

Evaluation of the data handling pipeline of the ASTRID framework

Guerino Lamanna, Matteo Repetto, and Alessandro Carrega

Abstract—Effective attack detection and security analytics rely on the availability of timely and fine-grained information about the evolving context of the protected environment. The data handling process entails collection from heterogeneous sources, local aggregation and transformation operations before transmission, and finally collection and delivery to multiple processing engines for analysis and correlation.

Many SIEM tools work according to the “funnel” principle: gather as much data as possible and then filter it to keep the relevant information. However, this might lead to unacceptable overhead, especially when monitoring containerized environments. As part of our activity in ASTRID, we therefore conducted experimental investigation on resource consumption of the data handling pipeline, starting from embedded agents up to delivery to the Context Broker.

Index Terms—Elastic stack, containers, monitoring, Kafka

I. INTRODUCTION

Effective and reliable detection of known and zero-day attacks heavily relies on the knowledge of the relevant context, which includes data, events, and measurements from the execution environment. The more information is available, the more the identification of attack patterns is likelihood. As a matter of fact, many SIEM tools work according to the “funnel” principle: gather as much data as possible and then filter it to keep only the relevant information. When the monitoring infrastructure is quite rigid and static, this is a safer approach from the detection perspective: if some data are not useful at a given moment, they may become essential in the future, to detect a different type of attack or a variant.

Since there are virtually no limitations to the range of monitoring and inspection aspects that may be considered, the computational and communication overhead become easily unpractical for any realistic scenario, especially for lightweight containerized applications. Therefore the design of cyber-security architectures has often been driven by the need to balance the accuracy of the retrieved security context with the overhead, which also affects the placement of analysis and detection tasks within the system.

Chasing the objective of better efficiency and improved awareness, ASTRID goes beyond static and rigid architectures, by introducing more flexibility and dynamicity in the detection process [1]. The specific objective is the ability to create a capillary and pervasive context fabric for security functions and data, which can be configured at run-time and hence

can continuously adapt the depth of inspection to the current detection needs. Based on this objective, we investigated the overhead introduced by the data handling pipeline, including both local agents, and delivery components in the Context Broker (CB). Our purpose is to provide a useful guide to their selection and usage at run time.

This document describes the main results and findings from our experimental analysis of the data handling pipeline; it extends the preliminary work in D2.1 [1] by moving from virtual machines to containers and by including Kafka. Our analysis mainly considered both performance and resource usage. In the first case, we considered latency introduced when collecting and delivering data, whereas the second aspect was evaluated in terms of CPU consumption and memory allocation. The scope is limited to components adopted from other frameworks, because those developed within the project are already discussed in separate documents [2], [3].

The rest of the paper is organized as follows. Section II briefly remind the data handling pipeline implemented in ASTRID, excerpted from D2.6 [4]. Section III describes the experimental testbed used for all the following experiments. Sections IV and V investigate the usage of Filebeat and Metricbeat for retrieving applications logs and metrics, respectively. Section VI reports measurements from the delivery bus, namely Kafka. Finally, we give our conclusion in Section VII.

II. ASTRID DATA HANDLING PIPELINE

ASTRID builds on the Elastic Stack framework, which consists of standalone components working harmoniously to realize real-time data streaming. It covers several data handling processes, from collection, to processing and storage, up to visualization; it ensures security, scalability, and flexibility of critical real-world applications.

As shown in Figure 4, data acquisition is managed by Beats¹, which collect heterogeneous data in a lightweight manner and can be deployed as local agents. The collection of logs (i.e., Filebeat) and metrics (i.e., Metricbeat) are the main functionalities utilized by ASTRID. Beats report and ship their collected data to a local Logstash instance for data transformation and enhancement before delivering it to the remote components. Logstash also can take inputs from external sources, managing heterogeneous data inputs. It aggregates, filters, and timestamps data from ASTRID-specific agents [3] before pushing it to Kafka.

M. Repetto is with CNR – Institute for Applied Mathematics and Information Technologies (IMATI).

A. Carrega is with CNIT – S2N National Lab.

G. Lamanna is with Infocom Srl

¹Getting started with Beats, [Online]. Available: <https://www.elastic.co/guide/en/beats/libbeat/current/getting-started.html>.

Kafka [5] is used to stream data into dedicated communication channels intended for different recipients. As shown in the figure, data is sent to the Kafka bus and received by both the centralized Logstash and security services that process real-time information. Elasticsearch further indexes the data for easier retrieval, which is utilized for data visualization or offline analytics. Kibana allows data visualization and navigation from the stack, and it is integrated into the ASTRID dashboard.

The architecture supports both online streaming and offline analytics, but only the streaming pattern has been implemented and used in ASTRID. From an architectural perspective, the most critical issues for a virtualized application can be summarized as follows:

- Monitoring and aggregation agents must run as close as possible to the virtual function. Though it is possible to leverage specific capabilities in the infrastructure to collect data, ASTRID follows an infrastructure-agnostic approach and therefore places such agents in the same virtual function. It is therefore fundamental that monitoring and inspection operation does not affect the execution of the main business logic.
- The collection bus represents a bottleneck for the whole data handling pipeline. We do not discuss the practical problems that come from Internet links, since this is an aspect that must be taken into account by the user of the ASTRID platform². However, the Kafka bus may impact the collection of large bulk of information in the centralized platform, hence it is important to be aware of any performance issues before the concrete deployment.

In the remaining of this paper we address the above issues by investigating performance and scalability in our project experimental testbed.

III. TESTED DESCRIPTION

We set up an experimental testbed for validation and identification of performance gaps. We did not use applications from our Use Cases, because in that case it is more difficult to simulate different working conditions; rather we deployed common services that are expected to be present and that allow to use the beats under investigation.

We considered two different services: an Apache web server (monitored by Filebeat) and a MySQL database server (monitored by Metricbeat). Both the main service and the corresponding agent are standalone containers that run in the same pod, together with an instance of Logstash and the LCP. However, the latter is not considered in this evaluation, because it is not involved in the data processing chain. Its performance related to control and management operations are subject of a parallel study [7]. We also deployed an instance of the Context Broker in a separate pod, that collects all data.

All pods are deployed in a local testbed, made of 3 Kubernetes nodes equipped with 2x Intel Xeon CPU E5-2660 v4 @ 2.00GHz with 14 cores and hyperthreading enabled, 128 GB RAM, 64 GB SSD storage. The local connection is a

plain 1 Gbps Ethernet. We used the default configuration for all containers (1 vCPU, 250 MB RAM).

