SN Computer Science (2021) 2:448
https://doi.org/10.1007/s42979-021-00830-2

ORIGINAL RESEARCH

Monitoring a CI/CD Workflow Using Process Mining

Ana Filipa Nogueira[1] · **Mário Zenha-Rela**[1]

Received: 31 March 2021 / Accepted: 20 August 2021 / Published online: 7 September 2021

Abstract
Process mining (PM) is a unique approach to extract workflow models of actual real-world activities, namely those related to software development. To be efficient and produce more reliable results, its algorithms require structured input data. However, actual real-world data originate from multiple heterogeneous sources; thus, integration and normalization are required preparatory steps before applying PM techniques. This problem is exacerbated by the need of performing this analysis in real time, rather than off-line in a batch-style approach. In this paper, we show how Apache Kafka pipelines can be used to support the integration and normalization of the event logs from multiple sources into data streams that feed the process mining algorithms in real-time. An application to the complex CI/CD pipeline of a major European e-commerce company is presented, showing that these techniques provide means to monitor and have higher observability of development processes.

Keywords Process mining · Kafka streams · XES · Process monitoring · Observability · Continuous integration · Continuous delivery · CI/CD · Software engineering

Introduction

In the last two decades, process mining (PM) techniques gained visibility due to their capabilities to model real-world processes and extract workflow models from data available in the information systems. Moreover, enterprises from different domains leveraged PM techniques to support their digital transformations. Examples of these domains include banking [1], healthcare [2, 3], security [4], and software development [5]. PM algorithms fall under one of three categories: *discovery*, *conformance checking*, and *enhancement*. The objective is to extract knowledge from the large amount of data collected by today's IT systems [6].

Software development poses itself as a challenging domain to model mainly due to the very diverse nature of possible projects, as well as the teams' characteristics and culture [7]. Even inside the same company, different development units can work quite differently. Nevertheless, some processes are usually designed to be shared across the organization to avoid repetitive work while ensuring the accomplishment of specific quality requirements.[1] For instance, DevOps teams require the internal adoption of identical Continuous Integration/Continuous Delivery (CI/CD) pipelines to support automation of testing and deployment. However, (e.g.) teams that maintain legacy code are frequently exceptions since the adaptation cost would offset the normalization benefits.

Process mining applied to the software development activities provides an evidence-based view of the actual activities and steps that a component goes through while it moves along the development pipeline, from a defect ticket or a new feature to the deployment of a revised software version. A major difficulty in applying PM techniques to software development processes is the integration of the data—referred to as the *event logs*— generated by the multiple and heterogeneous systems that support software development

This article is part of the topical collection "New Paradigms of Software Production and Deployment" guest edited by Alfredo Capozucca, Jean-Michel Bruel, Manuel Mazzara and Bertrand Meyer.

✉ Ana Filipa Nogueira
afnog@dei.uc.pt

Mário Zenha-Rela
mzrela@dei.uc.pt

[1] Department of Informatics Engineering, Centre for Informatics and Systems of the University of Coimbra, Coimbra, Portugal

[1] https://www.software-quality-assurance.org/cmmi-organizational-process-definition.html.

(e.g., version control, ticketing, testing, integration, deployment) [7]. Not only the systems are different by its nature, but also development teams may have different tools (e.g., SVN and Git) for the same purpose. This paper addresses this topic by proposing a way—via Apache Kafka— to collect and stream data that helps teams monitor their work.

Apache Kafka® is an open-source high-throughput distributed streaming platform.[2] It can be used as the single source of truth that connects every software/system component in an organization to every event log produced. Kafka provides the means for an architecture to evolve from a static perspective to continuous streams of events that deliver insights about the current state of their processes in near real-time. In this research, we adopted Kafka as the integration framework allowing the normalization of data from different tools into a standard format, available inside a common platform. In our specific context, the adoption of Kafka as the unified source of inputs for gathering process development insights emerged naturally as it is the technology supporting the event-driven architecture in place for the business processes.

Our subject of analysis is the CI/CD pipelines used by the software development unit of a major retail (e-commerce) European company operating in the global market. In this context, different integration orchestrators are used—Jenkins[3] and CirleCI[4]—due to the need to evolve and adapt to new architectures (e.g., cloud, event-driven) while simultaneously being able to support existing components that will not evolve in the near future. Hence, the team felt the need to *observe* what was happening in the CI/CD pipeline and collect an extensive set of integrated metrics, namely the total number of deploys and the time between deployments in different environments. A Kafka-based framework, which continuously streams events from multiple CI/CD pipelines and tools, was implemented. It comprises components that aggregate, normalize and stream the data; the normalized data are then ready to be fed into the PM algorithms. The raw data, non-aggregated and aggregated, and the output from algorithms' analysis can be used as input for dashboards, reports, and alerts. To the best of our knowledge, there is no prior research on the application of PM to monitor CI/CD pipelines. The literature includes research that collects data from development tools to attain a higher level of observability on their processes [8–11], but none uses PM techniques. PM algorithms can provide different types of analysis, including checking the conformity of steps defined in the pipelines and the social networks associated with them. By social networks, we mean the interactions between team members; for example, a developer's commit triggers one build, but another colleague or the manager initiates the deploy and tests stages. This type of information provides visibility on how teams work.

In this research, we use process mining techniques to transform data into coherent views and check if there are violations in the CI/CD recommended workflows. The first observation of our experiments is the deceiving perception of a CI/CD pipeline as a neat succession of events, i.e., a new release will proceed smoothly to each next expected stage, from build to production. A *posteriori* one can argue that this is evident since each stage in a pipeline may fail. Then, teams may perform actions that will fix the underlying reasons causing the failure, and the developers may trigger a new build (possibly at a non-expected next stage).

This paper is organized as follows: in the next section, we briefly present process mining and how it can be applied to extract workflows, check conformance and support process improvements in software development activities. In the subsequent, we present an overview of Kafka to highlight its capabilities to support the preliminary data integration and normalization required by PM algorithms followed by which we present the Kafka-based framework that can support the PM activities. This is the core contribution of this paper, as we highlight the synergies derived from using the two technologies together. Then we present a detailed description of an actual implementation of the proposed architecture. Next, we present some insights gathered from such implementation in the context of the software development unit of a major retail (e-commerce) European company operating in the global market. Before the final section, we compare and relate our proposal with other works described in the literature. Finally, we conclude the paper by presenting the main observations and ongoing work.

Process Mining Overview

Process mining emerged as a research field that analyses processes using real-world data. It focuses on end-to-end processes and is only possible because of the growing availability of event data, increased computing capability, and an extensive set of process discovery and conformance checking algorithms. Process mining aims to discover, monitor, and improve real processes by extracting knowledge from the enormous amount of data collected by today's information systems [6].

