Improving service-level agreements for critical systems using big data monitoring techniques

Ioan Cristian Schuszter[1] and *Marius* Cioca[2]

[1] University of Petrosani, Department of Computer and Electrical Engineering, Petrosani, Romania
[2] "Lucian Blaga" Faculty of Engineering, University of Sibiu, Sibiu, Romania

Abstract. The proliferation of big data in virtually every branch of society and industry comes with the need to adapt and develop monitoring and alerting systems in such a way that the system can cope with any kind of data stream, whilst also ensuring rapid response times. This paper presents a framework based on modern open-source technologies that can be used to improve the quality and reliability of a connected system (such as an industrial control system), through effective monitoring and alerting. Service level agreements are crucial in our modern society, where failures need to be detected quickly and effectively, especially when one is providing a service and every moment of downtime means a large quantity of lost money and potential customers, thus monitoring is essential. Benefits in terms of responsiveness and lower downtime are also discussed, with an emphasis on a prototype implementation for a major non-profit organization.

1 Introduction

Manufacturing lines need to be constantly running at maximum capacity, as they are important centerpieces of the whole production chain. Any downtime can imply huge financial losses for companies; even the quality of power is very important and leads to long-duration downtimes, according to a European power survey [1]. Vegunta et al. argue that large consumers affected by downtime can lose around 83.000$ in just one afternoon of downtime of the manufacturing line [2].

Engineers worldwide have thoroughly studied the problem of downtime in production systems for decades. Research on the collection of downtime data goes back to the 1990s, with engineers at the Ford Motor Company publishing guidelines based on their assembly lines and data they had collected [3]. However, today's manufacturing line is more and more digital, and a large portion of these down times could be due to computer systems or software failing and monitoring all these Internet of Things (IoT)-like systems becomes a challenge in the wake of Industry 4.0 and very large data being produced by these computers systems [4].

Systems, especially computer systems, are historically visualized as black boxes once they are deployed to a production site, with issues being very hard to detect and fix. Developers are not able to fully replicate the production environment on their personal machines, or the system is a mix between hardware and software components that are set up in a very precise way.

Sillito and Kutomi show that most failures that they studied are related to a code change or configuration deployment. Additionally, the authors observe that the automation of system monitoring greatly helps in timely detection of incidents as they occur [5]. As such, it is important to configure an efficient monitoring infrastructure which can aid the maintainers of a system to identify issues and fix them as soon as possible.

Another approach to ensure more resilient industrial systems is by using distributed programming. The applications are split into several different independent pieces, running on different environments. Through this approach, the failure of one component does not lead to the failure of the entire system [6].

Monitoring models have been widely discussed in the literature, but the following four activities generally comprise them, as presented in the paper by Mansouri-Saman and Sloman [7]:

- Generation: the actual production of the monitoring data
- Processing: a service is responsible for the aggregation of the monitoring data
- Dissemination: the processed monitoring data is sent to the parties interested
- Presentation: the information gathered is displayed in an understandable way through dashboard

The monitoring system developed performs all the duties previously described, while also taking into consideration the scale of the data processed and the scalability requirements that it will have for the future. More than **60000** people from institutions all over the world access computing services provided by the European Organization for Nuclear Research (CERN), most of it protected by Single Sign-On. The system needed to take into account a large number of sessions per user and rapid response in case of failure.

Several solutions from the field of "big data" have emerged as general computing platforms that can be scaled horizontally indefinitely. They are further discussed in **Section 2**.

2 Technology considerations and background

2.1 From batch to stream processing

With the growing size of data produced by computer systems, several "big data" solutions have emerged in order to handle the growth. Initially, Hadoop [8] surfaced from the engineers at Yahoo, being composed of a distributed file system as well as a *batch* processing system.

However, batch jobs are not particularly suited for the requirements of satisfying service-level agreements, as alerts would reach the personnel on duty later than it would be practical. The system design needed to consider a high-throughput and low-latency scenario.