The evaluation investigated how resource consumption varies while changing the offered load and frequency of sampling. Specifically, the evaluation was conducted by varying the following parameters:

- the *period of collection*, which affects the latency to access the context and in some cases the volume of traffic generated over the network (for Metricbeat, because it reports the current status at each request), from 1 to 20 seconds;
- the *rate of requests* to the Apache and MySQL servers, which in some cases increase the volume of logs generated (Apache function, which records every access), from 1 to 1000 requests/s. We used jmeter³ and mysqlslap⁴ to generate a variable amount of requests for Apache and MySQL, respectively.

Additionally, we investigated how Kafka performs with a variable number of messages, consumers, and producers.

IV. FILEBEAT

In case of Filebeat, the agent periodically scans the logs generated by Apache and checks for new records to be sent to the CB. In our testbed, Logstash adds a timestamp to each log records, and this implies more processing in case of larger workload.

A. CPU usage

Fig. 1 shows the average and standard deviation for CPU usage⁵ of Apache, Filebeat, the local Logstash instance (named Logstashbeat in the pictures) and the centralized Logstash instance in the Context Broker. With a low rate of requests, the agents uses about the same amount of CPU as the server that they are monitoring; however, when the number of requests increases, the overhead of the agents remains limited. The slight increase is due to the large volume of logs to be processed and transferred. Indeed, for Apache, a larger number of requests implies more logs, hence more CPU is used to read the data and send them to the Context Broker.

The Logstash instance in the CB raises CPU usage too, but this is not a problem, because this runs in the ASTRID platform, hence it does not impact the execution of virtual services. Indeed, the two instances in our simple testbed made the same thing (namely, they add a timestamp to the message), but they use different input/output plugins. The local instance reads data from a beat and writes to Kafka; the CB instance reads from Kafka and writes to Elasticsearch. Probably, the management of such plugins is not as efficient as the Filebeat implementation. We conclude that the impact of agents remains quite limited (below 10%), which is acceptable in most practical cases. By comparing Filebeat and Logstash, we conclude that the latter has a higher overhead, even if its operation is simpler.

³<https://jmeter.apache.org/>.

⁴<https://mariadb.com/kb/en/mysqlslap/>.

⁵CPU usage is computed as the fraction of time the container runs, according to measurements reported by Kubernetes.

²A discussion on the impact of the Internet is given as appendix to D1.3 [6], mostly from the security perspective.

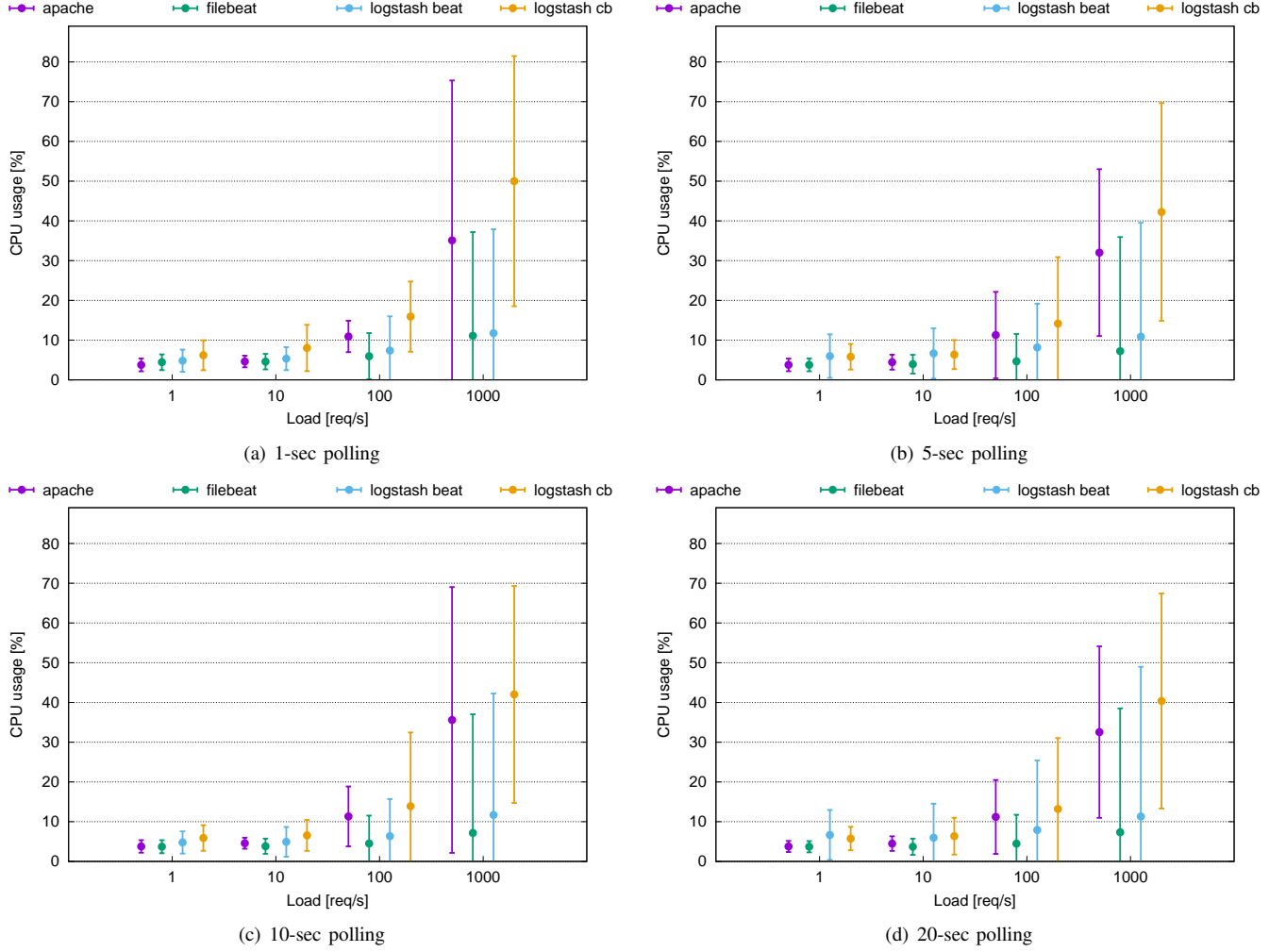


Fig. 1. Average and standard deviation for CPU usage under different load and polling conditions.

Finally, we compare the average CPU usage for all containers in the same pod (namely, Apache, Filebeat, Logstash) in Figure 2. We see that the overhead of the agents is rather limited (below 10% of the available CPU) in all conditions but for the largest number of requests.