The starting point for any process mining activity is an event log. Each event in such a log relates to an activity (e.g., build, quality assurance, user acceptance testing steps in the software development process) related to a particular case (e.g., a user story, a software component or a defect ticket). The set of events involving a case are

[2] https://kafka.apache.org/.

[3] https://www.jenkins.io/.

[4] https://circleci.com/.

ordered and can be seen as one specific execution of the underlying process. Event logs may contain additional information about events such as the resource (e.g. human or script) that initiated the event, the cyclomatic complexity of the software component, or any other data elements associated with a specific event (e.g., the error ID of a failed build).

Event logs can be used to conduct one of the three major types of process mining activities: discovery, conformance, and enhancement. Discovery involves using the event log to reverse-engineer the actual process model without using any other a priori knowledge. Conformance can be used to check if reality, as recorded in the log, conforms to the expected workflow and vice versa. Enhancement seeks to improve an existing process using information about the actual process recorded in the event log. Examples include identifying throughput times and bottlenecks.

The unique value of process mining comes from being based on hard facts: the event logs reflect the actual system behaviour, thus providing an evidence-based approach for getting insights into the actual dynamics of systems. There is intense research on techniques for automated process discovery [12–14]. Event log noise, errors, and incompleteness raise significant problems to the discovery algorithms, some of which have been addressed by genetic mining [15], fuzzy mining [16], and heuristic mining [17].

There are a set of competing quality attributes that can be used to select the algorithm that best reflects the process embedded in an event log namely *fitness* (how well the log is reflected in the extracted model), *simplicity* (whether the model is the simplest possible representation), *generalization* (how well the model is able to represent reasonable new events not present in the log, but not so much that any behaviour is possible), and *precision* (not underfitting the event log) [6]. In theory any process discovery algorithm can be used, and a weighed evaluation of the criteria can be used to select the model that best reflects the log.

Of particular interest to us is the heuristic miner that we observed to be one that best extracts the activities in a CI/CD pipeline. This algorithm starts by building a dependency graph based on the frequencies of activities and the number of times each activity is followed by another activity. Dependencies are then added to the dependency graph based on predefined (tunable) thresholds. This dependency graph embeds the core structure of the underlying process model. This initial structure is then used to discover the detailed split (multiple output arcs) and join (converging arcs) behavior of nodes using the original log.

To apply the above-described heuristics, it is necessary to collect the development-related events from multiple heterogeneous tools and normalize these event logs before feeding them into the process mining algorithms. This is where Kafka can be invaluable.

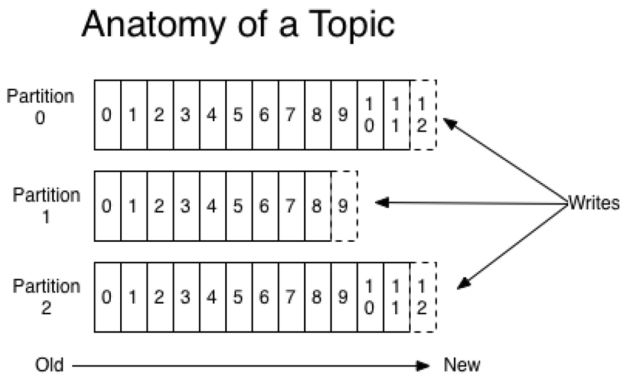

Fig. 1 Anatomy of a topic (Extracted from https://kafka.apache.org/intro)

Kafka Streams

Apache Kafka is an open-source and high-throughput distributed streaming platform. It was developed to replace typical messaging queues and/or enterprise service bus systems unable to cope with demanding data usage needs.

Kafka runs as a cluster that stores *streams* of records grouped in categories designated *topics*. A topic is normally associated with a particular stream of data, similarly to a table in a database, but without the constraints since the same topic may contain records with different formats. As depicted in Fig. 1, a topic inside a Kafka cluster is maintained as a partitioned log. A *partition* is a structured commit log in which records are continuously appended in an ordered and immutable sequence of records. Inside a partition, each record is assigned a sequential id, the *offset*, which uniquely identifies a record inside that partition. The offset allows consumers to unequivocally identify the last message read. A record—also referred as *message*—is defined by a *key/value* pair and a *timpestamp*. The *key* is typically used for partitioning, which allows that all messages associated with a key are sent to the same partition. If no key is defined, Kafka will distribute the messages by the available partitions. The key is valuable if the order is important; for example, an order delivery message should be read by a consumer after the order entry message. Finally, the *value* contains the message itself.

Kafka has five core APIs supporting different types of usage. The *Producer API* supports the *publishing* of data streams to a Kafka topic, while the *Consumer API* allows the *subscription* to one or more topics. A more advanced usage is supported by the *Streams API*, a streams processor, which allows a stream to be processed and used as

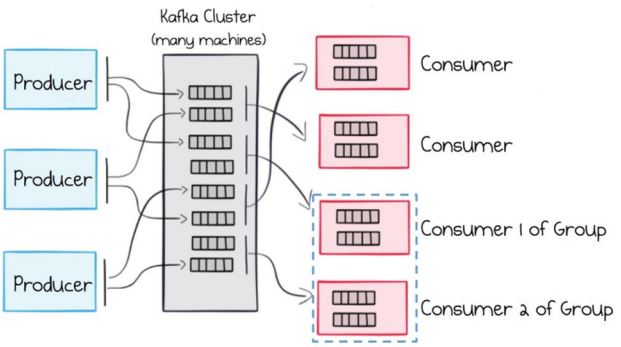

Fig. 2 Producers and consumers in a Kafka framework. (Extracted from https://de.confluent.io/blog/apache-kafka-for-service-architectures/)

input to an output stream. The *Connector API* provides means to connect Kafka topics to existing applications or data systems, e.g., changes done to a table in a relational database can originate a stream of data to be recorded in a topic. Finally, the *Admin API* allows the management and inspection of Kafka objects.

As depicted in Fig. 2, the core roles in such data-oriented architecture include *producers*, any type of entity that creates data (e.g., compilers, automated version control, integration and deployment managers, auditing and monitoring software, and scripts), and *consumers*, any component that ingests the data stored in the Kafka cluster. Consumers of the same information might have different processing behaviours, e.g., some may read development-related data in real-time while others may read the same data only once a day. Moreover, goals will vary since the data treated by consumers can be used as input to generate dashboards and reports, to feed other tools or alerts to provide near real-time visibility about the workflow, namely undesirable process deviations (e.g. a component failing a performance test was sent to deployment anyway).