Over time, streaming applications for big data systems have become more commonplace, with several powerful frameworks emerging. One of these early processing frameworks with streaming capabilities is *Apache Spark*, which provided a novel approach to data processing by partially storing job data and results in-memory, yielding much better performance than the competing MapReduce framework [9]. Recently, more powerful real-time streaming platforms have emerged, such as *Apache Flink* and *Apache Storm* [10, 11], allowing real-time aggregation of data, in-memory, across many machines. The architecture presented in this paper could be potentially improved by these technologies in the future, as currently the system metrics collected are served close to the raw form.

Message queues have also reached a certain level of maturity, ensuring high throughput of data but not offering the same analytics capabilities as the technologies previously described. They represent one part of the architecture that will be discussed further on.

2.2 Large-scale monitoring tools

Several different technologies were considered for the design of the system, the key point being that they should be well maintained and with an active community, as well as ideally adhering to open-source values, simplifying the process of contributing any potential fixes. The necessary technologies are split into several different categories.

2.2.1 Data ingestion

In order to build the monitoring system, the first component needed is the one installed alongside each monitored component in the system. The component in question is an application-programming interface (API) using Representational State Transfer (REST) for communication over the web. It needs to be able to ingest and parse log data in varying formats, as well as sending it for further archiving and processing. Another important aspect is its ability to scale, as many log messages might produce a spike that needs to be handled.

Two different software solutions are compared: *Apache Flume* and *Logstash*, with *Flume* being chosen as the more suitable candidate for the system due to lesser memory usage, more permissive licensing and comparable performance. **Table 1** presents the findings of the comparison. Using JavaScript Object Notation (JSON) logs addresses a multi-line log parsing issue, thus the system is able to contain multi-line information in a single-line document.

Table 1. Logstash vs. Flume evaluation

Comparison feature	Logstash	Flume
Memory used per container	900 Mb	150 Mb
License	Elastic and Apache 2.0	Apache 2.0
Multi-line parsing	Yes	No
Flexible Input/Output sources	Yes	Yes

2.2.2 Data processing and storage

As data logs need to be stored but also queried (regardless of their volume), this poses a challenge for the construction of the system. *Elasticsearch* clusters are fed data from the Flume instances, using *Apache Kafka* as a message queue for the temporary storage of data, based on general the Information Technology (IT) department infrastructure provided by the CERN Unified Monitoring Architecture [12]. In this way, data which is months old can be queried in a matter of seconds.

2.2.3 Alerting and metrics

In order to be aware of what is always going on with the running system, logs are not enough to tell the current state of the infrastructure. As such, metric collection is as crucial as log collection. If dashboards are set up to show real-time trends of these metrics, one can easily understand why something is misbehaving.

Prometheus [13] is an open-source time-series monitoring solution that produces metrics as key-value pairs. The metrics can represent any aspect of a system, such as memory or CPU usage, response times, or simply up / down status. It scales very well for large systems or systems with many clients (in the case of IoT) [14] and features its own query language for complex aggregation on the fly.

Grafana is a flexible open-source platform used for building interactive and real-time dashboards, using data from varied sources, including Prometheus, this being one of the main reasons it was chosen for the architecture presented in this paper. The solution has the possibility of alerting system maintainers, assuming certain conditions applied to the metrics are satisfied. Grafana is used widely and successfully in the monitoring of large-scale systems, such as High Performance-Computing within the German Climate Data Centre [15]. Additionally, the system uses the alerting features provided by Grafana, by evaluating several rules based on the metrics produced by Prometheus.

3 Proposed architecture

The system architecture leans on the Unified Monitoring Infrastructure deployed by CERN to service all the large experiments and IT services that are part of the Large Hadron Collider [12]. The ingestion part of the system makes use of a Kafka broker to send logs from all the VMs and Kubernetes containers that need to be monitored.

Two specific requirements needed to be addressed, the first one being the *traceability* of actions taken on the system and the second one being the *alerting* produced in case of abnormal behaviour. As such, the system produces two different types of monitoring metrics, each with its own use-case, with the first one being useful in the situation where information on raw logs is needed, such as the number of failing requests in the last hour or the changes performed on a certain entity. The system that is being monitored by this architecture is a complex service providing authentication and authorization functionality deployed at CERN, its uptime is critical to all those that need to access restricted services.