B. Memory allocation

Memory allocation is rather constants under the different conditions for Apache and Filebeat, but largely increases for Logstash, both local and remote instances. Fig. 3 shows that beyond the greater average value, it is also quite unstable during the experiments. Our understanding is that the current implementation of Logstash is not suitable for lightweight operation in cloud-native applications, because its memory footprint is often bigger than the main application. In general, it would be preferable to directly write to the Kafka bus with Filebeat, if additional transformation operations are not strictly necessary.

Also in this case we provide the cumulative memory allocation for the pod (Figure 4), which better highlights the impact of Logstash.

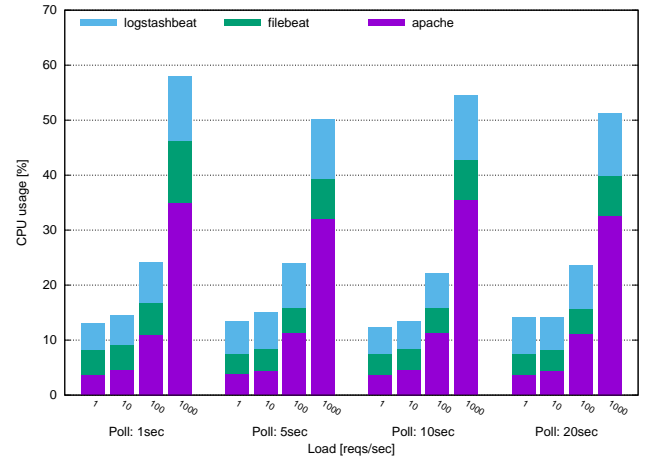


Fig. 2. Cumulative CPU usage by all containers in the same pod.

C. Latency

The last parameter that we considered is the processing latency introduced by the Elastic chain. Figure 5 shows the latency to gather data from the Apache log file by Filebeat,

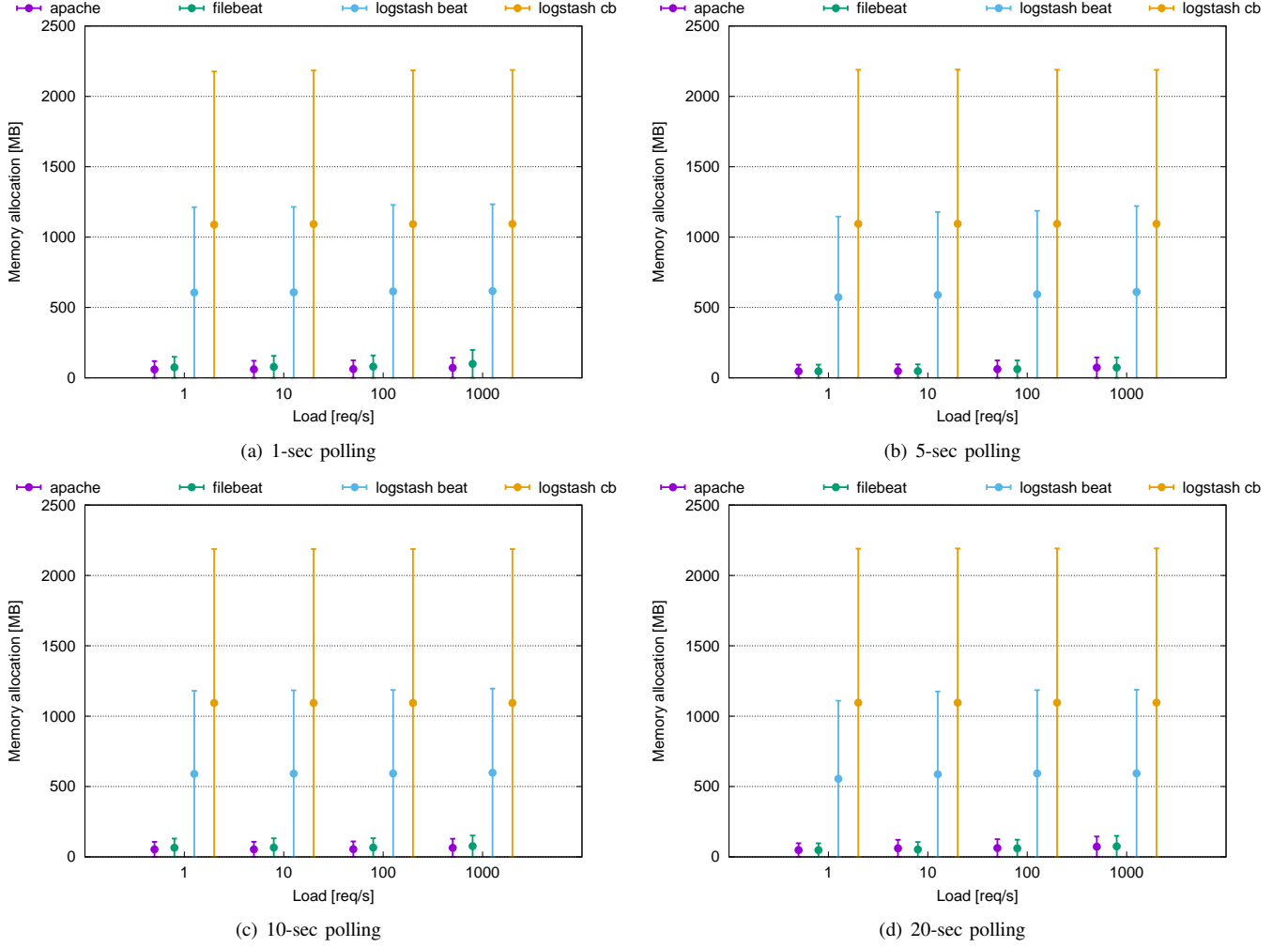


Fig. 3. Average and standard deviation for memory allocation under different load and polling conditions.

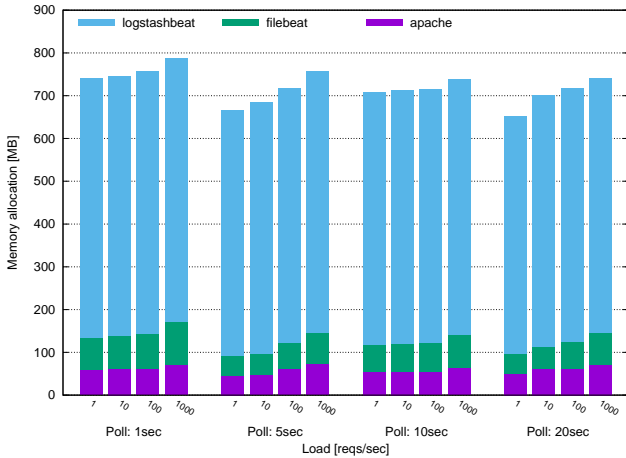


Fig. 4. Cumulative memory allocation for the pod.

to move data from the Filebeat to the local Logstash instance, and to transfer data to the CB. The latency is generally shorter than a few seconds, but it exhibits a small increase with the workload. As a matter of fact, the more the records in the log file, the bigger the message(s) to be sent to the CB. The

handover between Filebeat and Logstash is the component of the overall delay which is more sensitive to the increment of the number of records.