The relevant point is that, by adopting this stream-based architecture, the producers and consumers are completely decoupled, which provides significant benefits:

- the producers' design is independent of the type of consumers that will be connected;
- slow consumers will not impact the producers' ability to create new events;
- the failure of one or more consumers does not impact the whole system; in fact, replicated consumers can be used to increase the system's dependability.
- the number of producers and consumers is elastic, i.e., we can add consumers without impacting the producers and vice versa.

Apache Kafka can be deployed as a distributed solution running as an independent cluster; therefore, it is easily adaptable to redesign any enterprise architecture and to be used as the single source of truth connecting every software/system component to every event produced. It can be extended with persistent storage and provide historical views of events by allowing users to configure suitable data retention periods for their applications.

All these characteristics, namely (1) its intrinsic decoupling from the observed system, and (2) the capability to transform changes done to a table in a relational database into a stream of data recorded in a topic, motivated our research to explore the potential of Kafka as the foundation of an integrated process mining framework to support tracking of software development activities in a DevOps environment.

A Kafka-Based Process Mining Framework

Regardless of the size of a software development team, there is always an array of tools continuously generating data about the development activities that can provide useful insights on the development effectiveness and quality. One of the most valuable perspectives is to automatically extract best practices and process-related information directly from the development tools rather than explicitly require developers to describe or report their activities. For example, in test-driven development environments, it would be valuable to verify whether unit tests are actually defined and run before the associated code is written. While this kind of information can be extracted from the project repository, process mining algorithms can not only transform this raw data into coherent workflow views, but also check for conformance violations according to a predefined workflow.

A significant challenge of analyzing the data from different tools relates to its normalization into a standard format and merging this data in a common platform. A straightforward example is when different versions of the same tool are used, which generates slightly different data outputs, requiring any custom dashboard connected to one version to be reviewed to integrate the new data formats imposed by a new version. Moreover, different tools can be used for similar goals, such as Jenkins or CirleCI integration orchestrators, but a unified view of the integration pipeline is desired.

The adoption of Kafka as the integration framework addresses these problems by providing means to access normalized data views and the easy integration of new data sources. In Fig. 3, we present an overview of the proposed framework. At the left, a heterogeneous set of tools generate data related to software development. The data from these tools are fed into the Kafka cluster consumers through dedicated connectors using Kafka API libraries

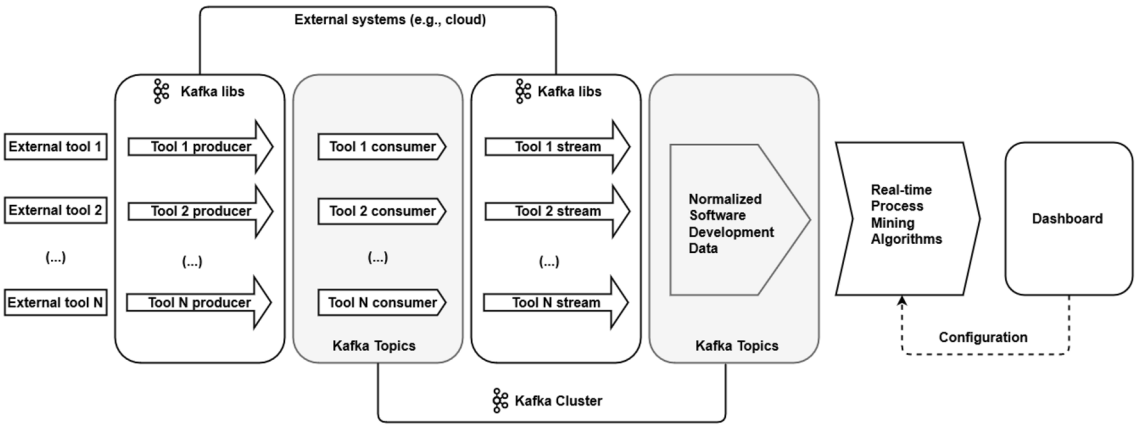

Fig. 3 A process mining reference framework using Apache Kafka

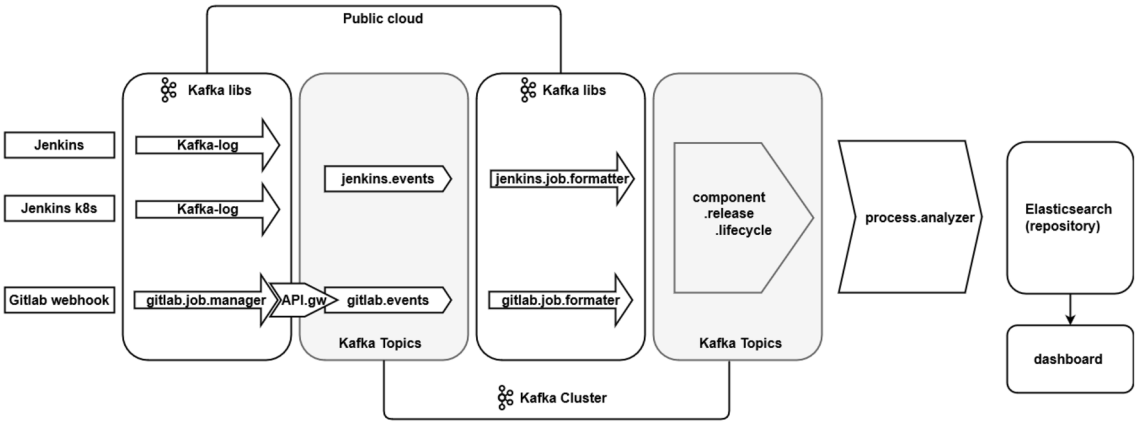

Fig. 4 An actual process mining framework using Kafka

(the producers). The consumers are Kafka topics, and they can be viewed as records of the events occurring in the external tools. Kafka streams then process these topics. At this stage, the relevant events are selected and fed back into the Kafka cluster to be stored into (normalized) topics with software development data. Note that at this stage, data from different tools (and different tool versions) are aggregated and normalized, ready to be fed into the process mining algorithms.

After analysis, the event data are displayed in a dashboard. This dashboard must also provide a configuration panel to customize the process mining activities (which algorithm to use, operation modes —discovery, monitoring, analysis—data selection and filtering, conformance thresholds, among others).

In the next Section, we describe a concrete implementation of this architecture, used to monitor the actual processes behind a complex CI/CD pipeline used by the ~ 300 staff software development division of a major retail (e-commerce) European company operating in the global market. This e-commerce platform must fulfil five-nines business availability requirements (5.26 minutes of downtime a year).

