Scalability Evaluation of the Sentilo Platform Running on Containerized Resources

Diogo C. P. Domingos[1], **Marcelo A. C Ismael**[2,3], **Guilherme Galante**[1], **Eduardo Prasniewski**[2], **Wesley K. G. Assunção**[4], **Marcio S. Oyamada**[1], **Luis C. E. de Bona**[3], **Edson T. de Camargo**[1,2]

[1]Western Paraná State University (Unioeste) – Cascavel, Brazil

[2]Federal Technology University of Paraná (UTFPR) – Toledo, Brazil

[3]Federal University of Paraná (UFPR) – Curitiba , Brazil

[4]North Carolina State University Raleigh (NCSU) – Raleigh, USA

`diogopaganinidomingos@gmail.com, {marceloismael,edson}@utfpr.edu.br`

`prasniewski@alunos.utfpr.edu.br,wguezas@ncsu.edu`

`{guilherme.galante, marcio.oyamada}@unioeste.br, bona@inf.ufpr.br`

Abstract. *A smart city software platform is an integrated environment that helps developers design, implement, deploy, and manage smart city applications. Sentilo is the platform selected to manage smart city solutions in a medium-sized city (Toledo, PR, Brazil) through a partnership between the local university and city administration. This paper aims to evaluate Sentilo's scalability under different workloads and hardware configurations for both data receiving and provisioning. Results show that Sentilo offers sufficient performance to support the planned and new applications in the city.*

1. Introduction

Cities are hubs for economic activity and human needs. With urban areas growing, specially in population, smart cities have emerged to address major challenges like disaster prevention, environmental quality, energy efficiency, security, traffic, and waste management [Lai et al. 2020]. The primary goal of smart cities is to improve the quality of life for their inhabitants [Sánchez-Corcuera et al. 2019]. For implementing smart cities solutions, information and communication technologies are indispensable [Santana et al. 2017], which requires supporting technologies such as the Internet of Things (IoT), Cloud Computing paradigms, Mobile Crowd Sensing and Computing (MCSC), Big Data, and Artificial Intelligence [Goumopoulos 2024]. The integration of the different solutions developed by different actors and using different technologies relies on fundamental services of the underlying software infrastructure. In this context, a *software platform* for smart cities is defined as an integrated middleware environment that supports software developers in the design, implementation, deployment and management of applications for smart cities [Santana et al. 2017, Pereira et al. 2022, Goumopoulos 2024].

Sentilo [Sentilo 2014] is the software platform chosen to manage the smart city solutions in Toledo, a medium-sized city in the interior of the state of Paraná, in Brazil. A partnership between the local university and the city council aims to implement the concept of smart cities in the city. To this end, a low-power, wide-area network using LoRaWAN technology has already been set up [Camargo et al. 2021]. Applications such as real-time monitoring of air and water quality, odours, as well as tracking of recycling trucks using low-cost sensors are under development [Camargo et al. 2021, Xavier et al. 2022, de Camargo et al. 2023, Roncaglio et al. 2023, Droprinchinski et al. 2023].

The Sentilo platform has already been installed and configured for managing the collected data. However, one of the open questions in our project is whether the performance of the Sentilo platform is sufficient to ensure the operation of the existing and planned new applications. Considering the large amount of data to be received and managed, the scalability of such systems is paramount. Scalability requires the software platform to accommodate the city's expansion and increasing data volumes [Goumopoulos 2024]. Although the Sentilo platform is designed to ensure scalability [Santana et al. 2017], to our knowledge, no studies have yet investigated its performance.

The aim of this work is to evaluate the scalability of the Sentilo platform using many test configurations. To achieve this, the platform was installed and configured in containers on virtual servers. For evaluation purposes, the servers were equipped with five different hardware resource configurations. The tests evaluate the insertion and consumption of data. The data was queried both via Sentilo and via ElasticSearch.

This paper is organized as follows. Section 2 presents related work. Section 3 describes the Sentilo Platform and its setup in virtualized resources. Section 4 provides the evaluation methodology and Section 5 the results obtained. Section 6 presents our conclusion.

2. Related Work

The scalability of smart city platforms is crucial for efficient data handling and service demands. This section chronologically presents pieces of works addressing platform scalability.

Ismail et al. [Ismail et al. 2018] evaluate the throughput, CPU and memory utilization, and robustness of the ThingsBoard and SiteWhere platforms under heavy sensor data loads using HTTP REST and MQTT protocols. JMeter was used for data generation and Prometheus for monitoring. The study concluded that ThingsBoard outperformed SiteWhere, except with MQTT, which had a higher error rate.