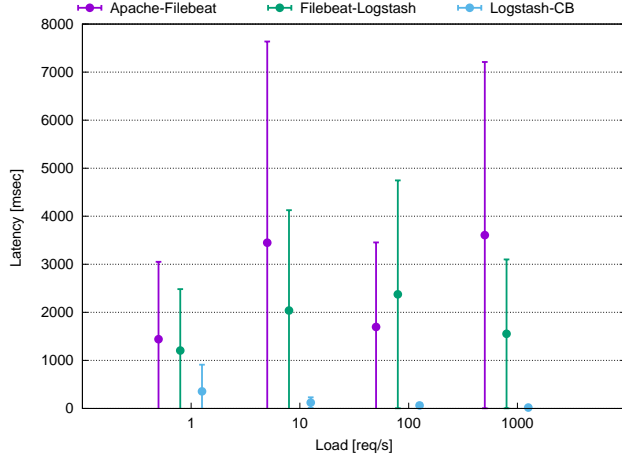
The polling interval has a larger impact on the overall latency. This is because Apache logs are read less frequently when this interval increases, hence the latency from the generation of the record to its availability in the centralized platform increases. Fig. 5 shows that the collection of records from the Apache logs is largely affected by this parameter, due to the need of concurrent writes/reads to the same file by different processes.

A greater latency in data collection is not a specific performance limitation of the agents, but it may have side effects on the timely of the detection. The polling interval must therefore be selected case-by-case depending on the specific needs of the detection process.

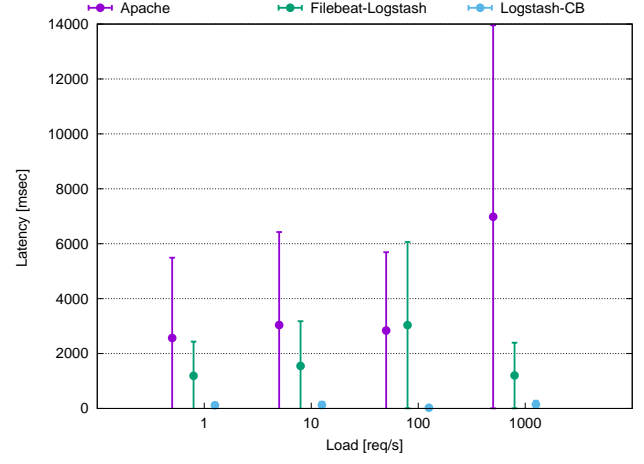
We finally provide an intuitive comparison of the average latency in each pod in Figure 6.

V. METRICBEAT

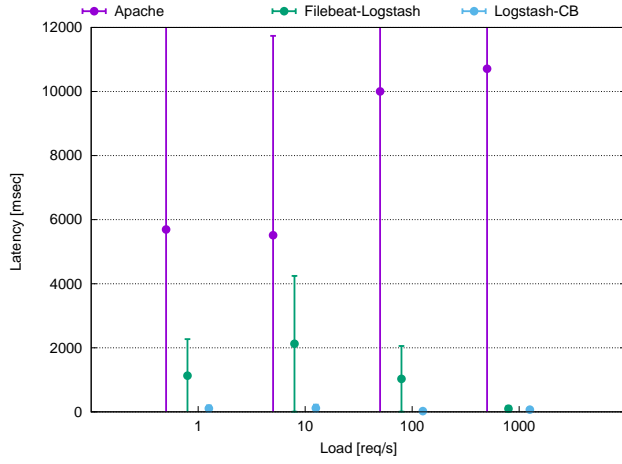
The couple MySQL/Metricbeat works rather different than the previous case. As a matter of fact, Filebeat periodically collect logs, hence the amount of information is directly



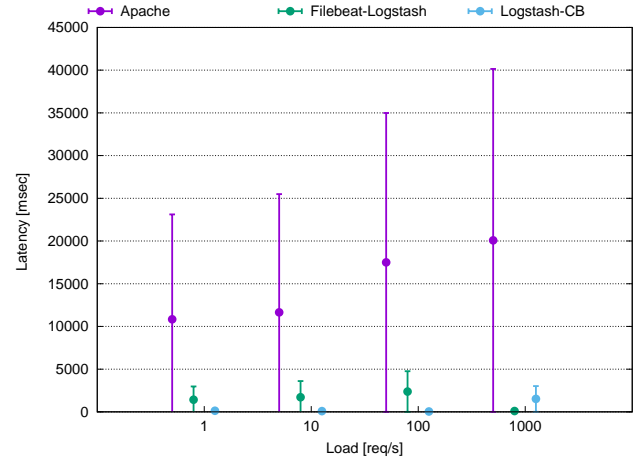
(a) 1-sec polling



(b) 5-sec polling



(c) 10-sec polling



(d) 20-sec polling

Fig. 5. Latency measured under different load and polling conditions.

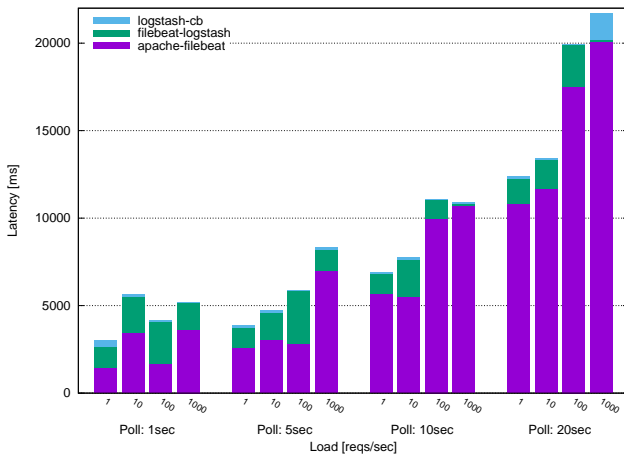


Fig. 6. Cumulative latency for the pod.

proportional to the number of records, which in turns come from the requests. Instead, Metricbeat periodically queries MySQL for given metrics, and the answer is always the same size, independently from the current load. We expect this to have an impact on the overall data handling pipeline.

