

Extended Metrics Project for Apache Airflow

Julian Legler, Marcin Ozimirski, Matthias Seiler

Abstract— Due to the data characteristics of scientific experiments, Scientific Workflow Management Systems (SWMS) emerged to process massive amounts of data in parallel data pipelines. While SWMS became more powerful, the complexity of the system and its possible workflows increased, which requires to closely monitor the activity within the system. While today's SWMS expose built-in metrics regarding their workflow and tasks, none of them is able to gather information about low-level metrics from the operating system or hardware. We propose a microservice oriented solution to monitor events from the Linux kernel using eBPF. Our system demonstrates that it is possible to enrich monitoring with a wide range of metrics, such as cache statistics, task waiting time, system calls and more. We imagine such low-level metrics to be extremely helpful for developers and administrators, in order to identify potential bottlenecks and bugs in SWMS pipelines.

Index Terms—Scientific Workflow Management Systems, Monitoring, eBPF

I. INTRODUCTION

A scientific workflow management system (SWMS) is a commonly used software solution in the field of computational science. It can execute a series of calculations to examine (big) data sets in a structured and organized manner. Such computations are commonly called workflows consisting of tasks related to one another. The Management System is responsible for a transparent orchestration and execution of specified tasks, considering dependencies between them. The software allows for an efficient way to process and extract information from the examined data sets, leading to scientific advances in various fields such as biology, physics, and astronomy[1]. Yet, such workflows are often resource-intensive and require a distributed, dynamic, and scalable architecture to achieve meaningful results in a reasonable amount of time. The cloud and microservice paradigms offer several advantages when deploying these systems, such as accessibility, flexibility, and scalability [2], which allow workflows to be deployed and executed dynamically. These characteristics are crucial, since with a changing workflow during execution also comes different requirements regarding resources. This could for example depend on an input data set, the parallelism of computed tasks, and their level of complexity. The available resource pool can scale to adjust the computing resources throughout the workflow execution process. However, depending on the technological solutions offered by cloud services, hardware and software factors can vary significantly from one provider to another, which leads to different processing speeds, resource utilization, and error ratio[3].

This paper was produced by students of the Technische Universität Berlin in the course "Master Project: Distributed Systems" in summer 2022. The source code is available in the GitHub repository <https://github.com/maseiler/swms-monitoring>

Built-in monitoring features provide information on the amount of time and resources needed for the computation of a particular task or resource utilization on the active virtual machines or containers. Unfortunately, for more intense calculations, like machine learning, data delivered by the built-in tools are often insufficient to provide enough information for deep analysis and detection of bottlenecks or finding opportunities for code optimization. In such scenarios, the more detailed (low-level) metrics available, the better conclusions can be drawn regarding the performance of the current solution. Producing exact information on the number of I/O operations, CPU load, or memory usage per system process allows for a more accurate analysis of executed workflows (or their tasks) and the infrastructure on which the calculations are performed. This would open doors for improving of the system architecture, hardware and code. However, the implementation of solutions capable of gathering such information or the extension of the already existing ones rests on the shoulders of the end users. This project intends to provide a solution allowing the collection of high- and low-level metrics, independently of underlying cloud technology and SWMS used for computational science.

The rest of the report is structured as follows: Section II evaluates previous work in this domain. Section III details the technology in our project, while Section IV discusses our approach on how to use these. Monitoring results can be obtained from Section V, followed by a discussion on limitations and future work in Section VI. Finally, we conclude or report in Section VII.

II. RELATED WORK

Due to the nature of high volume data, modern science is often conducted on high-performance computing infrastructures. In recent years, the complexity of both scientific experiments, as well as that of its underlying infrastructure has grown tremendously, reaching the point where it becomes challenging to execute the workflows in a reliable and error-free way. On the scale at which such projects are carried out it is almost certain that software or infrastructure anomalies will occur, which makes the detection and diagnosis of system failures more challenging than ever before. In order to address such issues, Valerio et al. created a prototype system monitoring events related to both activity and workflow execution. The premise of the design was to enable the user to answer the following questions at any time during and after the execution of the workflow:

- What is the status of the workflow?
- Did the user make changes to the workflow before executing it?
- Why did it fail at a certain point?

- What was the execution context of the workflow directly before it failed?
- How many resources does it consume?
- How has each activity in the workflow performed (in the past)?