A Real-World Implementation

The initial motivation for implementing the proposed architecture was our need to gain visibility about the CI/CD deployments happening in the development ecosystem. Different teams were aware of the deployments in their own scope, but they lacked a common view of all the deployments from all teams and a human-friendly approach to display the data. The upstream activities (before the build) are not addressed in this paper, as our focus is on the CI/CD pipeline from the initial build to the "live" deployment.

```
18:16:47 [JaCoCo plugin] Done.
18:16:47 [JaCoCo plugin] Overall coverage: class: 0, method:
18:16:47 [INFO] Triggering a new build of kafka-log...
18:16:47 No emails were triggered.
18:16:47 [WS-CLEANUP] Deleting project workspace...[WS-CLEAN
18:16:47 Warning: you have no plugins providing access contr
18:16:47 Finished: SUCCESS
```

Fig. 5 Jenkins output console with step that sends information about the job into Kafka

The actual architecture is presented in Fig. 4 and is described below.

Event Originators

Builds are automatically triggered when a developer commits code into the software repository. At first, only one instance of a Jenkins server was plugged into this system, a regular *on-premise* solution with no containerization. Therefore, the authors implemented a custom component (*Kafka-log*) in the Jenkins jobs (Fig. 5), using the Kafka *producer* API, responsible for sending the job's execution data into the Kafka cluster to the topic *jenkins.events*.

The logs sent to Kafka correspond to data also accessible via Jenkins APIs (Fig. 6), which provide information about the actions executed, the users involved, the timestamps, and outputs from the Jenkins job.

As part of the digital transformation happening in the company, new business components started to be deployed in containers instead of the typical Tomcat web servers. A first approach was to use an on-premise cluster managed by the Kubernetes[5] (k8s) container-orchestration system. This shift into a k8s ecosystem required the creation of a new Jenkins instance. Even though the new instance was different, we instantiated the same *Kafka-log* component, thus allowing jobs' execution data to be sent into the same *jenkins.events* topic. Figure 7 displays the content of the Jenkins' topic using the open-source tool AKHQ.[6] This consumer now treated information from two sources and sent it into the same repository without impacting the configured dashboards.

The transition into a public cloud solution, using containers also managed by k8s, promoted the usage of Gitlab as a new source version control and the CircleCI's CI/CD pipelines. The Kafka infrastructure allowed us to stream data from these new pipelines into a new topic, *gitlab.events* (see Fig. 4). The process of retrieving information from this new pipeline is slightly more complex since it evolves a Webhook configuration that pushes information into a local microservice, the *gitlab.job.manager*. This service is a Kafka producer since it is pushing records into the topic *gitlab.events*. For security reasons, we had to implement an *API-Gateway* working as a middleware layer between the Webhook and the Kafka producer.

The simplicity of the evolution from a single Jenkins pipeline to three different pipelines using different technologies running in different platforms was our first major evidence of how effective Kafka could be as the backbone of the data normalization step for the process mining analysis.

Streaming Software Development Information

To handle the data originating from each consumer (*jenkins.events* and *gitlab.events*) we implemented a dedicated streaming component for each one: *jenkins.job.formatter* and *gitlab.job.formatter*. These streaming components are consumer/producers since they are responsible for consuming data from the corresponding topic, selecting the relevant fields, processing it if necessary (e.g. normalizing

Fig. 6 Details from a Jenkins job retrieved by its REST APIs

[5] https://kubernetes.io/.

[6] https://akhq.io/.

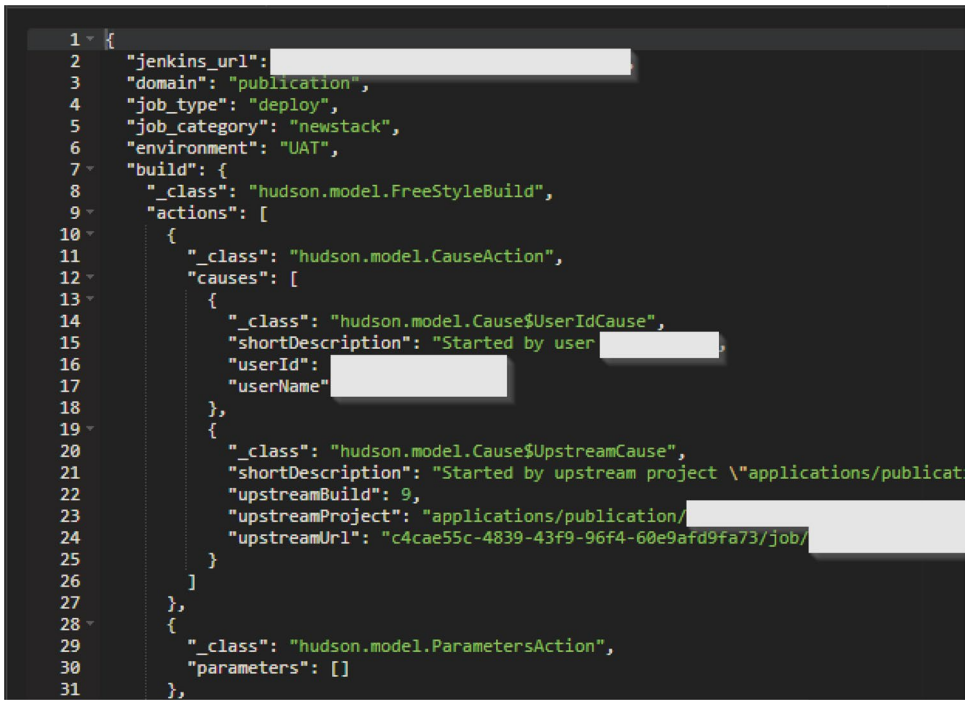

Fig. 7 Visualization of a message containing data from a Jenkins job in AKHQ

data formats), and sending this standardized data into into a process-oriented Kafka topic, which has been named *component.release.lifecycle*.

As mentioned before, when producers define a key, Kafka sends all messages into the same partition, thus providing order. The records inside the standard *component.release.lifecycle* will have specified a key, the <*component_name*>. This key defines each Java component's name and is unique inside the enterprise's software development division. Therefore all logs of events related to a specific software component are aggregated in the same Kafka topic, regardless of its origin.

The data in the *component.release.lifecycle* topic were integrated and normalised. From here, the information in the standard format is now available for any entity that needs to process it, namely the *process.analyser*, where we have configured the process mining algorithms.