A. CPU usage

Fig. 7 shows the average and standard deviation for CPU usage of MySQL, Metricbeat, the local Logstash instance (named Logstashbeat in the pictures) and the centralized Logstash instance in the Context Broker. With a low rate of requests (below 10 requests per second), CPU used by the agents is comparable with the server they are monitoring; however, when the number of requests increases, the overhead of the agents remains limited. Differently from Apache, in this case the CPU usage of MySQL becomes very high, even beyond 100%. In our understanding, the high load put on the server affects the precision of the measures, which anyway should be intended as “very close to 100%.”⁶

Another meaningful difference with the Apache use case is the CPU usage of Logstash. This is likely due to the fact that a smaller and constant amount of information is reported in this case. We conclude that the impact of agents remains quite limited (below 10%), which is acceptable in most practical cases.

⁶CPU usage is computed in the same way as for Apache, namely by using the number of milliseconds the container ran. The percentage is then calculated on the assumption that these measures are reported exactly every seconds by Kubernetes, but likely this process is not precise under high-load conditions.

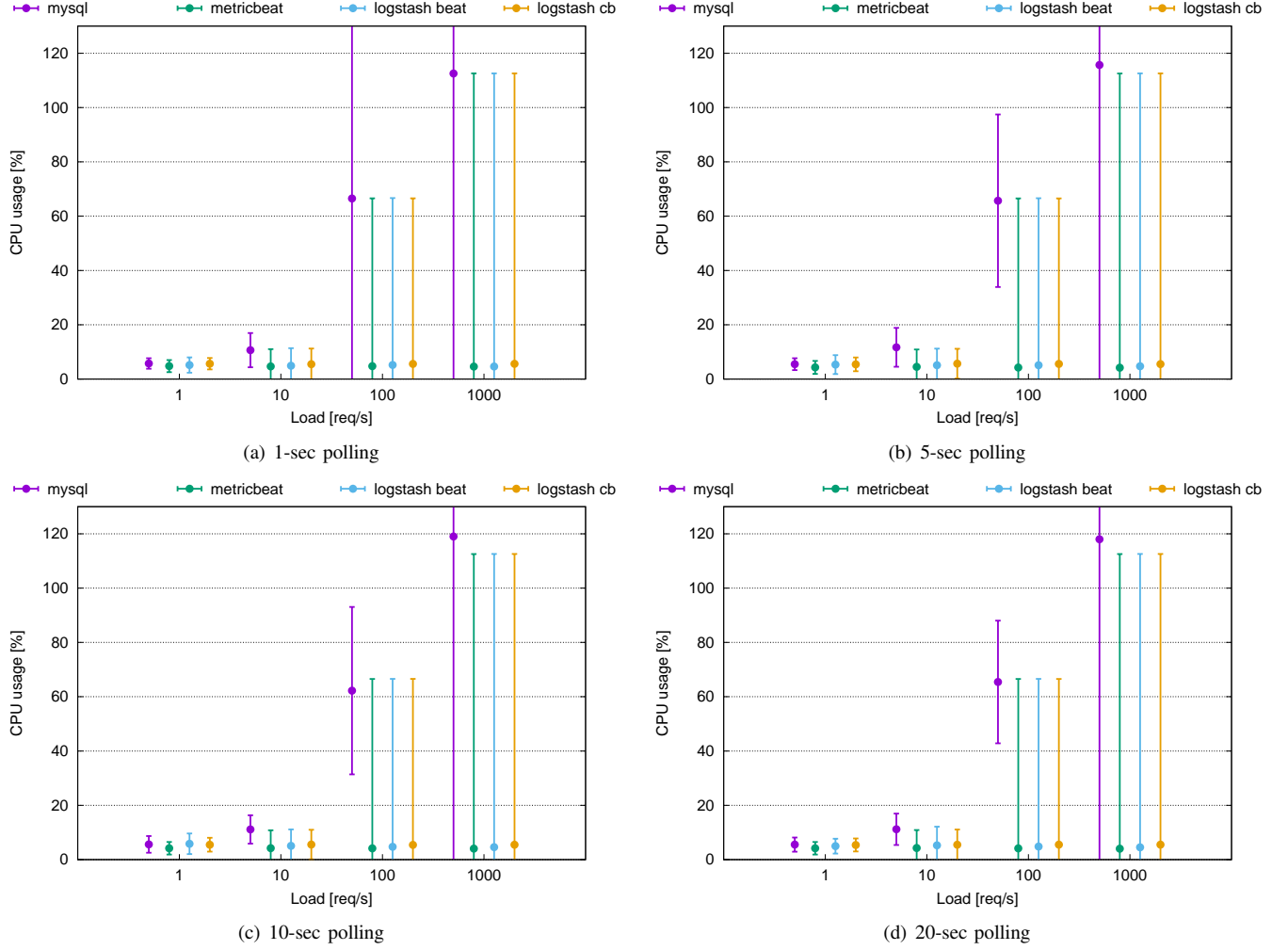


Fig. 7. Average and standard deviation for CPU usage under different load and polling conditions.

Finally, we compare the average CPU usage for all containers in the same pod (namely, MySQL, Metricbeat, Logstash) in Figure 7. We see that the overhead of the agents is rather limited (below 10% of the available CPU) in all conditions, similar to the Apache use case, but this time the relative impact on MySQL is much lower, especially at high-load.

B. Memory allocation

Memory allocation is rather constants under the different conditions for our agents, and slightly increases for MySQL at higher loads. Fig. 9 shows that memory consumption for MySQL is around one order of magnitude greater than Apache (even if this may change in more complex scenarios), whereas we got almost the same value for the agents in the two use cases. Hence, the main findings about the issues in running Logstash in virtualized functions also hold here.

Also in this case we provide the cumulative memory allocation for the pod (Fig. 10). In this case, given the larger memory allocation to MySQL, the impact of Logstash looks more limited than with Apache, but still comparable with resource usage of the main business function.

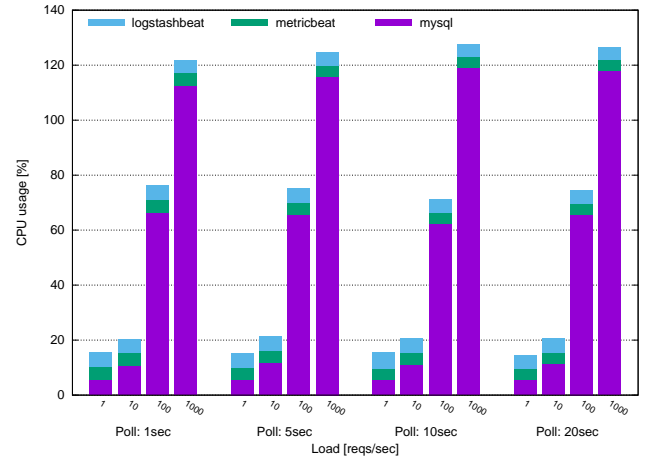


Fig. 8. Cumulative CPU usage by all containers in the same pod.