The ability to answer the aforementioned questions can be guaranteed by a system monitoring the general properties of the workflow and its tiniest details. Therefore, this type of solution usually consists of various components that collect data at different layers, starting from the general parameters of the workflow through the infrastructure on which it is run, ending with individual processes responsible for performing calculations. In addition, the software must be able to map the relevant processes (which can be executed in a parallel and distributed manner) to the corresponding machines and tasks and publish them in a readable form. Of course, all this makes sense only if one can assign a chronological order of these processes concerning the time domain.

Furthermore, an environment in which such scenarios are executed also plays an important role. Regardless of whether the code is invoked in the operating system context or a virtualized container created for the task, it is often characterized by restrictive standards. Such standards fulfill the functions of stability and security and thus may not provide any or only limited opportunities for adjustment. However, without it, it may not be possible to access system values available to the system's kernel.

Fortunately, one does not have to develop everything from scratch as there are open technologies capable of solving a single or part of the listed conditions. Levin and Benson used Linux's extended Berkeley Packet Filter (eBPF) [6] software for a collection of system metrics in the containerized environment and exposure of these to a specified endpoint. In a further step, the authors used Grafana [7], an observability framework, to present the collected information using programmable dashboards. This strategy allows the collection of data from various sources and then processing and presenting it in the form of histograms and graphs, as described by the user. It also reflects well the current trend for creating monitoring solutions. It is based on a combination of available platforms and software to collect and present data that are

relevant to a carried-out project. For such approaches, it is not important what generates it or where the data comes from, as long as it is published in a suitable format, which is most often JSON or some derivative of XML.

We are trying to close this research gap combining SWMS monitoring solutions with low-level metrics using eBPF.

III. BACKGROUND

Today exists a vast number of workflow management systems, ranging from general purpose ones like Apache Airflow [8], Nextflow [9] or Pegasus [10] to domain specific ones e.g. Taverna [11] for bio-informatics and astronomy, Arvados [12] for biomedical data, and many more. Each of these naturally have their advantages and disadvantages over one another, due to their design and architecture and can also be found in different stages of maturity.

To begin with, we decided for Apache Airflow, since it serves the general purpose, is widely used and one of the most mature SWMS [13]. The workflow in Apache Airflow is described with a Directed Acyclic Graph (DAG), whose nodes consists of tasks that are being executed. The edges represent the dependencies and therefore the execution order of these tasks. The description of the DAG is specified by the user in a Python script. Figure 1 shows a DAG that runs tests in a Kubernetes environment [14].

As depicted in Figure 2, the basic architecture of Apache Airflow consists of the following components: the *User Interface* (UI), that is served by the *Webserver*, is the entry point for the user, and not only feature-rich, but also easy to use. At the heart of Apache Airflow resides the *Scheduler*, "which handles both triggering scheduled workflows, and submitting Tasks"[15]. It therefore requires access to a *DAG Directory*, in order to makes use of one of the provided *Executors* [16], that handle these tasks. The actual execution of tasks is performed by one or more *Workers*. Finally, a *Metadata Database* is "used by the scheduler, executor and webserver to store state"[15].

Apache Airflow provides built-in logging and monitoring [17], whose monitoring stack relies on StatsD [18], that gathers the internal metrics and exposes them to Prometheus. StatsD creates buckets and fills them with values for the specific

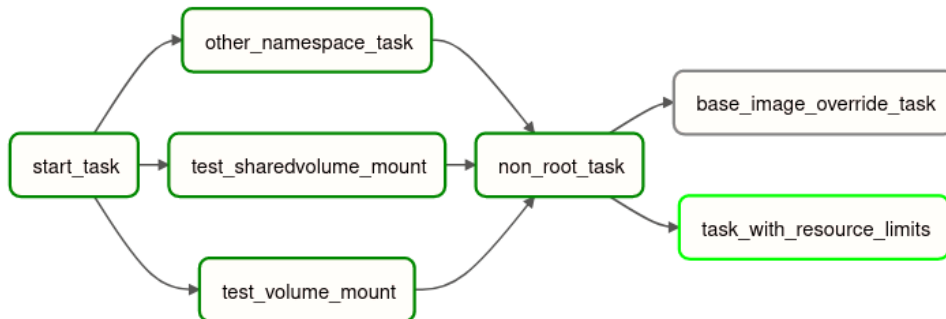


Fig. 1. Airflow DAG with five tasks run successfully (dark green), one currently running (light green) and one in queue (gray). The execution order is defined by the dependencies which are represented by edges between the individual tasks. After the `start_task` task is finished, the connected tasks can start in parallel. Afterwards, the task `non_root_task` has to wait until `other_namespace_task`, `test_sharedvolume_mount` and `test_volume_mount` has successfully finished before it can get executed.