Araujo et al. [Araujo et al. 2019] tested FIWARE's vertical and horizontal scalability in a public cloud using an extended JMeter to generate IoT protocol loads. The testbed emulated large-scale data updates from IoT devices using MQTT and CoAP-based LWM2M protocols. The study identified bottlenecks and limitations in FIWARE components and their cloud deployment.

Del Esposte et al. [de M. Del Esposte et al. 2019] propose InterSCity, a microservices-based, open-source smart city platform. Using the InterSCSimulator to generate realistic workloads, experiments showed that the platform scales horizontally to handle large smart city demands while maintaining low response times.

Pereira et al. [Pereira et al. 2022] describe the Smart Geo Layers (SGeoL) platform for developing smart city applications. Experiments using virtual machines and

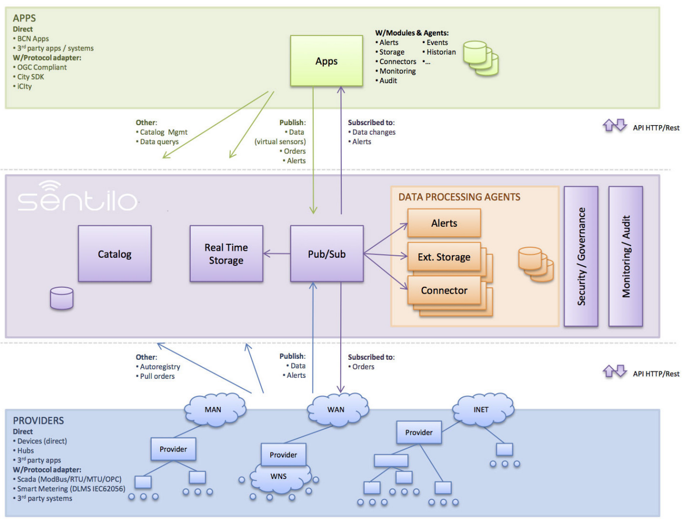


Figure 1. The Sentilo Platform Architecture (adapted from [Sentilo 2014]).

JMeter assessed its performance and scalability under numerous concurrent requests. Results showed that SGeoL scales horizontally to handle more users, but performance degradation increases linearly with the number of concurrent requests.

Tabassum et al. [Tabassum et al. 2023] evaluated the OpenCyberCity platform's scalability, reliability, and latency with millions of virtual IoT devices in a simulated environment. The testbed involved data collection from IoT devices, processing through an MQTT broker and Apache Kafka, and storage in Cassandra. A Python script created multiple threads to generate data points from virtual devices.

The works presented above share similarities with ours regarding the evaluation of different platforms scalability. However, our literature review did not find any studies evaluating the scalability of the Sentilo platform. In this sense, this is the main contribution of our work.

3. Sentilo Software Platform

Sentilo [Sentilo 2014] is a software platform designed to manage sensors and actuators, specifically tailored for smart cities, emphasizing openness and interoperability. Figure 1 presents Sentilo architecture consisting of three layers. The bottom layer contains sensors and actuators that are distributed throughout the city. In order to publish sensor data, the platform needs to set up providers, components, and sensors. A provider is an entity that manages devices (sensors). To do this, one or more sensors (e.g., temperature and humidity) must be connected to a component (e.g., an Arduino board or a smartphone). A component is then connected to a provider. For example, there would be one provider for the air quality application and each air quality station would be connected to one component. The data generated by a provider is called observations. To publish an observation, Sentilo uses a REST API via the HTTP protocol, which is generally transmitted via a wireless communication link.

The middle layer is the core of Sentilo. The Pub/Sub module allows users to publish and retrieve information and subscribe to the event system. The module is im-

