio. online: `https://www.trulia.com/blog/tech/microservice-observability-with-istio/`. Last accessed 2 July 2021
5. Jaramillo, D., Nguyen, D.V., Smart, R.: Leveraging microservices architecture by using docker technology. In: SoutheastCon 2016. pp. 1–5. IEEE (2016)
6. Milkovich, K., Shirur, S., Desai, P.K., Manjunath, L., Wu, W.: Zenden - a personalized house searching application. In: IEEE Sixth International Conference on Big Data Computing Service and Applications (BigDataService). pp. 173–178 (2020)
7. Montezuma, J., McGarrigle, J.: What motivates international homebuyers? investor to lifestyle 'migrants' in a tourist city. Tourism Geographies **21**(2), 214–234 (2019). https://doi.org/10.1080/14616688.2018.1470196
8. Richardson, C.: Microservices Patterns, vol. 1 (2018)
9. Rossi, F., Cardellini, V., Presti, F.L.: Hierarchical scaling of microservices in kubernetes. In: 2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS). pp. 28–37. IEEE (2020)
10. Shree, R., Choudhury, T., Gupta, S.C., Kumar, P.: Kafka: The modern platform for data management and analysis in big data domain. In: 2017 2nd International Conference on Telecommunication and Networks (TEL-NET). pp. 1–5 (2017). https://doi.org/10.1109/TEL-NET.2017.8343593
11. You, J., Lee, J., Kwon, H.Y.: A complete and fast scraping method for collecting tweets. In: 2021 IEEE International Conference on Big Data and Smart Computing (BigComp). pp. 24–27 (2021). https://doi.org/10.1109/BigComp51126.2021.00014
12. Zhao, J.T., Jing, S.Y., Jiang, L.Z.: Management of API gateway based on microservice architecture. Journal of Physics: Conference Series **1087**, 032032 (sep 2018)