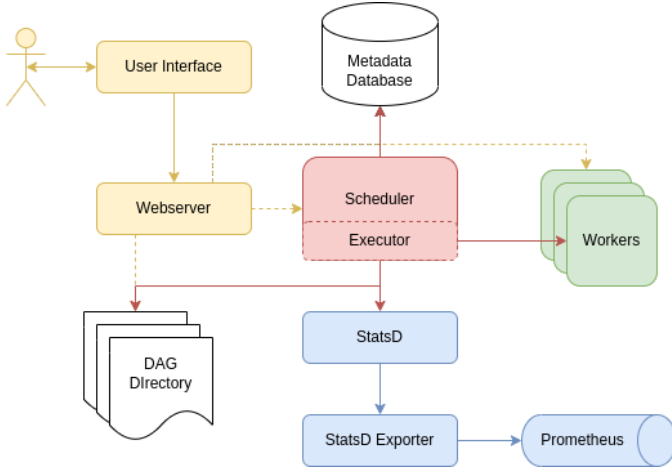


Fig. 2. Airflow architecture

metric, that will be aggregated after a predefined period and flushed to an endpoint for further processing. While there are numerous metrics on DAG and task level, none of them consider metrics on lower levels [19].

With eBPF [6] however, it is possible to gather metrics directly from the Linux kernel. It extends the kernel without changing the kernel source code or loading kernel modules and has therefore unrestricted access to all hardware. These little programs get triggered by certain hook point (e.g. system calls, function entry/exit, network events) and can thus be attached to almost anywhere into kernel or user applications [20]. eBPF programs are usually written using frameworks such as the BPF Compiler Collection (BCC) [21]. BCC provides a library and "Python, C++, and Lua interfaces for writing programs"[22] and many tools for efficient kernel tracing. Figure 3 depicts the involved steps to attach an eBPF program: in order to use the program in the Linux kernel, it expects the program to be in bytecode. With the `bpf()` system call, the program will be loaded into kernel space, where it first has to be verified (that it is actually safe to run) and secondly compiled to machine specific instructions. "This makes eBPF programs run as efficiently as natively compiled kernel code or as code loaded as a kernel module" [20].

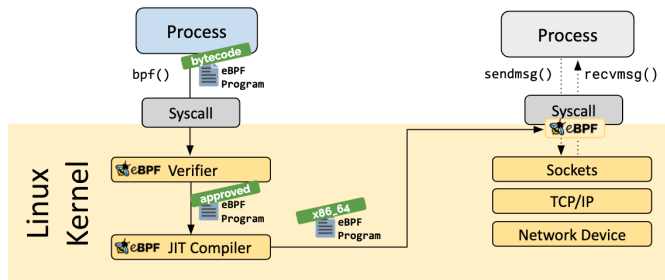


Fig. 3. Steps to attach an eBPF program [20]

In order to store collected information (e.g. in the form of metrics), eBPF leverages *eBPF Maps* [23], that can be accessed not only from kernel, but also from user space.

The data from these eBPF maps can be exported to a more

powerful monitoring systems such as Prometheus, Graphite or InfluxDB, and visualized with an analytics platform like Grafana or Kibana.

IV. APPROACH

As mentioned earlier, we will now describe how we built a monitoring solution on top of the SWMS Apache Airflow and leverage eBPF in combination with Prometheus and Grafana as the monitoring stack.

A. Deployment of Apache Airflow

Apache Airflow is a widely used and adopted SWMS and as such, there are many approaches for deployment. These include fully managed solutions like Google Cloud Composer, Amazon Managed Workflows for Apache Airflow and astronomer.io. The advantage of using such services is an easy and fast deployment. The downside is that one may lack some flexibility and have restricted access to more granular and in-depth functionalities and resources. An Airflow instance deployed by such services has often to be treated as a black box system. This approach may block some needed functionalities and access to the raw resources required to collect metrics at the level of granularity we try to achieve.

On the other end of the deployment complexity spectrum are deployment strategies that involve deploying or even compiling the required components on bare-metal servers. This requires the database and logging systems to be properly installed beforehand. At the same time, machines must be configured to form a cluster, and Apache Airflow must be installed and run on all of them. This could also be combined by first deploying Kubernetes and then using Kubernetes orchestration technologies to run all the necessary services in the cluster. This deployment strategy has the advantage of fine and granular access to all system resources, but the disadvantage of the overhead associated with managing everything oneself.