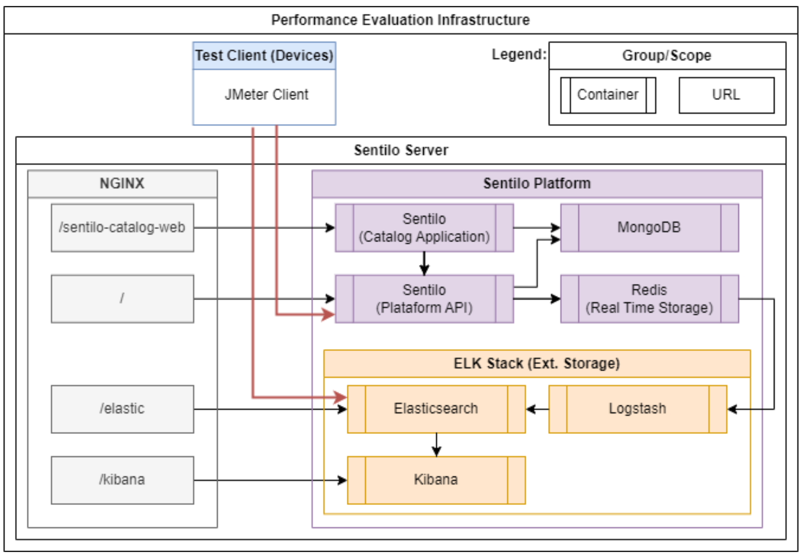


Figure 2. Sentilo's containers. ***Source: Figure created by the author.***

plemented as a standalone process that uses Redis,[1] an in-memory NoSQL database, as a central repository to store all received information. Redis is configured to make copies to disk at regular intervals. However, before an observation or a client request reaches Redis, it is processed by a pool of threads that are responsible for assigning it to a group of processes, the workers. The workers forward the request to Redis depending on the characteristics of the observations or client request. The Sentilo platform also offers the concept of agents that use the Redis publish-subscribe mechanism. With the help of agents, it is possible to persist data stored in Redis in long-term data stores such as ElasticSearch and OpenTSDB.

According to its documentation, Sentilo is horizontally and vertically scalable thanks to its Pub/Sub module in combination with Redis. With vertical scalability, the limits of the work queue and the workers are extended. With horizontal scalability, the load is distributed across multiple instances or server nodes. Finally, the upper layer (or APPS layer) communicates with the Pub/Sub module, especially when retrieving data (observation subscriber). However, they can also act as publishers, either as a virtual sensor or by sending orders that are forwarded to the bottom layer. In terms of security, each request must contain an HTTP request parameter called IDENTITY_KEY. This token is unique for each provider or client application.

3.1. Running Sentilo in Virtualized Resources

To maintain simplicity and organization, the Sentilo project is segmented into four distinct folders (i.e., sub-projects), as presented in Figure 2: (i) Sentilo-core, (ii) ELK (Elasticsearch, Logstash, Kibana), (iii) monitoring, and (iv) proxy. Each of these utilizes the same custom virtual network provided by Docker. The sentilo-core Docker contains all configurations required to start a Sentilo instance, including agents, Redis, MongoDB, the Sentilo web catalog, and the platform API. MongoDB stores Sentilo's default settings, and the catalog is a web application used to manage the Platform.

[1]`https://redis.io/docs/latest/`

The ELK Docker contains the open source platform ELK, the acronym for ElasticSearch, LogStash, and Kibana. ElasticSearch[2] is a distributed, multi-tenant full-text search engine with an HTTP web interface and schema-free JSON documents and is a highly recommended option for integration into the platform. LogStash is a data collection engine with real-time pipelining capabilities that can dynamically merge data from different sources. It is typically used for log management, but can also serve as an ETL (Extract, Transform, Load) tool. Logstash is connected to Redis Pub/Sub to perform the necessary data transformations before sending the data to an Elasticsearch instance. Kibana, in turn, provides a dashboard software for data visualization for Elasticsearch.

4. Evaluation Approach

In this section, we detail the application load estimation, the tools used and the evaluation approach.

4.1. Application load estimation

Four applications are currently being developed in the city of Toledo, which are described next.

- **Air Quality** [Roncaglio et al. 2023]: a station designed with low-cost sensors to measure atmospheric parameters.
- **Odour**: a device capable of identifying and measure the presence of gases normally found in bad odours [Droprinchinski et al. 2023].
- **Water Quality** [Xavier et al. 2022]: an IoT device equipped with low-cost sensors to monitor water quality parameters.
- **Vehicle Tracking** [Camargo et al. 2021, de Camargo et al. 2023]: the tracker is a device attached to the vehicle that transmits the latitude, longitude, altitude and speed determined by a GPS module.

We want to estimate the required throughput and then check whether the Sentilo platform can handle the amount of requests generated. This study will also tell us about the hardware requirements needed to host the platform.

To estimate the number of air quality stations in Toledo, we assumed the same proportion of stations as predicted for Barcelona, as presented in the work of Sinaeepourfard et al. [Sinaeepourfard and outros 2016], which predicted 40,000 stations. Toledo has about 8.8% of the population of Barcelona, so about 3,857 stations would be needed. The same number was assumed for the stations responsible for detecting bad odours. If the data is recorded every minute, this results in 58.44 requests per second (rps) for both air quality and odour monitoring..