Process Mining Analyser and Alerting

The *process.analyser* is a SpringBoot[7] application that we have deployed in a public cloud, following the company's standard development practices and sharing its current infrastructure. We thus configured the *process.analyser* to retrieve messages that arrive into the stream of data from *component.release.lifecycle*. This stream of data is configurable, and if desired, it is possible to configure which messages to read (e.g., all messages, Jenkins' only or Gitlab's only). The *process.analyser* is responsible for executing the process mining algorithms and publishing the results into the dashboard. As a first approach, the PM results are stored in an Elasticsearch[8] repository. Our goal is to feed the PM algorithms directly with data arriving in the Kafka topic to provide the workflows and alerts in real-time.

This architecture's flexibility paves the way to easily include new streams of data (in Kafka) that contain the process mining results, thus allowing other consumers to have access to this information and use it for novel insights.

The following section presents some outputs from the implemented architecture and a few gained insights into the CI/CD pipeline behaviour.

The Observability Ecosystem and Insights

In this section, we present the implementation described above to illustrate its concrete application and the insights gathered. To grasp the complexity of the technology stack and its environment, suffice to say that components are developed in various technologies: legacy programs in COBOL or Mainframe, Java batches, SOAP web-services, REST APIs, web applications developed in different technologies, and newer applications as micro-services, reflecting

[7] https://spring.io/projects/spring-boot.

[8] https://www.elastic.co/.

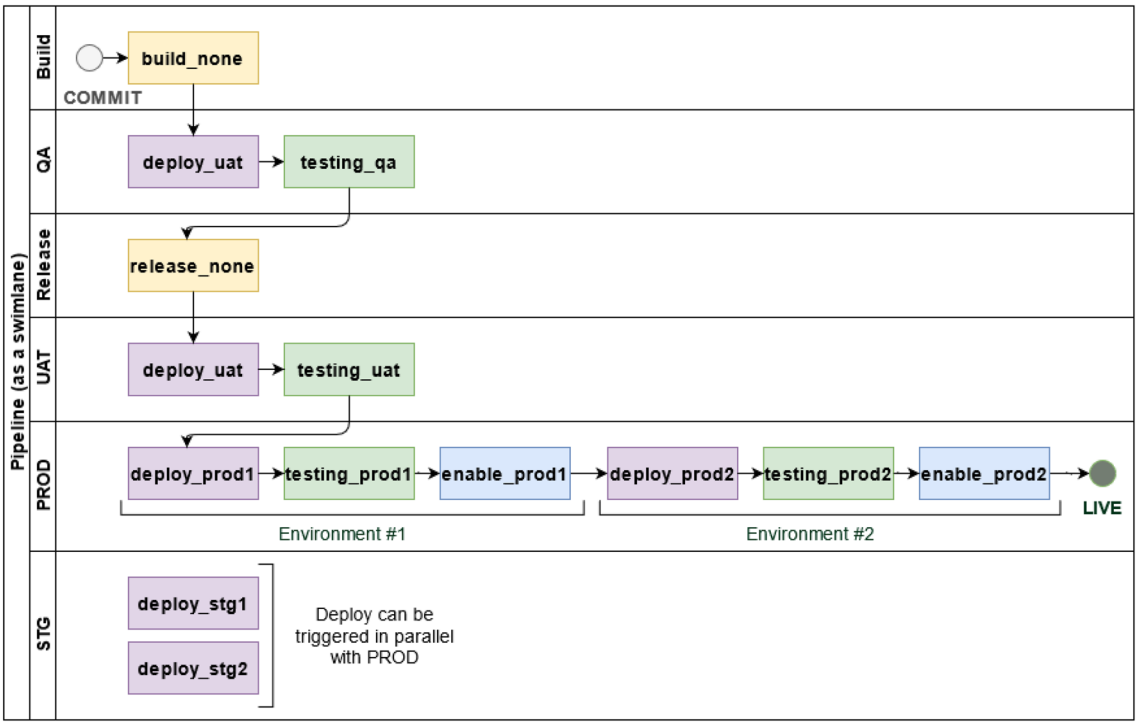

Fig. 8 Representation of the Jenkins CI/CD pipeline monitored. Colour scheme: yellow for the steps without environment; purple for deployment steps, green for the testing steps; blue for "enable" steps implemented only in PROD

an organic evolution from the company's first systems initially deployed in the 80s. Before going into detail in the case study itself, we present a quick overview of the observability ecosystem in the company.

Observability: Logs, Metrics and Traces

The tools and frameworks used for observability are varied; the deployment of these tools follows the development stack to fit the requirements from new solutions and architectures.

- **Logs** are collected from the infrastructure using Filebeat[9] agents and made available in Elasticsearch instances, which in turn are data sources for dashboards developed using Grafana[10], Kibana[11] or PowerBI[12] (used by business users). These logs – business and technical – allow teams to debug and troubleshoot issues on systems. Events from CI/CD pipelines are also part of the data collected and made available. The Operations teams implemented an additional Kafka layer – between the logs' sources and the Elasticsearch instances – to ensure that there was no loss of information (logs).
- **Metrics** the monitoring team implemented Prometheus[13] and Thanos[14] to store, visualize, and query metrics. Metrics are also exploitable with the same dashboarding tools and are used for alerting purposes. Anomaly detection using time series metrics are in the roadmap for future work.
- **Traces** tracing capabilities are implemented using Dynatrace[15] for Application Performance Monitoring (APM) in a specific scope of the IT system; and Istio[16] and Jaeger[17] for distributed tracing in the micro-services architecture.

In terms of alerting, the monitoring team uses a combination of tools that includes AlertManager[18] (from Prometheus), Centreon[19], Nagios[20], and Thruk.[21]

[9] https://www.elastic.co/beats/filebeat.
[10] https://grafana.com/.
[11] https://www.elastic.co/kibana/.
[12] https://powerbi.microsoft.com/en-us/.
[13] https://prometheus.io/.
[14] https://thanos.io/.
[15] https://www.dynatrace.com/.
[16] https://istio.io/latest/about/service-mesh/.
[17] https://www.jaegertracing.io/.
[18] https://prometheus.io/docs/alerting/latest/alertmanager/.
[19] https://www.centreon.com/en/.
[20] https://www.nagios.org/.
[21] https://www.thruk.org/.

The CI/CD Pipeline Being Monitored

The input data (event logs) to be collected and analysed come from different sources along the CI/CD pipeline, represented by Fig. 8. New developments and maintenance (changes) to existing components are developed and unit-tested on developers' workstations or shared infrastructures. When a developer commits code on the version control system (SVN[22]), the commit is listened to by Jenkins, which initiates the build of the component into the deployment pipeline.

This pipeline encloses three non-production environments—Quality Assurance (QA), User Acceptance Test (UAT), and Staging (STG), followed by two Production environments (PROD1 and PROD2). After the activation in both production environments, the software is considered fully deployed (LIVE).