We chose a deployment strategy that combines the two mentioned above. Google Cloud Platform (GCP) is used as the provider for the needed computation infrastructure provided by their different cloud services. GCP provides managed resources and guarantees an easy-to-use process of deploying and managing the needed infrastructure. With the Google Kubernetes Engine (GKE) it even provides a managed Kubernetes deployment. Using this, the services no longer have to be installed on the individual machines, they can rather be just send to Kubernetes, which then orchestrates the needed services and deploy them on the available infrastructure. At the same time, one has full access to all the underlying components, like the virtual machines, Kubernetes and the Apache Airflow services running on. This combines the granular access to all resources, and the ease of using a managed service. The complexity of deploying and configuring multiple services in a Kubernetes cluster should not be underestimated, even when the cluster's infrastructure is managed by GKE.

GKE is configured to create three nodes of the type *e2-standard-4* which consist of 4 vCPUs and 16 GB RAM [24]. For the deployment of Apache Airflow on top of Kubernetes, the official Apache Airflow Helm Chart is used [25]. Using

a Helm Chart is a good way to reduce further complexity by using a pre-defined, out of the box running Apache Airflow configuration. The downside is, that the complexity is just covered by slim high-level configuration files. If deeper configurations have to be done, one still ended up to have to dive into the whole complexity of the used services plus the complexity added by Helm Charts to make the front-facing configs more elegant and dynamic.

The original Helm Chart [25] was altered in the following way to deal with our requirements.

1) *Executor*: There are different Apache Airflow Executors with different approaches on how to handle the execution of tasks. It would make sense to use the default Celery Executor or the Kubernetes Executor for this project. The Celery Executor is adding a queue to the deployment. On all available nodes, a worker will be deployed. The workers continuously fetch new jobs from the queue and execute them. The Kubernetes Executor, as shown in Figure 4, on the other hand, is changing the behavior in the way, that each node does not have only one worker, but rather for each job there is one worker which is terminated when the job is finished. The huge advantage of the Kubernetes Executor is, that this executor is using the labeling capabilities of Kubernetes. The Kubernetes Executor is very communicative regarding the worker name and metadata, which makes it easy to collect this information without accessing the machine [16][26]. The following data is attached to every worker pod when using the Kubernetes Executor:

- `dag_id`, which is the name of the executed DAG. This name is not unique, but helps with identifying the context.
- `run_id`, which is a timestamp and a unique ID that is assigned when triggering the DAG.
- `task_id`, which is the name of the executed task in this worker pod. This name is also not unique, but is important information on what is currently running inside this pod for later analyses.

Thanks to this information we can reliably identify the

current task, the containing DAG and the corresponding triggered run instance of the DAG. These labels are important gathering the metrics. Without knowing what's inside the pods, a lot of valuable information would get lost. Luckily, all this information is supplied simply by using the Kubernetes Executor. Alternative strategies would probably include writing own code to detect the running processes on the pods and then mapping them via the Airflow API to the corresponding DAG and workflow run. Our work in this direction was eventually abandoned because of the advantages offered by the Kubernetes Executor.

2) *StatsD*: StatsD is the monitoring service used by Apache Airflow. To get the metrics compatible to Prometheus, a mapping is configured so that the values can be interpreted correctly by Prometheus. The StatsD Exporter is shipped by default with the official Helm Chart for Apache Airflow.

3) *Webserver, Worker, Environment*: Here are just minor changes, so that the webserver can be accessed easily for executing test DAGs: (1) Apache Airflow knows the exact number of nodes the executor can use, (2) the environment parameters are changed to enable the backend authentication mechanism for accessing the Apache Airflow API, and (3) Apache Airflow is deployed with a set of sample DAGs for better testing and demonstration purposes.

4) *Prometheus and Grafana*: For our monitoring stack, we rely on Prometheus and Grafana, which we had to adjust to our new needs:

- Apache Airflows StatsD Exporter is added as a scraping target. Prometheus now collects the mapped metrics collected by StatsD every 15 seconds.
- Every endpoint orchestrated by Kubernetes is scraped every 5 seconds to collect the values collected by the eBPF Exporter.

The *kube-state-metrics* service is configured to send all labels and annotations from all pods in all namespaces and not stripping them away before sending them to Prometheus, which is the default behaviour and can lead to confusion.