Toledo has four rivers and four lakes in its urban area. Considering the hydrological characteristics of the region, we assume 500 water quality stations, with 8 rps.

The vehicles are tracked with an electronic device equipped with a GPS and a LoRaWAN transmitter. The devices are currently attached to the collection vehicles for recyclable waste. The geolocation is transmitted every 30 seconds. We also expect to be able to track other public vehicles. For this application, we expect about 17 observations per second.

[2]https://www.elastic.co/guide/en/elasticsearch/reference/current/

Table 1. Estimated number of request per second.

Type	Obs/Minute	Devices	Req/Sec
Air Quality	1	3,506	58.44
Odour	1	3,506	58.44
Water Quality	1	500	8.33
Vehicle Tracker	2	500	16.67
Future Applications	1	10,000	166.67
Total	**6**	**18,012**	**308.54**

To make room for future applications, we consider future applications (parking, security, etc.) that could require about 167 rps. In Table 1 we summarize the estimated total number of requests per second. According to our estimates, the platform should be able to process a total of 308.54 rps.

4.2. Tools and Configurations

The Apache JMeter[3] tool was used to generate and send requests to the platform. Apache JMeter enables performance testing and the collection of metrics such as throughput and error rates and latency results.

The test configuration utilized 30 threads for each request, with a maximum response time of 30 seconds. Each thread represents an active user. The "ramp-up" time, which is the duration to reach the maximum number of active threads, was set to 20 seconds. Each test runs for 1 minute. The client and server are connected via the Internet using a VPN network. The Sentilo Platform is installed and configured using containers on a server located at UTFPR university in Toledo.

On the server side, several configurations were adjusted for performance optimization. The size of the Redis "pool" was increased from 10 to 1,000 clients, and the Sentilo API pool queue size was expanded from 400 to 20,000. Additionally, the number of thread pool cores was increased from 4 to 16, and the maximum thread pool size was raised from 10 to 100. Other potential Redis platform and database configurations were not tested.

4.3. Evaluation Approach and Test Cases

The evaluation was conducted through a comprehensive load test across three different platform usage scenarios, six different request loads per second, utilizing five different server hardware configurations. Each combination was tested with three repetitions.

Regarding the usage scenarios, we simulate three functionalities: (i) sentilo-input, (ii) sentilo-output, and (iii) elastic-output. The sentilo-input refers to the platform feeding test, simulating the sending of data from an IoT air quality monitoring device containing 22 different observations, such as the sensor ID and the numerical value corresponding to the measurement of the respective sensor. This uses HTTP requests with the mentioned data, containing a JSON body and a header with a token identified as "IDENTITY_KEY" extracted from the web interface of the platform in the providers and applications config-

[3]`https://jmeter.apache.org/`

uration section. The approximate average size of each request sent is 5 kilobytes, and the approximate average size of the response is 157 bytes.

The scenario called “sentilo-output” executes requests to the Sentilo platform. In this scenario, data from the last 20 observations fed into the platform is queried. As stated in Section 3, the data collected on the Sentilo platform is temporarily stored in the Redis in-memory database. To do this, an HTTP request is made using the “GET” method, along with path and query parameters attached to the end of the URL to provide additional information to the server. The identifier token “IDENTITY_KEY” extracted from the Sentilo web platform, is also included in the request header. Each request is approximately 394 bytes, and each response is about 31 kilobytes.

The third scenario, called “elastic-output”, involves requests to the Elasticsearch database made through the Sentilo platform. Elasticsearch persistently stores the data received and processed by the platform. In this scenario, the last 30 observations are queried. An HTTP request is made using the “GET” method, with a message body in JSON format containing the query in Query DSL, and a token with the “Authorization” identifier in the request header. The approximate average size of each request sent is 454 bytes, and the approximate average size of the response is 1 kilobyte.

The tests were executed in batches: first sentilo-input, followed by elastic-output, and then sentilo-output, adding data to the platform’s databases before making requests via ElasticSearch and Sentilo. Six scenarios were tested with 100, 200, 300, 400, 500, and 1000 rps, each executed three times. After each set, the platform containers were restarted.