C. Latency

For what concerns the latency, estimation of the latency introduced by Metricbeat for collecting data was not possible this time. Hence, Fig. 11 only shows the latency to move data

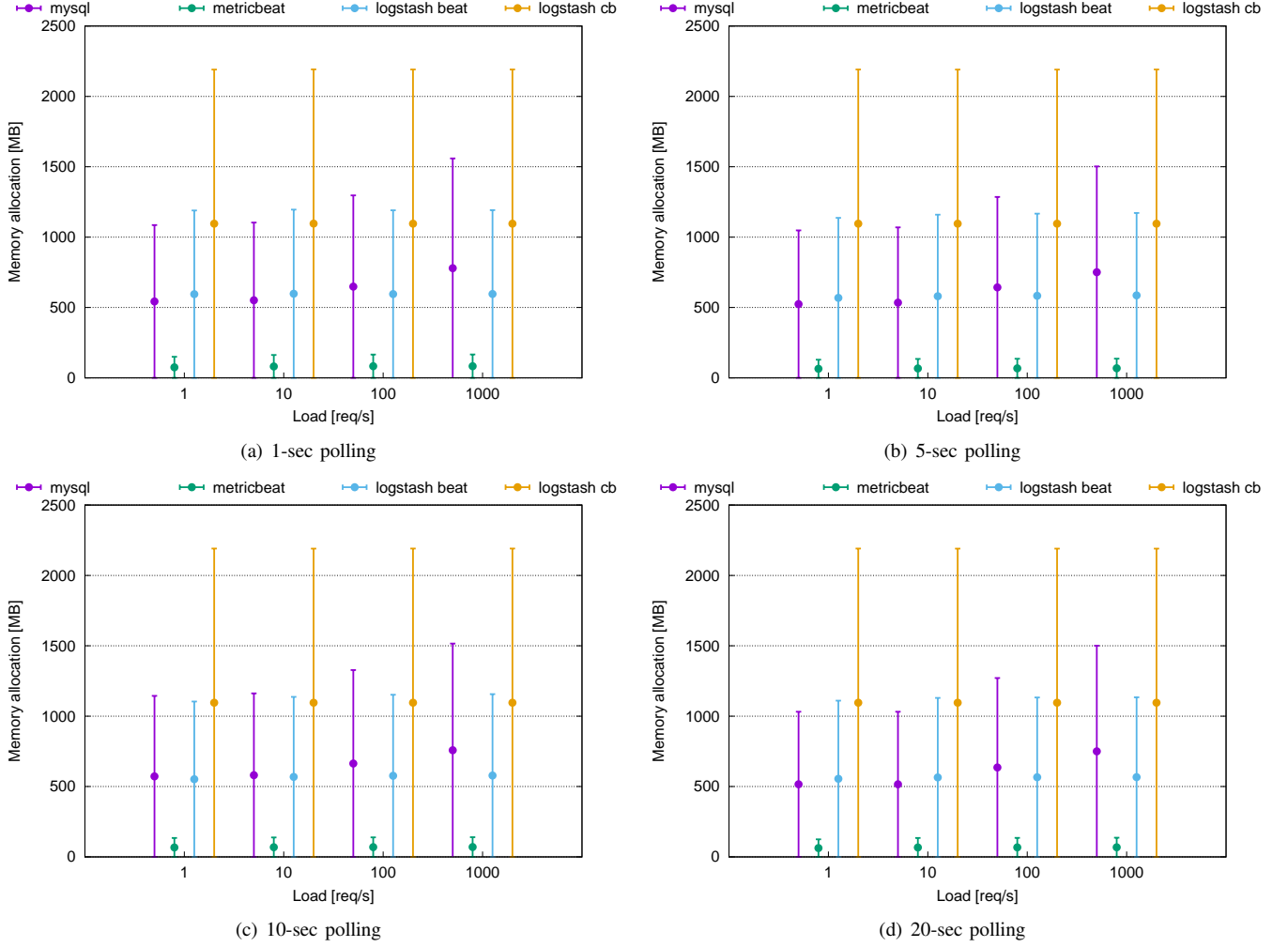


Fig. 9. Average and standard deviation for memory allocation under different load and polling conditions.

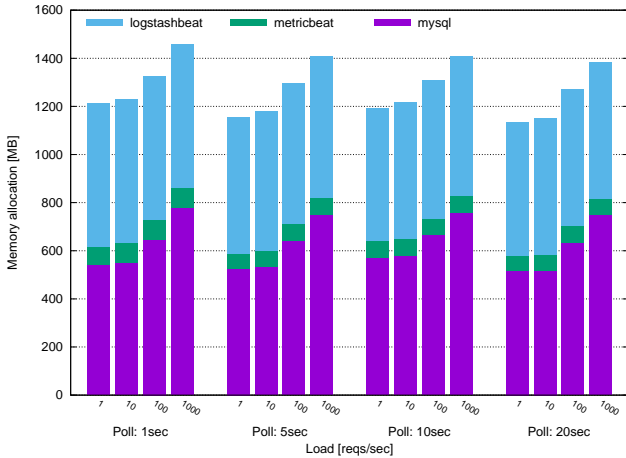


Fig. 10. Cumulative memory allocation for the pod.

from Metricbeat to the local Logstash instance and to transfer data to the CB. The latency is generally around one second, and this time does not increase with the workload or the polling interval. Again, the reason is because the same amount of information is collected, independently of the workload.

This is more or less the same behaviour of other agents used in ASTRID, especially those related to network monitoring (with the notable exception of flow collection).

The lower latency for the smaller polling interval (1 second) appears an anomaly with respect to other data. The lower value is due to the fact that sometimes Metricbeat packs two samples and sends them together; the first sample experiences the usual delay of around 1 second, but the second is available immediately afterwards, and has a latency of only 100 milliseconds. Eventually this lowers the total average of all samples. The same effect was not observed for other polling intervals, where samples are available with a quite constant delay of 1.1 s. We do not know in detail both the lumberjack protocol between Metricbeat and Logstash and the internal implementation of Metricbeat; however, it seems that using the same polling interval as the delivery delay creates some sort of de-synchronization in the metric forwarding operations.