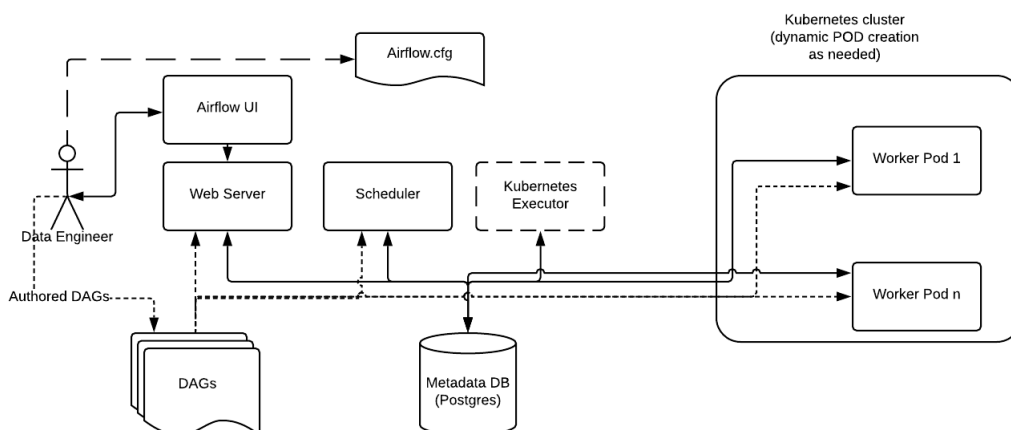


Fig. 4. Overview of the architecture when using the Kubernetes Executor. Showing the services and possible interaction by a user. Every task is run on an newly created worker pod that is permanently destroyed when the job has finished. The graphic is lacking the metric and logging part of Airflow which is shown in Figure 2. The graphic is from [26]

Because of the mapping already done via the StatsD exporter, the metrics are parsable and interpretable as normal Prometheus metrics and can be used to create custom Grafana dashboards. A few possible examples of metrics and how to visualize them are demonstrated in section V.

B. eBPF Exporter

Since we use Prometheus for metric gathering, we need an exporter that is able to publish the eBPF metrics. Developers at Cloudflare created an *ebpf_exporter* [27] that does exactly this. However, we had to adjust the exporter [28], so that we can retrieve the airflow labels *dag_id*, *task_id* and *run_id*. For this, a previously unaccepted pull request from the Cloudflare project was used as an example on how to use eBPF to get information from Kubernetes [29]. This pull requests contain an example implementation for exporting the eBPF metrics using also information provided by Kubernetes about the containing node or pod. The implementation was used as a starting point, but had to be completely re-implemented as they used the *dockerd* implementation and therefore the Docker API which is no longer supported in newer Kubernetes versions [30]. Because of the lack of documentation for the now used containerd API, the code was rewritten to use the Kubernetes API. This introduces new problems, because now one also has to deal with authentication mechanics, as the Kubernetes API can manage the whole cluster and has no internal pod specific permissions for just getting information on ourselves. To tackle this, a cluster role binding with the needed permissions is used, as suggested by [31]. With these changes the *dag_id*, *task_id* and *run_id* are published by the eBPF exporter correctly together with the selected metrics. These labels can be used to filter the results in Grafana.

To make the deployment as easy as possible, we created a Helm chart using a Docker image, where one can supply a configmap with eBPF programs. Per default we provide five examples [32]:

- *biolatency* traces the block device I/O (disk I/O), and records the distribution of I/O latency,
- *cachestat* records page cache accesses,
- *dcstat* exposes statistics on the Linux directory entry cache,
- *runqlat* monitors how long tasks spent waiting their turn to run on-CPU, and
- *syscalls* measures the latency of system calls, including the endpoint and audit logging.

The configmap can be easily extended with additional eBPF programs.

V. RESULTS

In the following we will show some results of our system, when we executed the DAG *example_complex* [33] provided by Apache Airflow on three worker nodes. This workflow was executed around 11:36:30 and completed within 2:50 minutes.

As we can see by the colors in Figure 5, there is increased activity between 11:37:00 and 11:39:00, with a peak (red square) where over 5700 CPU tasks had to wait between 8

μ s and 16 μ s before they could be processed by the CPU. A similar behaviour can be observed in Figure 6, which shows that the average waiting time is around 185 μ s and the maximum 90th and 95th percentile at 243 μ s and 1.58 ms respectively.

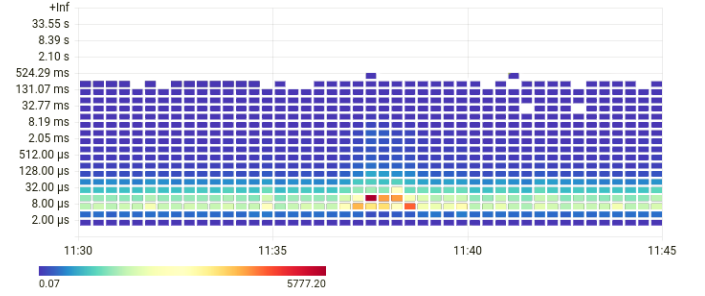


Fig. 5. Heatmap of run queue latency of a single node.