4.4. Computational Resources and Metrics

The tests were conducted on virtual machines with five different hardware configurations, as follows:

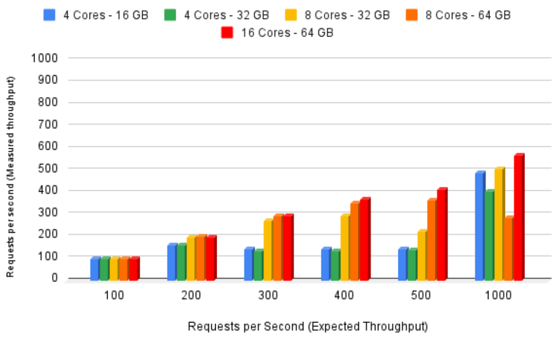
- **c4m16**: 4 CPU cores, 16 GB of RAM.
- **c4m32**: 4 CPU cores, 32 GB of RAM.
- **c8m32**: 8 CPU cores, 32 GB of RAM.
- **c8m64**: 8 CPU cores, 64 GB of RAM.
- **c16m64**: 16 CPU cores, 64 GB of RAM.

The client machine used for tests has an AMD 4700S 3.6GHz octa-core processor, 16GB RAM, and Windows 10 Pro, connected via a fiber optic link with a 28 ms average latency to the server. For the performance evaluation, we collected the following metrics: average response times of requests, failure rates of requests, and the transaction rate per second achieved by the client.

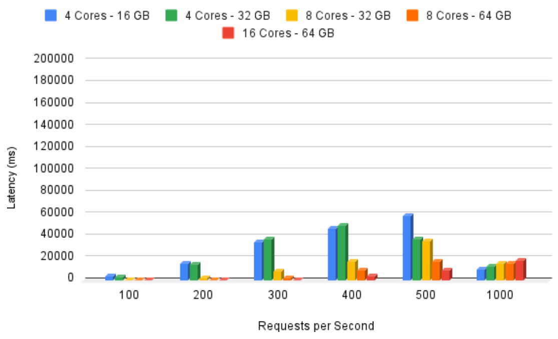
5. Results

The results are divided into two scenarios. The first represents the data insertion and the second the data consumption, both with different hardware configurations and number of requests. The insertion is done via the Sentilo platform and the consumption is done either via the Sentilo platform or directly from ElasticSearch. To compare the results of the individual scenarios, the metrics throughput, latency and error rate were selected. In terms of throughput, *expected throughput* means the number of requests executed in each test, while *measured throughput* is the number of requests measured in each test, i.e., the requests accepted, processed, and answered by the server. *Achieved throughput* is the percentage of achieved requests in relation to the number of expected requests. Error rate is the percentage of failed requests. A high error rate may indicate issues with the system's ability to handle larger data sets or system overloaded.

5.1. Insertion Scenario



(a) Insertion throughput.



(b) Insertion latency.

Figure 3. Insertion throughput and latency.

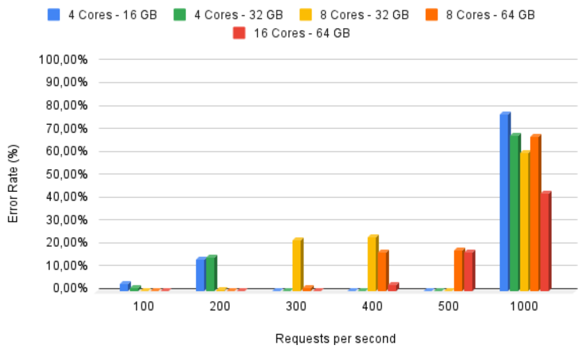


Figure 4. Insertion Error Rate.

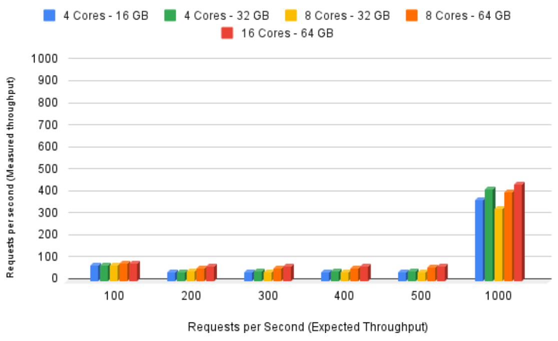
Figure 3 shows the results for throughput (Figure 3(a)) and latency (Figure 3(b)), whereas Figure 4 presents the error rate. At 100 rps, the *achieved throughput* was 99.5%, meaning that the *measured throughput* is very close to the *expected throughput*. However, for the configurations with only 4 cores, the difference between the *expected throughput* and the *measured throughput* increases when having 200 to 500 rps, reaching an *expected throughput* of only 27% in the 500 rps scenario. The latency behaves similarly and is higher when using configurations with 4 cores from 200 to 500 rps. At 500 rps, the highest configuration (c16m64) achieves 415 rps, i.e., 83% of the *achieved throughput* and the lowest latency. These results show that increasing the resources in the hardware configurations improves throughput and latency.