Immediately after the build, the component is submitted to SonarQube[23], a static analysis tool that reports several quality metrics (e.g., technical debt, test coverage, complexity) and enforces specific rules enclosed in a predefined quality profile. If the build and static validation are successful, then the component is automatically deployed (e.g. into a web server or a batch server) in the first non-production environment (QA).

Successful deployment triggers the test campaign via Cerberus [18], an automated testing tool (which embeds Selenium[24]) that ensures non-regression validation for usage scenarios specific to the company's context. In this QA environment, interface testing is performed, ensuring that the new code has no impact on the existing functionalities (regression testing). The system's vital functionalities (new and existing ones) are then tested. In this environment, developers can fully manipulate the system (e.g., configurations and data) to run tests in components, expected to be more bug-prone if we compare them to the next stage (UAT). It ensures that developers can test on a non-production environment, different from their local machines, immediately after the commit.

The campaign execution outputs a result (OK or KO) used by the teams to decide if the component is ready to be installed in the next pipeline stage, the deployment in UAT. All steps that lead to the deployment in QA are fully automated; however, the deployment into UAT and *live* requires a manual step ("click GO to deploy") ensuring that human validation occurs.

UAT is the environment where the users—or their proxies—test the solution, in line with what would occur in real-life scenarios. It is a more restricted environment, given that the components deployed here are stable versions that may be installed in production. Hence, teams do not have free access to the environment's configurations; despite being a non-production environment, UAT is very similar to the actual production environment; this is ensured, e.g., by the frequent refresh of the databases with production data. In addition to users manual tests, in UAT, automated functional and end-to-end campaigns are executed using Cerberus.

STG is a final environment for testing that resemble an actual production environment as closely as possible and may connect to other production services and data (such as databases). In this environment, automated load, stress, and performance tests are performed. Other installation/configuration/migration scripts and procedures are executed before the new version is deployed live. It is also here that new features or validation integration with live versions of external dependencies are tested. The types of tests performed in STG ensure that major and minor upgrades to the live environment are completed reliably, without errors, and in a minimum of time.

The application is deployed sequentially in each of the two production environments (PROD1 and PROD2). After being deployed successfully in the two production environments, the new software version is considered *live* to the end-users.

Kafka Data Collection and Normalization

The specific input sources fed into Kafka to be consumed by the process mining algorithms are originally provided by Jenkins logs (Fig. 5). These logs include the time elapsed between deployment in different stages for each software component committed into the CI/CD pipeline and the person who did it or whether it was an automated process.

We programmed the Kafka consumer to collect the following information for each pipeline stage:

- Component_Name—the name of the java component to which the pipeline job belongs.
- Pipeline_Stage—combination of the type of job with the environment where it is being performed, e.g., *build_none* or *deploy_qa*.
- Timestamp_entry—job start timestamp.
- Timestamp_exit—job end timestamp.
- Owner_author—who triggered the job, it can be a person or an upstream job.

Other information is also collected, for example, the internal business unit that developed the component and the repository link to the component version. However, these are used for different purposes, e.g. diagnostics.

[22] https://subversion.apache.org/.

[23] https://www.sonarqube.org/.

[24] https://www.selenium.dev/.

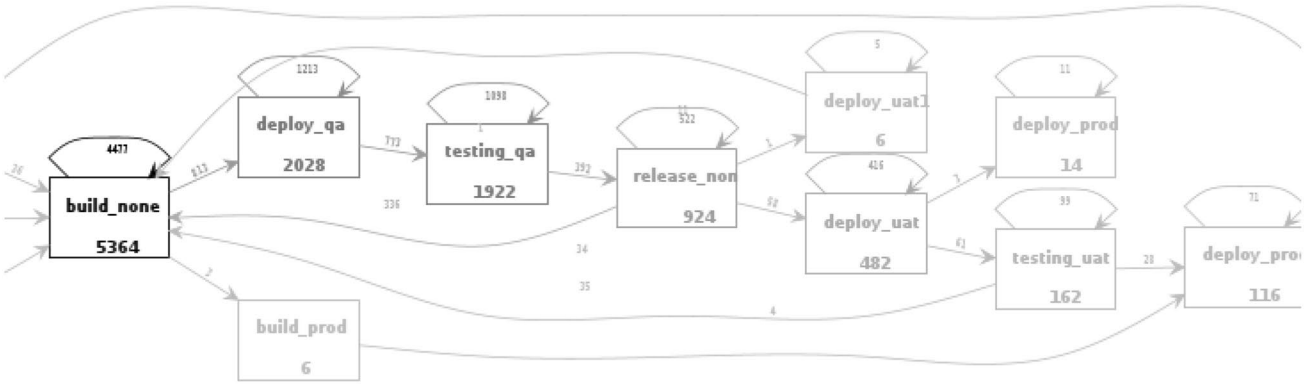

Fig. 9 A partial view of the component workflow on the CI/CD pipeline generated by a PM algorithm. Each box represents a stage, each box contains the number of submissions/executions for that stage, and the arrows represent the number of transitions between stages

These fields are ready to be used by the process mining algorithms without any further processing, as intended.

Observations and Insights

The analysis presented in this paper concerns only the CI/CD pipeline usage from January to May 2020 and fed it into the heuristics miner [19] algorithm. A partial view of the workflow extracted is depicted in Fig. 9.

Even with this partial view of the workflow generated by the PM algorithm, the first most evident observation is the "deceiving perception" of a CI/CD pipeline as a clean and ordered sequence of events. A *posteriori* this is obvious, as components' integration can abort at any pipeline stage, developers can reconfigure the test environment at every abort stage and relaunch the build process at that stage, and even manually skip testing stages. Also, we can observe the heavy usage of the initial build stage (5.364 submissions) if compared with other pipeline stages. We could also observe six occurrences where the software component submitted was sent directly to 'build-prod', bypassing the initial quality assurance and user acceptance testing. This observation is an example of behaviour that should have raised an alert to QA management. The frequency of activation of each stage is also valuable insight to be used by project managers and the operations team to adequately size the computational resources associated with each stage.