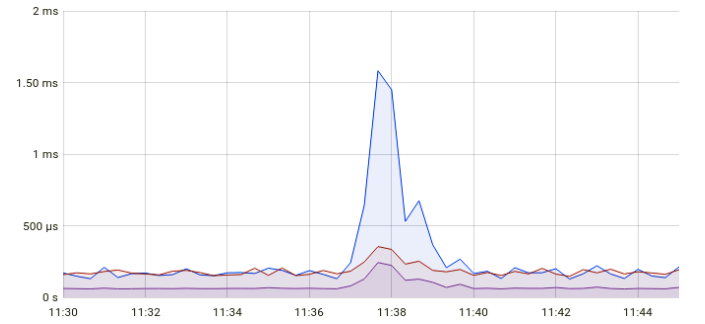


Fig. 6. Average (red), 95th percentile (blue) and 90th percentile (purple) of run queue latency on a single node.

In regard to system calls, we can see in Figure 7, that *do_syscall_64*, which was instructed from user space to perform a system call, had higher latencies before the DAG was triggered.

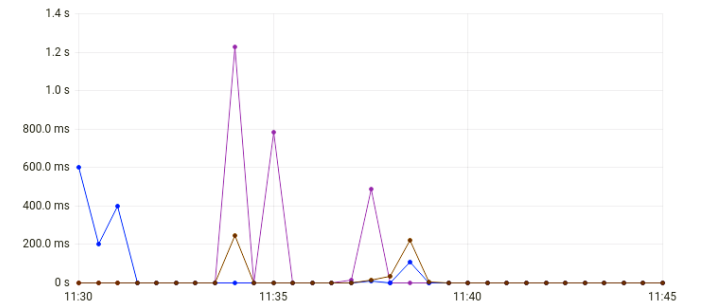


Fig. 7. System call latency for three worker nodes.

The cache statistics are unfortunately not very meaningful, presumably because the exporter could not keep up with larger spikes of data. Nonetheless, one can observe in Figure 8 that there has been a large increase in cache reads during workflow execution.

VI. DISCUSSION

While our five examples demonstrate that it is possible to extend metrics for SWMS using eBPF, it does not fulfill the

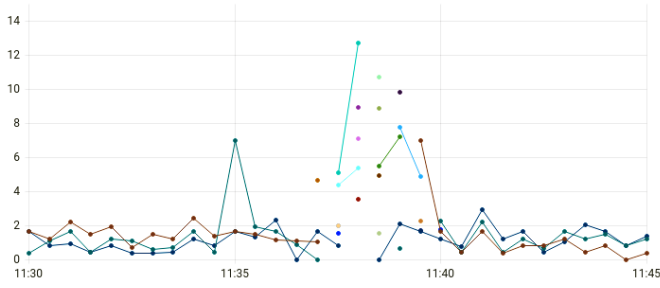


Fig. 8. Cache reads for three worker nodes.

scope we imagined. The eBPF exporter strongly depends on the underlying hardware and its access through the virtualization layer. Unfortunately, GKE does not allow access to certain `perf` events and kernel modules (e.g. CPU cycles, CPU instructions, cache misses, ...), limiting the number of available programs quite significantly. This issue pertains also in other virtualization solutions [34][35][36]. Thus, deploying our solution with k3s on Google Cloud’s virtual was also not successful. We assume, that this issue does not occur on bare-metal hardware, but were not able to verify this hypothesis yet.

Furthermore, GKE does not always deploy the eBPF exporter successfully and needs some retries. Unfortunately, we were not able to solve this issue.

Apart from that, we attempted to monitor the execution of Nextflow, another popular SWMS. Nextflow requires access to a persistent volume claim storage with a `ReadWriteMany` access model, which at the moment of writing this report, is not one of the services provided by the GKE. The issue can be solved via the usage of network file system packages, which are available in the helm charts. Another obstacle was the access rights of the service account used by the Nextflow driver to dynamically create and modify pods in the cluster. Such privileges are necessary for the driver to be able to instantiate pods and assign them tasks for the correct pipeline execution. The working solution was to grant the account administrative rights. It is not a secure approach, because it grants all possible permissions to all pods created by the driver in the corresponding namespace. Yet, for the sake of this project, such risk is acceptable. Another rather unexpected obstacle appeared to be the execution commands from the official documentation [37] used to run the workflow on the cluster. In the case of GKE, the received response from the server was unintelligible and despite the attempts made, could not be decrypted. The research led to *segeralabs* GitHub repository [38], which provides an alternative to the mentioned commands in the form of a bash script. The execution of the workflows became possible when using it. However, even then, still, another problem emerged. After triggering the workflow, the SWMS terminates its execution for an unknown reason, pointing at the external system as the cause. A further investigation pointed to a discussion from Nextflow’s official repository [39], where other users encountered similar behaviour. According to the users, the issue is caused by Nextflow pods not publishing their results to