All the services being monitored can be elastically scaled as they are deployed to the open-source container platform named *Openshift*. The benefits of container-based deployments for cloud services have been heavily discussed in the literature, as such the system heavily relies on the auto-scaling and recovery features of *Kubernetes* [16].

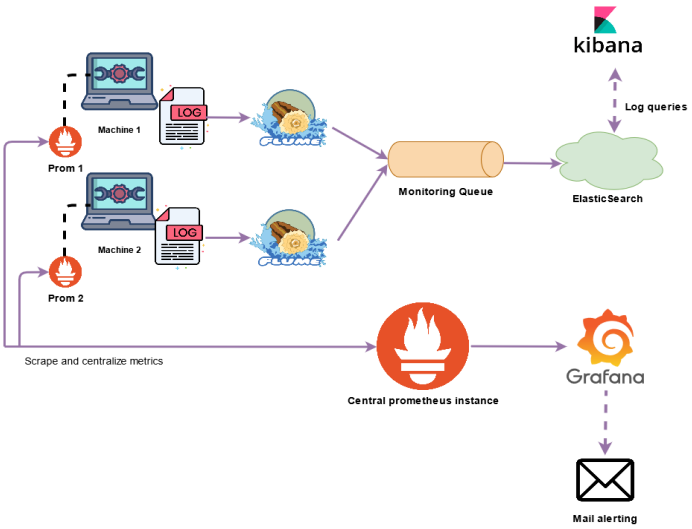

Fig. 1. Diagram of the full system architecture

3.1 Flume setup

As presented in **Section 2**, *Apache Flume* is used to collect logs from the, this subsection goes a bit further in detail on the implementation aspects of the log collection. As the main service being monitored is a REST API, logs are being served as JSON documents. In order to avoid any potential lost logs, they are published to a common volume shared between the API container and the Flume container.

Flume listens to changes in the files on the volume, so whenever new logs are produced, it picks up from where it stopped. A checkpoint is stored in a separate JSON file (*flume-checkpoint.json)*, so that the restart of the container does not cause all the previous logs to be re-published. The diagram of the setup can be consulted in **Figure 2**. The logs contain several extra fields other than the message, such as the *log level*, which allows filtering important events as opposed to informational ones, *timestamp* for accurate timekeeping and the *logger* to know exactly where the information comes from.

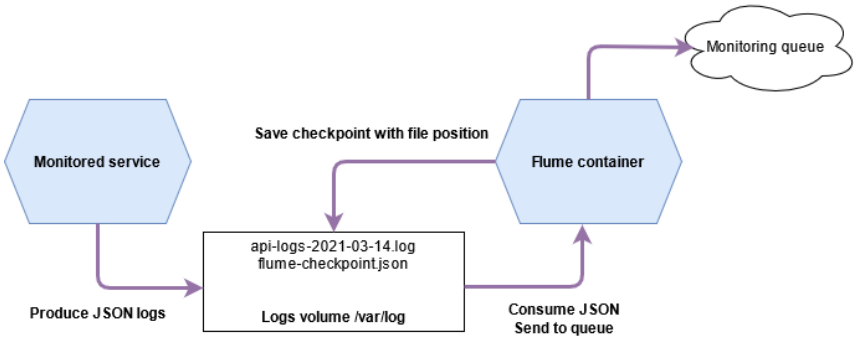

Fig. 2. Architecture diagram of the log ingestion system

3.2 Prometheus setup

Prometheus is setup individually for each of the services monitored, as well as providing one central instance that aggregates the data and serves it to the Grafana dashboard. Prometheus provides an option to add *federation*, meaning that metrics are collected from all the related services and aggregated with a common label. Every 5 seconds, the central instance queries the other secondary ones and gathers their metrics.