The experiment with 1000 rps resulted in a throughput that is far below expectations, showing that the Sentilo platform is not capable of handling this amount of requests in all available hardware configurations. Nevertheless, the c16m64 configuration reached the highest throughput of 570 rps. However, this result is only 57% of the expected value.

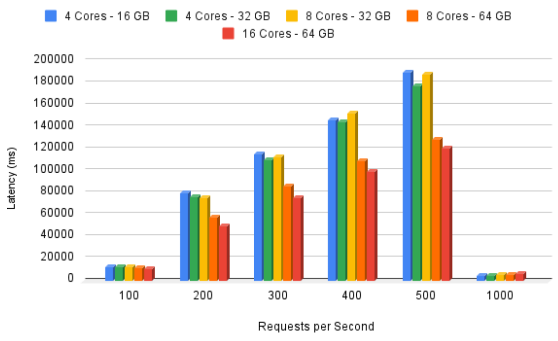
We also noticed that increasing the number of cores increased throughput, but the same did not happen when memory was increased for the same number of cores. We conjecture that the irregular throughput and latency values in this scenario may be a result of the high percentage of error rates, as shown in Figure 4. The error rate is close to 80% in the c4m16 configuration and decreases when hardware resources are added, but is still over 40% in the c16m64 configuration.

5.2. Consumption Scenario

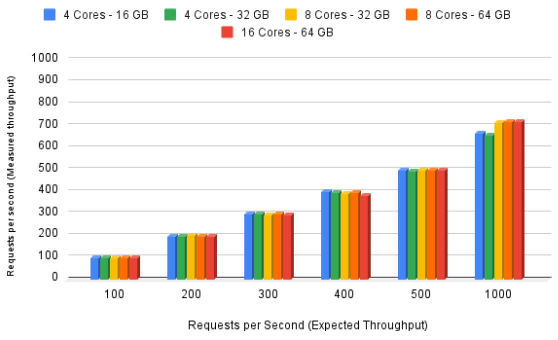
Figure 5 presents the results of the throughput and latency in the consumption scenario. For consumption via Sentilo (Figure 5(a)) and (Figure 5(b)), low throughput and high latency were reached for all hardware configurations between 100 and 500 rps, with values similar to each other. The best result was 80.89 rps on the c16m64 configuration for 100 rps (Figure 5(a)). On the other hand, the consumption tests via the ElasticSearch API (Figure 5(c)) and (Figure 5(d)) show an increase in throughput and low latency as the hardware configurations increase. In the tests from 100 rps to 500 rps via ElasticSearch, the throughput is very close to what is expected (Figure 5(c)).



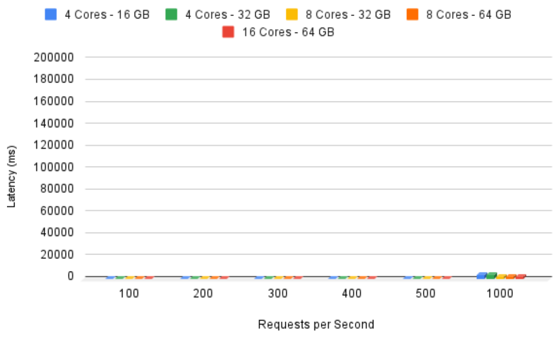
(a) Sentilo Consumption Throughput.



(b) Sentilo Consumption Latency.



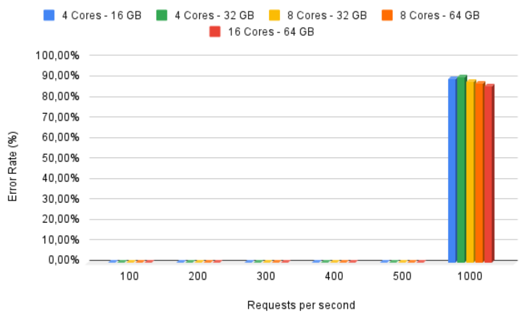
(c) Elasticsearch Consumption Throughput.