Similar to the previous case, the impact on transferring data through the message bus is rather limited with respect to the handover internal to the pod. This is more clear from the comparison of the average latency in each pod, shown in Fig. 12.

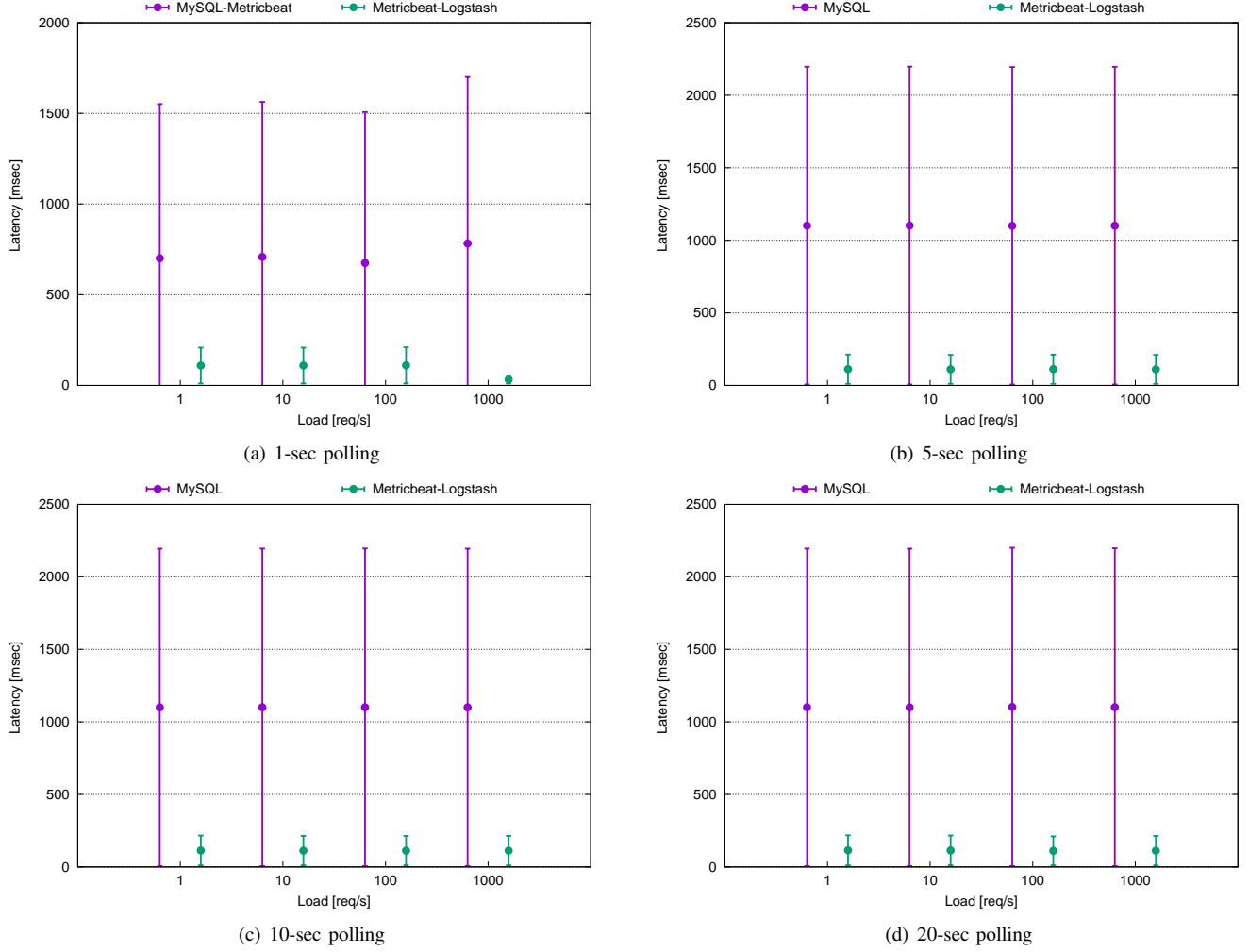


Fig. 11. Latency measured under different load and polling conditions.

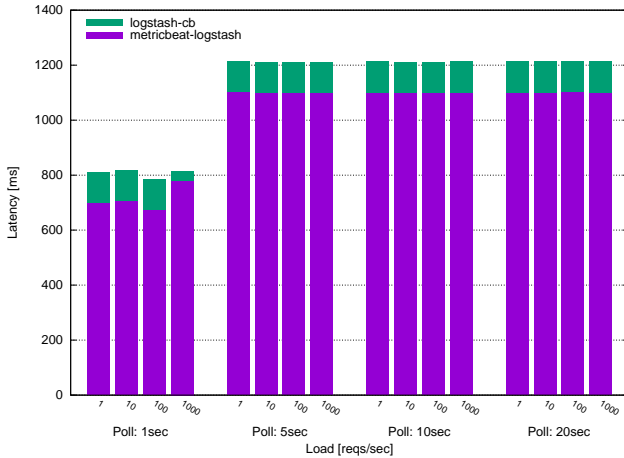


Fig. 12. Cumulative latency for the pod.

VI. KAFKA

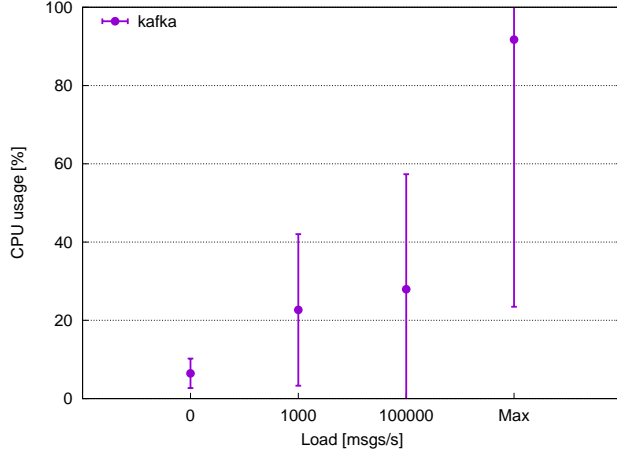
Kafka is part of the CB pod and it is a potential bottleneck because all messages from agents goes through this bus. Indeed, also notifications from the detection algorithms share

the same Kafka bus, but their rate is expected to be negligible with respect to traffic generated from agents.