Prometheus metrics have a key and a variable number of labels that can be used to filter them. Examples are provided below, for the .NET core service that is being monitored. They can be easily exposed by implementing a few functionalities that are available for most common programming languages, if a library does not already exist.

process_virtual_memory_bytes 12879405056
process_working_set_bytes 788279296
aspnetcore_healthcheck_status{name="AuthorizationServiceContext"} 1
up{kubernetes_namespace = "authorization-service-api-dev"} 1

Once the Prometheus setup has been completed, the Prometheus query language can be used to filter out interesting properties of the service. For example, verifying how many requests/second the API is receiving, one can use the *http_requests_received_total* metric and sum over the rate:

sum(rate(http_requests_received_total{kubernetes_namespace="$instances",
controller=~"$controllers"}[3m]))

More complicated metrics can also be achieved, such as taking a histogram of response time and computing the 95-th quantile in order to know what is the upper bound in terms of time for 95% of requests:

histogram_quantile(0.95,sum by(le)
(rate(http_request_duration_seconds_bucket{kubernetes_namespace = "$instances"}[3m])))

3.3 Grafana setup

Since the Prometheus metrics are available on the central instance, the remaining step was connecting the dashboard to the data source and setting up several metric panels on a dashboard that the team can consult.

Uptime for the three different environments, requests per second, quantiles and heat-maps for the response times are among a few of the metrics evaluated by the dashboard every 10 seconds, alongside other simpler metrics such as the rate of CPU or memory usage of each of the pods. The full dashboard setup from Grafana is presented in **Figure 3.**

Fig. 3. View of the fully functioning dashboard setup (from Grafana)

Finally, alerts have been set up on the *up* metric of each of the services, rapidly alerting the team in case of failure or malfunction of the service, via an email message.

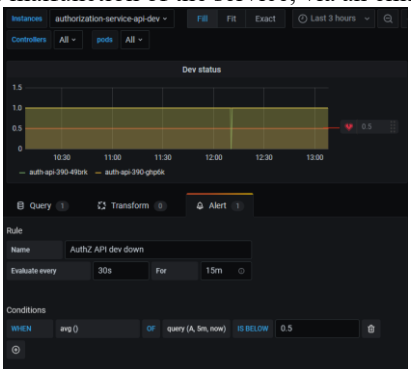

Fig. 4. Alerting setup panel (from Grafana)

4 Benefits of monitoring for Service Level Agreements

Service level agreements (SLA) are a form of formal contract established between the provider of a service and its consumers, establishing the parameters in which the system should always run. Recently, there have been several approaches to the problem and languages have been defined for expressing these SLAs for cloud computing services, highlighting the importance of such metrics in a service-oriented world [17].

Systems that are being heavily used by users daily can be considered as utility computing systems, hence customer satisfaction (and SLA fulfilment) is vital in these situations [18]. The system presented in this paper is not a subscription-based system, but it is required to be always available, and parallels can be drawn on the importance of satisfying these metrics and that of a utility service.

Examples of Service Level Agreements for the monitored service are:

- 99% of API requests must complete in under *Tx* seconds, where *Tx* is agreed upon with the customer.
- Monthly downtime must be less than *Ty* minutes.

The alarms that the Grafana dashboard provides can provide essential aid in order to maintain these parameters within the agreed limits.

4.1 System failure case study

A case study of incident response for the system is presented, highlighting the impact of a monitoring infrastructure being properly set up and configured to alert invalid states.

On the 10th of March, around 11:10 AM, the container infrastructure suffered from an out-of-memory error on several pods.

An email message immediately alerted the team, springing into action and noticing the containers failing from the out of memory errors. The alert triggered a re-deployment, and the system regained its availability. The re-deployment was visible on the dashboard, showing less than a minute of unavailability, as the new containers took over the load of the service while the platform terminated the failing one.