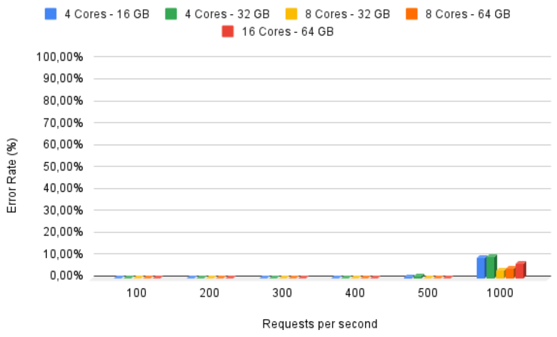
(d) Elasticsearch Consumption Latency.

Figure 5. Consumption Throughput and Latency for Sentilo and ElasticSearch.

Tests between 100 rps and 500 rps in the consumption scenario via Sentilo (Figure 5(a)) and (Figure 5(b)) reveal a certain throughput limit at which an increase in hardware resources has little or no effect on the final result. In the tests with 1000 rps, the throughput results via Sentilo increased significantly with a lower latency, but the results are not linear with the increase in hardware resources.



(a) Sentilo Consumption Error Rate Results.



(b) Elasticsearch Consumption Error Rate Results.

Figure 6. Consumption Error Rate Results.

In the consumption tests via Elasticsearch, the throughput result for 1000 rps increased compared to the 500 test (Figure 5(c)). The maximum value reached was 718.46 rps for the c8m64 configuration, followed by 718.19 for the c16m64 configuration, i.e., very similar results. The latency for 100 rps follows the throughput result and shows an increase in request time (Figure 5(d)). This demonstrates some kind of overload in the system, reducing the processing and response capacity.

When analyzing the error rate (Figure 6(a)), tests conducted with 100 to 500 rps showed 0% errors across all configurations. The same behavior was observed in the consumption via Elasticsearch (Figure 6(b)), where the average error rate remained below 0.1% up to 400 rps across all configurations, with a significant increase in the tests with 500 rps, reaching a maximum of 1.02%. In the 1000 rps scenario, it was noted that increasing the RAM configuration, without increasing the number of computing cores, led to an increase in the error rate.

5.3. Discussion

The data insertion scenario demonstrated that providing additional hardware resources improved throughput capacity, while simultaneously reducing both latency and error rates, up to the 500 requests per second (rps) scenario. However, beyond this point, further resource increases did not result in proportional performance improvements.

In the experiment with 1000 rps, the throughput observed was significantly lower than expected, indicating that the Sentilo platform is not capable of handling this volume of requests across all tested hardware configurations. The reason is indicated in Figure 7: the sentilo-platform-server container (Sentilo API) reached a CPU utilization peak of approximately 200%, which was accompanied by an error rate of 100%. No other performance bottlenecks were identified in the other containers that make up the solution. System-level metrics, such as network throughput, memory usage, and disk I/O performance, remained within acceptable operational thresholds, and did not exhibit signs of resource saturation that could contribute to performance degradation. These findings suggest that the high error rates observed in all graphs at 1000 rps can be directly attributed to the CPU limitations in request handling capacity of the sentilo-platform-server container.

The experiments conducted at 1000 rps reveal that the platform reaches its performance limits between 500 and 1000 rps, for the hardware configurations tested. The platform successfully supports the number of requests estimated for the applications to be developed, specifically 308 rps, in the two most robust configurations (c8m64 and c16m64). For higher loads, it becomes necessary to scale the system either by enhancing the existing hardware resources, such as CPU and memory, or by implementing horizontal scalability strategies.

In the data consumption scenario, it was found that Sentilo could not efficiently handle requests made directly to the platform. In contrast, consumption via Elasticsearch produced highly satisfactory results across all scenarios. One potential reason for the sub-optimal performance when consuming data directly through the platform could lie in its architecture. The platform's publish-subscribe model facilitates integration with external tools such as storage solutions and databases (Section 3). In this context, configuring an agent to send received data to third parties, immediately upon reception in Redis, would likely provide an optimal solution. Notably, an agent is already in place to forward data from Redis to Elasticsearch.

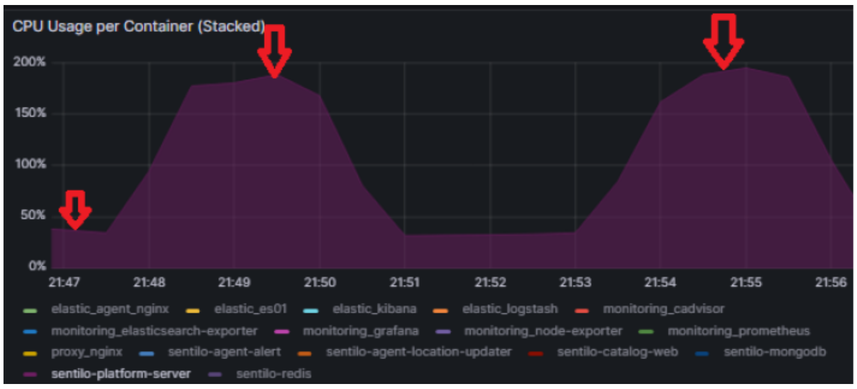


Figure 7. CPU sentilo-platform-server.