From the output of the PM algorithm, we extrapolate developers' behaviours and pinpoint suggestions for improvement. A first point to consider is the differences among frequencies of execution. The *build* stage in a pipeline is the most executed (build_none=5364), followed by the deploy in QA (deploy_QA=2028), and the testing in QA (testing_QA=1922). In contrast, the stages related to UAT and PROD environments do not have the same amount of occurrences. Since deployment and testing in QA are automated stages (no manual action to trigger them), it is reasonable that the frequency is also high. From our perspective, managers and developers can review these differences by asking themselves:

- Are developers using the build stage to compile and visualise the output from Sonar analysis? Developers should execute the Sonar plugin locally in their IDEs and commit the feature only when ready. The high frequency of the build can signal that developers rely on the build performed by Jenkins to compile and validate Sonar rules.
- Are developers using *trunk-based development*, and the number of commits is related to that (as a way to ensure frequent commits)? If this is the case, maybe it will be worth creating specific feature branches that allow frequent commits and, in turn, do not pollute the *trunk*. When developers finish a feature, and it is ready to be integrated, the developers should merge the feature branch into the *trunk*. The same rule applies to code versioned with Git.
- Are teams developing and committing new features without waiting for the corresponding deployments in UAT and PROD? If yes, this can also justify the high number of submissions in QA compared to other environments. Managers need to assess if this is the case, promote the quicker deployment of features, and ideally have each feature moving independently from QA to PROD. For example, different reasons can justify these delays, such as the unavailability of stakeholders to test the applications, freeze periods caused by commercial operations (e.g., Black Friday), among others. In any case, teams should consider implementing, for example, feature toggles to allow quicker deployments and posterior activation of features when other dependencies are ready.

A second observation extracted from the workflow generated by PM (Fig. 9) is the existence of stages not defined in the reference model (Fig. 8) such as *build_prod* and

deploy_uat1. By looking at the frequency of execution, these are sporadic events that we managed to confirm with the "domain" teams. The *deploy_uat1* was a stage misconfigured at some point in time. A specific team uses the *build_prod* stage for a specific project, and its purpose is to archive the last version deployed in PROD before executing a new deployment. Thus, this stage is an exception, not a standard practice. Therefore, the execution of process mining tools not only revealed unknown (anomalous) practices, but also highlighted potential improvements in the configuration of a few Jenkins pipelines, namely

- The Pipeline_Stage was misconfigured for a few jobs—for example, instead of having the **testing**_*qa* as the description of the job, we had **qa**_*testing*, and this was revealed by the process mining visualization since we saw that we had an unexpected "activity" in our process flow.
- Some jobs didn't have the property "domain"—this property identifies the functional domain of the pipeline, e.g., logistic, finance, or technical (jobs that deploy components related to the infrastructure). We observed this when analyzing some specific events, and we drilled down the data and observed that the "domain" was not available.
- When we tried to use the property *build_id* as the *Case ID*, we observed that some jobs had the property empty. This was not supposed to happen.

For the Jenkins pipelines, the Engineering teams require DevOps engineers to do the fixes. In contrast, for Gitlab/Git pipelines, the developers are autonomous since they can configure the *.gitlab-ci.yml* file. Before setting up this framework, such changes were mostly unknown even by team leads.

From these simple observations on the pipeline's dynamic behaviour, it is manifest that the adoption of process mining provides the software development teams with actionable data to improve the relevant KPIs impacting the effectiveness of their CI/CD pipeline, namely to reduce the overall lead time.

A significant advantage of using real-time analysis of the process is triggering alerts immediately in cases where an abnormal stage appears in their flow, or a flow is missing (e.g., a testing stage skipped). Such real-time alerts allow managers to provide quick and factual feedback to their teams instead of relying on a batch checking the status every *x* hours or days.

Some lessons learned that might be useful to other teams include

- Setup the developers' environment to reduce the load on the integration servers; if local configurations (e.g. sonar) are an issue, automate the import of standard configurations.
- Decide on adopting trunk-based or feature-based development, but be consistent. Automate checks for compliance.
- Minimize human bottlenecks in the pipeline; consider using feature toggles as a standard practice.
- Standardize pipeline configurations organization-wide; if deviations are justified, formalize them, do not fight people's efforts trying to do their job.
- If people can change the pipeline, they will. Create alerts to know about that; understand before judging, someone may know something you don't.
- Pipeline configurations are code too. Embrace a TDD mindset and create automated checks for valid/complete configurations as part of the pipeline setup.
- There is a social network embedded in the pipeline (PM can easily extract this network). Check for unexpected people performing unexpected roles (e.g. integration gatekeepers) and look for the underlying reasons.
- Keep watching your CI/CD pipeline: abnormal behaviour is an early warning of latent issues in development.

Related Work

This section discusses the related work focusing on the process mining techniques linked to our research, continuous practices and approaches for monitoring such continuous practices implemented using CI/CD pipelines. The research presented in this paper looks at the data provided by CI/CD tools from a process-centric perspective. The aim is to apply PM techniques that provide insights and visualization capabilities for our CI/CD processes. Even though we rely upon some metrics commonly used by other tools such as time information, our approach treats an execution of a pipeline as a process that should follow certain stages from the commit up to production. Flows that do not match the reference model (i.e. the expected set of execution stages) reflect misconfiguration of the process, wrong usage of the pipelines or even the need to improve the current process.

Process Mining

Process mining is a group of techniques in process management that supports the analysis of business processes. It focuses on process discovery, conformance checking, and process enhancement [6]. Process discovery aims at discovering the underlying processes using as input the event logs collected from digital systems. There is a large body of process discovery algorithms [12, 14, 20, 21].

Despite all the intense work on process mining's core issues, all these algorithms are very dependent on the

preliminary steps of data collection and normalization. This data is commonly stored in relational databases, thus not fit to be input directly into the process mining algorithms since each event of the process must be related to precisely one case by a case id. Therefore, it often requires a tedious manual or semi-automatic analysis of the input sources, highly dependent on the data analyst's expertise.

The first problem is cleaning the event logs [22, 23]. Solutions for specific areas have been addressed, e.g. in healthcare [24], and much research has been done to devise more general solutions [25–28]. These approaches focus mainly on extracting data present in multiple relational databases and converting it into a relevant set of timed events that can be fed into the process mining algorithms. *We take a completely different approach by moving what can be described as an Extract-Transfer-Load (ETL) problem into the architecture realm*. Since in Kafka streams, the relational nature of data is hidden under the stream abstraction, we can simply plug-in a data consumer according to the needs of the process mining algorithms. This approach is not only more general, but also completely domain agnostic; it is only constrained by the need to adopt Kafka pipelines. However, if the application architecture already uses this widely adopted open-source technology, plugging-in a process miner Kafka consumer is a trivial task.