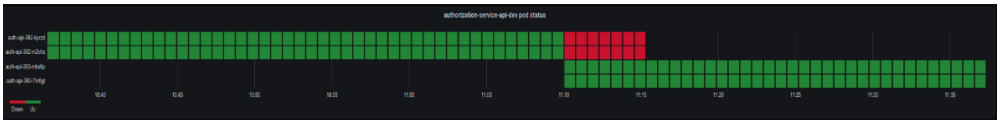

Fig. 5 Failure and re-deployment of the containers (from Grafana)

Without a proper monitoring setup, it is likely that this incident would not be resolved in such a timely fashion, as team members are generally performing their daily activities and they do not constantly check the availability of the service.

5 Conclusions

The monitoring architecture presented and implemented in this paper was already put to the test in a production environment where the availability of the service is highly critical, and it has shown to be the right tool for this particular use case.

In terms of advantages, the system provides near real-time traceability and analysis of the running service, allowing teams to react rapidly to any issue that might occur and to track whether the implemented fix that is enough to restore the state of the system.

The monitoring system is flexible enough to integrate with production-line systems, if they generate any sort of messages that a computer can interpret. A thin application layer on top of metrics collected by the system could be served to Prometheus in order to monitor values such as electricity voltage in real time, number of parts produced per hour, heat levels of the machines and so on. Thus, the full power of real-time monitoring is applicable to any industrial control system with very little intrusion on the system itself. A total halt of a production line could trigger a Grafana alert, like the ones that the API produces.

Future work will consist of analyzing system logs in real time to be aware of possible future failures, before they even have the chance of occurring. This is achievable through techniques such as machine learning and anomaly detection.

References

1. Leonard Energy, Poor Power Quality costs European business more than 150 billion euros a year, [Online]. Available: https://leonardo-energy.pl/wp-content/uploads/2017/08/Poor-power-quality-costs-european-business-more-than-150-billion-a-year.pdf, [Accessed 14 May 2021]
2. S. Vegunta, J. V. Milanovic, IEEE Transactions on power Delivery **26** (2009)
3. E. J. Williams, WSC'94, *Downtime data-its collection, analysis, and importance*, (San Diego, USA, 1994)
4. A. Khan, K. Turowski, IITI'16, *A Survey of Current Challenges in Manufacturing Industry and Preparation for Industry 4.0*, (Rostov-on-Don, Russia, 2016)
5. J. Sillito, E. Kutomi, ICSME'20, *Failures and Fixes: A Study of Software System Incident Response*, (Adelaide, Australia, 2020)
6. L. I. Cioca, M. Cioca, WSEAS Transactions on Information Science and Applications **4**, pp. 303-308 (2007)
7. M. Mansouri-Samani, M. Sloman, IEEE network **7**, pp.20-30 (1993)
8. K. Shvachko, H. Kuang, S. Radia, R. Chansler, MSST'16, *The Hadoop distributed file system*, (Incline Village, USA, 2010)
9. M. Zaharia et al, Communications of the ACM **59**, pp. 56-65 (2016)
10. https://flink.apache.org/, [Accessed 15 March 2021]
11. http://storm.apache.org/, [Accessed 15 March 2021]
12. A. Aimar, A. A. Corman, P. Andrade, S. Belov, J. D Fernandez., B. G. Bear, et al, Journal of Physics: Conference Series **898** (2017)
13. https://prometheus.io/docs/introduction/overview/ [Accessed 15 March 2021]
14. M. Brattstrom, P. Morreale, CSCloud'17, *Scalable agentless cloud network monitoring*, (New York, USA, 2017)
15. E. Betke, J. Kunkel, HiPC'17, *Real-time I/O-monitoring of HPC applications with SIOX, Elasticsearch, Grafana and FUSE*, (Jaipur, India, 2017)
16. D. Bernstein, IEEE Cloud Computing **1**, pp. 81-84 (2014)
17. P. Patel, A. H. Ranabahu, A. P. Sheth, Service Level Agreement in Cloud Computing, [Online]. Available: https://corescholar.libraries.wright.edu/knoesis/78/, [Accessed 14 May 2021]
18. L. Wu, R. Buyya, *Performance and dependability in service computing: Concepts, techniques and research directions*, (IGI Global, 2012)