the locations where the Nextflow driver is attempting to read them. An in-depth analysis of the created pods and the settings used to run the pipelines showed that the PVC settings are set as described in the discussion, and yet the error continued to occur. At this point, due to a large amount of time spent on the ineffective attempt to run Nextflow on the GKE cluster, the decision was made to discontinue further study, and devote the remaining time to improving the features of the proposed solution that work properly. The complete write-up of the approach is available in the project’s Git repository.

A. Future Work

As stated in previous chapters, we were able to demonstrate a proof of concept, that could become relevant in industry, if the above issues were solved. Our solution with BCC has manifold opportunities to gather more system information. On top of that, one could integrate `bpfftrace` as an alternative to BCC, to get an even richer set of eBPF programs. In order to deploy our monitoring solution to other cloud providers (e.g. AWS, Azure), other setup scripts are needed. Due to the above mentioned issue, a solution for bare-metal machines using k3s [40] would be extremely valuable. More than that, other SWMS should be supported to attract a wide range of users and developers. Making the metrics loosely coupled is a crucial step hereby. Ultimately, a small benchmark suite would be very helpful to evaluate the overhead of the eBPF exporter on the overall system performance.

VII. CONCLUSION

In this paper we presented the Extended Metrics Project for Apache Airflow, which can collect fine-grained and low-level metrics by utilizing the power of eBPF, to get access to kernel space information. The identification and connection of the running Airflow DAGs, tasks, processes, pods and nodes to create meaningful metrics is done by use of existing features and configuration options of Kubernetes and Apache Airflow, so that in the end no individual solution had to be programmed. This was possible because of already existing projects that did some previous work in the corresponding areas separate from each other. These solutions are linked together and are expanded to deal with our requirements. A project is delivered that shows the capabilities and prove the possibility to use eBPF to get very low-level information even from within the kernel space of the individual machines. The solution can easily be extended by deploying other eBPF programs and creating custom dashboards in Grafana.

ACKNOWLEDGMENTS

We thank our supervisor Sören Becker for his enormous support and patience.

REFERENCES

- [1] Y. Gil, E. Deelman, M. Ellisman, T. Fahringer, G. Fox, D. Gannon, C. Goble, M. Livny, L. Moreau, and J. Myers, “Examining the Challenges of Scientific Workflows,” vol. 40, no. 12, pp. 24–32.