6. Conclusion

This work evaluates the scalability of the Sentilo platform to determine whether it is capable of guaranteeing the execution of smart city applications planned for a medium-sized city. The configuration of Sentilo using containers facilitated rapid deployment on different computer configurations. The tests were carried out using the JMeter tool. The results show that the platform is capable of handling the estimated number of requests for the applications to be developed, and that it is capable of scaling vertically as hardware resources increase. Future work includes evaluating CPU and memory consumption during test execution for all scenarios and configurations, conducting a study on the platform's ability to scale horizontally by replicating containers in a clustering strategy, and evaluating how the platform behaves when the number of agents increases.

References

Araujo, V., Mitra, K., Saguna, S., and Åhlund, C. (2019). Performance evaluation of fiware: A cloud-based iot platform for smart cities. *JPDC*, 132:250–261.

Camargo, E. T., Spanhol, F. A., and Castro e Souza, A. R. (2021). Deployment of a lorawan network and evaluation of tracking devices in the context of smart cities. *JISA*, 12(8):1–24.

de Camargo, E. T. et al. (2023). Low-cost water quality sensors for iot: A systematic review. *Sensors*, 23(9).

de M. Del Esposte et al. (2019). Design and evaluation of a scalable smart city software platform with large-scale simulations. *Future Generation Computer Systems*, 93:427–441.

Droprinchinski, L., Tavares de Camargo, E., and de Morais, M. V. B. (2023). Low-cost sensors for odor monitoring: the state of the art and challenges. In *2023 IEEE International Smart Cities Conference (ISC2)*, pages 01–04.

Goumopoulos, C. (2024). Smart city middleware: A survey and a conceptual framework. *IEEE Access*, 12:4015–4047.

Ismail, A. A., Hamza, H. S., and Kotb, A. M. (2018). Performance evaluation of open source iot platforms. In *2018 IEEE global conference on internet of things (GCIoT)*, pages 1–5. IEEE.

Lai, C. S., Jia, Y., Dong, Z., Wang, D., Tao, Y., Lai, Q. H., Wong, R. T. K., Zobaa, A. F., Wu, R., and Lai, L. L. (2020). A review of technical standards for smart cities. *Clean Technologies*, 2(3):290–310.

Pereira, J., Batista, T., Cavalcante, E., Souza, A., Lopes, F., and Cacho, N. (2022). A platform for integrating heterogeneous data and developing smart city applications. *Future Generation Computer Systems*, 128:552–566.

Roncaglio, M. M. et al. (2023). Development of an air quality station using low-cost sensors. In *2023 XIII Brazilian Symposium on Computing Systems Engineering (SBESC)*, pages 1–6.

Santana, E. F. Z. et al. (2017). Software platforms for smart cities: Concepts, requirements, challenges, and a unified reference architecture. *ACM Comput. Surv.*, 50(6).

Sentilo (2014). Sentilo - sensor and actuator platform for smart cities. `https://joinup.ec.europa.eu/collection/egovernment/document/sentilo-sensor-and-actuator-platform-smart-cities`. Accessed on: June 18th. 2024.

Sinaeepourfard, A. and outros (2016). Estimating smart city sensors data generation. In *2016 Mediterranean Ad Hoc Networking Workshop (Med-Hoc-Net)*, pages 1–8.

Sánchez-Corcuera, R. et al. (2019). Smart cities survey: Technologies, application domains and challenges for the cities of the future. *Int. JDSN*, 15(6).

Tabassum, M., Puryear, N., Kuzlu, M., Jovanovic, V., and Abdelwahed, S. (2023). Performance evaluation of a cloud-based iot platform for smart cities: Opencybercity. In *2023 12th Mediterranean Conference on Embedded Computing (MECO)*. IEEE.

Xavier, F. et al. (2022). Evaluation of low-cost sensors for real-time water quality monitoring. In *Anais Estendidos do XII Simpósio Brasileiro de Engenharia de Sistemas Computacionais*, pages 56–61. SBC.