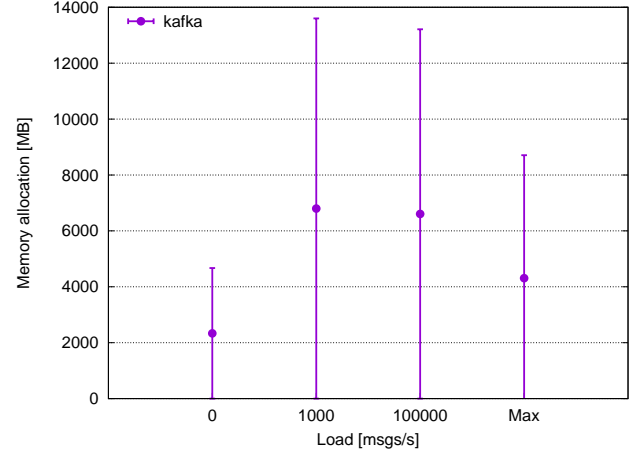
We measured both CPU usage and memory allocation when no messages are transmitted, with a rate of 1,000 and 1,000,000 messages per seconds and at the maximum speed achievable by the generator (with packet size of 100 bytes). To this aim, we used the tools available from Kafka (kafka-*-perf-test suite⁷). A single producer/consumer was used in this case. A single broker was used in our experiments, even if Zookeeper would allow to scale to more instances.

Figure 13 shows how CPU usages increases with the number of messages. We do not expect more than 100 agents to be realistically available in the use-case of interest for the project. In this worst case, therefore, up to 1,000 messages per second from each agent will not use the CPU for more than 30% of its time. We think this is more than acceptable, also taking into account the measurements for Logstash in the previous Sections and considering that the ASTRID platform would likely run on dedicated hosts with at least 4-8 cores available (it could be either a virtual machine or a physical server).

⁷<https://docs.cloudera.com/runtime/7.2.10/kafka-managing/topics/kafka-manage-cli-perf-test.html>.



(a) CPU usage



(b) Memory allocation

Fig. 13. CPU usage and memory allocation for the Kafka container.

TABLE I
PERFORMANCE FOR DIFFERENT CONFIGURATIONS OF PARTITIONS, CONSUMERS AND PRODUCERS.

Partitions per topic	Consumers/Producers pairs	Msg size [byte]	Throughput		Latency	
			[Msgs/s]	[MB/s]	Avg [ms]	Max [ms]
1	1	100	141643	13.51	45.2	407
		1000	42517	40.55	519.15	654
2	2	100	141442	13.49	8.98	411
		1000	62814	59.9	288.54	395
3	3	100	149031	14.21	15.82	417
		1000	71736	68.41	241.26	414

With regard to memory allocation, only a few megabytes of RAM are necessary, even at the highest rate, which makes this solution perfectly scalable. We note that, quite oddly, memory allocation decreases for higher rates. In our understanding, this is due to the congestion of the bus (see the high level of CPU usage), which blocks the reception of packets.

We then performed a stress test on Kafka when varying the number of consumers and producers, also taking into account packets of different size. In this case, messages are always generated at the maximum rate achievable with the generator, which yields to different performances in terms of number of messages and bytes exchanged.

Table I shows that better efficiency is possible by transmitting bigger packets, namely the data transfer rate is higher. Unfortunately, this is not always possible. Data aggregation should be performed at the producer side, for instance increasing the polling interval, but this might introduce unacceptable delays in case of rare data. We already discussed the impact of larger polling intervals in Sec. IV.

As a general consideration, it seems that increasing the partition number has a positive effect on data transfer when multiple producers and consumers are present.

If, on the one hand, bigger packets improve data transmission rate, on the other side they lead to much higher delivery delays. The difference between the average delay between 100-bytes and 1,000-bytes packets is more than an order of magnitude. Overall, the differences in the maximum delays

are not so large, and are comparable in almost all scenarios.

VII. CONCLUSION

Our analysis largely confirmed the preliminary results we got at the beginning of the project. Running standard beats in the same pod as business functions increases resource usage. From the computing perspective, the impact is rather low and only marginally affects the operation of the application. In general, resource consumption is bigger when the amount of information to be collected is proportional to the workload (e.g., application/system logs).

Memory allocation for Logstash is not negligible, and requires far more resources than what is needed to run the main function. In this respect, this component should be avoided as much as possible, and more efficient solutions should be found. In this perspective, the usage of eBPF represents a valid alternative for performing simple data processing, even if it is not suitable for more advanced data fusion and transformation operations.

We also demonstrated that a single instance of the Kafka broker is enough for most use-cases covered by the project. As a matter of fact, the target size of virtual services foreseen at the beginning of the project was around 10 components. We estimated that Kafka can serve at least 100 nodes that generate traffic at 1,000 messages per seconds each with a limited CPU usage, which is more than enough for our target use cases. However, this does not mean that this is possible in any

deployment scenario, because we did not take into account the impact of Internet links. Low bandwidth and packet loss over the Internet is a common problem for any distributed cybersecurity framework; to mitigate this problem, the ASTRID platform should be deployed in the same virtualization infrastructure as the main service, as discussed in D1.3.

ACKNOWLEDGMENT

This work was supported in part by the European Commission under Grant Agreement no. 786922 (ASTRID).

REFERENCES

- [1] “D2.1 – programmable components and context models,” October 2020, v2.0. [Online]. Available: <https://private.astrid-project.eu/Documents/PublicDownload/47>
- [2] M. Repetto and A. Carrega, “Efficient flow monitoring for virtualized applications with ebpf,” ASTRID project, Tech. Rep., July 2021. [Online]. Available: <http://doi.org/10.5281/zenodo.5113889>.
- [3] M. Repetto, M. Zuppelli, and A. Carrega, “Efficient, portable and extensible packet inspection with ebpf,” ASTRID project, Tech. Rep., July 2021. [Online]. Available: <http://doi.org/10.5281/zenodo.5121392>
- [4] “D2.6 – data handling: collection, fusion, harmonization,” March 2021, v1.0. [Online]. Available: <https://private.astrid-project.eu/Documents/PublicDownload/86>
- [5] J. Koshy, “Kafka ecosystem at linkedin,” Blog post, April 2016. [Online]. Available: <https://engineering.linkedin.com/blog/2016/04/kafka-ecosystem-at-linkedin>
- [6] “D1.3 – final astrid architecture,” October 2020, v1.1. [Online]. Available: <https://private.astrid-project.eu/Documents/PublicDownload/79>
- [7] A. Carrega and M. Repetto, “Cb-manager and lcp: enhance the security cloud with the programmability,” ASTRID project, Tech. Rep., August 2021. [Online]. Available: <http://doi.org/10.5281/zenodo.5156076>