- [2] A. Apostu, F. Puican, G. Ularu, G. Suciu, G. Todoran *et al.*, “Study on advantages and disadvantages of cloud computing—the advantages of telemetry applications in the cloud,” *Recent advances in applied computer science and digital services*, vol. 2103, 2013.
- [3] R. Aljamal, A. El-Mousa, and F. Jubair, “A comparative review of high-performance computing major cloud service providers,” in *2018 9th International Conference on Information and Communication Systems (ICICS)*, pp. 181–186.
- [4] M. D. Valerio, S. S. Sahoo, R. S. Barga, and J. J. Jackson, “Capturing Workflow Event Data for Monitoring, Performance Analysis, and Management of Scientific Workflows,” in *2008 IEEE Fourth International Conference on eScience*, pp. 626–633.
- [5] J. Levin and T. A. Benson, “ViperProbe: Rethinking Microservice Observability with eBPF,” in *2020 IEEE 9th International Conference on Cloud Networking (Cloud-Net)*, pp. 1–8.
- [6] eBPF - Introduction, Tutorials & Community Resources. [Online]. Available: <https://ebpf.io/>
- [7] Grafana: The open observability platform. Grafana Labs. [Online]. Available: <https://grafana.com/>
- [8] Home. Apache Airflow. [Online]. Available: <https://airflow.apache.org/>
- [9] A DSL for parallel and scalable computational pipelines — Nextflow. [Online]. Available: <https://www.nextflow.io/>
- [10] Main. Pegasus WMS. [Online]. Available: <https://pegasus.isi.edu/>
- [11] Taverna - Apache Incubator. [Online]. Available: <https://incubator.apache.org/projects/taverna.html>
- [12] Arvados Homepage — Arvados. [Online]. Available: <https://arvados.org/>
- [13] B. Harensiak and J. de Ruiter, *Data Pipelines with Apache Airflow*. Manning Publications Company, 2021.
- [14] “Apache Airflow,” The Apache Software Foundation. [Online]. Available: https://github.com/apache/airflow/blob/ffee6bceb32eba159a7a25a4613d573884a6a58d/airflow/example_dags/example_kubernetes_executor.py
- [15] Architecture Overview — Airflow Documentation. [Online]. Available: <https://airflow.apache.org/docs/apache-airflow/stable/concepts/overview.html>
- [16] Executor — Airflow Documentation. [Online]. Available: <https://airflow.apache.org/docs/apache-airflow/stable/executor/index.html#executor-types>
- [17] Logging and Monitoring architecture — Airflow Documentation. [Online]. Available: <https://airflow.apache.org/docs/apache-airflow/stable/logging-monitoring/logging-architecture.html>
- [18] “StatsD,” statsd. [Online]. Available: <https://github.com/statsd/statsd>
- [19] Metrics — Airflow Documentation. [Online]. Available: <https://airflow.apache.org/docs/apache-airflow/stable/logging-monitoring/metrics.html>
- [20] What is eBPF? An Introduction and Deep Dive into the eBPF Technology. [Online]. Available: <https://ebpf.io/what-is-ebpf/>
- [21] “BPF Compiler Collection (BCC),” IO Visor Project. [Online]. Available: <https://github.com/iovisor/bcc>
- [22] C. Cassagnes, L. Trestioreanu, C. Joly, and R. State, “The rise of eBPF for non-intrusive performance monitoring,” in *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium*, pp. 1–7.
- [23] Types of eBPF maps — Prototype Kernel 0.0.1 documentation. [Online]. Available: https://prototype-kernel.readthedocs.io/en/latest/bpf/ebpf_maps_types.html
- [24] Maschinenfamilie für allgemeine zwecke — compute engine-dokumentation — google cloud. [Online]. Available: <https://cloud.google.com/compute/docs/general-purpose-machines?hl=de#e2-standard>
- [25] Helm chart for apache airflow — helm-chart documentation. [Online]. Available: <https://airflow.apache.org/docs/helm-chart/stable/index.html>
- [26] Kubernetes executor — airflow documentation. [Online]. Available: <https://airflow.apache.org/docs/apache-airflow/stable/executor/kubernetes.html>
- [27] cloudflare/ebpf_exporter - prometheus exporter for custom ebpf metrics. [Online]. Available: https://github.com/cloudflare/ebpf_exporter
- [28] J. L. “Ebpif_exporter.” [Online]. Available: https://github.com/JulianLegler/ebpf_exporter
- [29] cloudflare/ebpf_exporter - prometheus exporter for custom ebpf metrics - add kubernetes context decoder. [Online]. Available: https://github.com/cloudflare/ebpf_exporter/pull/75/commits
- [30] Don’t panic: Kubernetes and docker. [Online]. Available: <https://kubernetes.io/blog/2020/12/02/dont-panic-kubernetes-and-docker/>
- [31] Using rbac authorization. [Online]. Available: <https://kubernetes.io/docs/reference/access-authn-authz/rbac/>
- [32] Ebpf_exporter/examples at master · cloudflare/ebpf_exporter. GitHub. [Online]. Available: https://github.com/cloudflare/ebpf_exporter
- [33] “Apache Airflow,” The Apache Software Foundation. [Online]. Available: https://github.com/apache/airflow/blob/ffee6bceb32eba159a7a25a4613d573884a6a58d/airflow/example_dags/example_complex.py
- [34] krrish_007. “perf” alternate performance management tools that can run on Google Cloud Platform. Stack Overflow. [Online]. Available: <https://stackoverflow.com/q/60988282>
- [35] Jason. Running perf within a VirtualBox guest. Stack Overflow. [Online]. Available: <https://stackoverflow.com/q/49866115>
- [36] Re: Perf CPU profiling in Xen guest fail & workaround — Linux Perf Users. [Online]. Available: <https://www.spinics.net/lists/linux-perf-users/msg02385.html>
- [37] Nextflow’s documentation. [Online]. Available: <https://www.nextflow.io/docs/latest/kubernetes.html>
- [38] Nextflow on kubernetes: Best practices. [Online]. Available: <https://github.com/sequeralabs/nf-k8s-best-practices>
- [39] Issue “command.run: No such file using k8s” - Nextflow Git Repository. [Online]. Available: <https://github.com/nextflow-io/nextflow/issues/1606>

[40] K3s: Lightweight Kubernetes. [Online]. Available:
<https://k3s.io/>