Continuous Practices

In their literature review, Rodrigues et al. [29] concluded that research on continuous practices focused mainly on continuous testing and QA practices, continuous and quick releases, automation, and on lean and agile software development. Less frequent were studies focusing on architecture, experimentation and R&D, and customer involvement. Senapathi et al. [30] identified two streams in this research topic, one that focuses on the concept of DevOps and its inherent characteristics, and the other that focuses on the benefits and challenges of adopting continuous practices and how real teams are implementing them. Most of the literature mentioning the word *monitoring* are related to code, features deployment, and users behaviour [8]. The pipelines monitoring is usually addressed in a security perspective [31], or by analysing the adoption and related metrics [8, 30, 32]. Moreover, Whittingham [33] and Fedeczko [34] reflect on the concepts of observability and monitoring, its relationship with CI/CD and how pipelines should integrate it, and the importance of collecting metrics and having actionable insights.

In our perspective, the advent of self-service pipelines is somehow a positive disruption since development teams can implement their pipelines. Therefore we believe that real-time pipeline monitoring will be a common practice and new approaches shall be required. Rodrigues et al. [29] also emphasised, lined up with other studies, that any software domain can benefit from continuous practices. The advantages have been proven in the industry, including short and continuous feedback loops, continuous testing, and fast time-to-market, ultimately increasing customer satisfaction and business value. Automation is well known for improving product quality and reliability, but it also positively impacts developers' productivity, whatever their level of experience. With a reliable and automated means of analysing, testing and deploying software, newcomers become more confident and more productive earlier [8]. The domain of expertise and the time of the year are also variants for CI practices, e.g. CI can be suspended during specific commerce periods such as the Black Friday [35].

Observability

Logs and metrics produced by CI/CD tools have been used by teams to understand what is happening, to identify root causes of CI failures and debug [36–39], and to have visibility of tests executions [8]. Fedeczko [34] pinpoints some advantages of applying continuous monitoring for CI/CD pipelines such as long-term trends analysis, over-time comparison, vulnerability scans, and alerts.

As mentioned in "The Observability Ecosystem and Insights", the company uses different tools and frameworks to ensure the observability of its systems. With this in mind, we wanted to leverage the existing stack —Kafka system—to handle the event logs produced by the different CI/CD pipelines. Implementing new solutions, especially commercial solutions, would require a validation period and acceptance (or not) by the responsible teams in charge of the costs and decision-making. The usage of the same infrastructure available for business processes allowed us to have the resources in place quickly and develop the custom components to support our needs. Our Operations teams validated this solution.

Currently, different commercial solutions can handle large amounts of data, allowing users to create dashboards and alert rules, following a real-time approach. Examples of these solutions include Splunk[25]—also used by Neely and Stolt [8])—and Datadog[26]— a well-known solution for monitoring. The major difference between these commercial solutions and the one presented in this paper is that we want to take a process-centric view of what is happening in our pipelines. PM algorithms and visualization tools support this approach.

From this perspective, in this section, we highlight industry case studies that focus on the monitoring, observability, and modelling applied to CI/CD pipelines.

[25] https://www.splunk.com/.

[26] https://www.datadoghq.com/.

Neely and Stolt [8] describe their journey to adopt continuous delivery, moving from deploying in production every eight weeks to deploy *at will*. This work is pertinent to mention since their experience pinpoints ideas and actions that can be addressed and supported by process mining. Four golden rules are proposed to readers when starting continuous delivery: *prune manual steps*, *invest on tests*, *mimic production environment*, and *business should embrace feature toggles*. To succeed in the first golden rule—*prune manual steps*—it is essential to know our processes; this should not be an issue for DevOps teams, but it might not be the case for business teams. The authors recommend Value Stream Mapping (VSM) [40] to construct a process map while gathering everyone in a room and then having better visibility of the system and discovering optimization areas.

PAFnow [41] presents PM as a progression from traditional Value Stream analysis since VSM is not an effective method for handling numerous variants. Moreover, gathering people from different teams in one event may not be feasible. Discovery PM techniques that use process-centric data retrieved from the IT systems can help teams automatically construct their value stream maps, identify bottlenecks, and the optimization potentials. Combinations of value stream mapping with process mining were reported in the context of Industry 4.0 [42, 43].

Invest in our tests, the second rule from Neely and Stolt [8], tell us to make our tests "fast, solid, and reliable". This topic focuses on two principles: *(1) accelerate test executions* and *(2) "destroy flaky tests"* [41]. If test executions are too long, e.g., hours, it will create long feedback loops and frustration on the teams. Ignoring flaky tests by re-running the pipeline step that executes the test campaign is not a good solution since we may bypass a real problem that will be revealed sooner or later in production. For both *rules*, PM analysis and visualization techniques can be invaluable. First, it is possible to check which test steps are taking too long, and by doing a drill down on the process, it is possible to spot a possible underlying root cause: functional scope, type of project or even the team involved. Finally, PM will show if people are re-running the test steps to reach a successful test execution instead of fixing the root cause. The loops on a test step will inform managers that there is improvement potential, either in the test campaign or on the team's methodology.

Feature toggles is another point mentioned by Neely and Stolt [8]; it allows teams to deploy faster to production, reducing the lead time. Moreover, business teams will be able to activate features when everything is ready. PM techniques allow teams to communicate the process more clearly and transparently. Teams can look to Graphs or Petri Nets and see all the phases that belong to a CI/CD pipeline. With the frequent usage of feature flags, the reduction of the lead time will also be observable. Similarly to our work, in [8], the engineering team used metrics from Jenkins to gain visibility of the tests. They developed a Splunk connector that ingests those metrics and custom apps to track the build's health and check flaky tests.

Ståhl and Bosch [44] reviewed different implementations of continuous integration and observed differences deemed acceptable. They propose a descriptive model which addresses all the implementation variants identified in the study. The model includes two parts: (i) the "Integration Flow Anatomy" which defines the activities (or jobs), input nodes (triggers for activities) and relationships; and (ii) the attributes applied to these nodes. The authors focused on the differences among processes, not in tooling as we did; also, context parameters such as project characteristics or functional domains were not considered in their study. For example, the study reports that the source-code compilation step typically precedes a testing step; the type of tests executed varies (e.g., unit, integration, or functional). Testing stages are often combined with static and/or dynamic code analysis. The mentioned practices are also true for our case study. Additionally, for K8s components deployed in public clouds, the new pipelines will include a stage for security analysis using the Clair framework[27]; the goal is to check if security vulnerabilities are detected.

The application of the model [44] started by selecting the project; then, the authors and the team sketched the Integration Flow Anatomy; this was done using a whiteboard and reviewing all the activities configured in their CI server (Jenkins). According to the authors, the exercise itself was educational for team members, and they identified improvement points. It is similar to VSM exercises that aim to discover the process and identify areas of improvement. Regarding the attributes, the activity ones were collected from Jenkins configurations, whereas the input attributes were collected from unstructured interviews. While analyzing Jenkins configurations, the authors observed incorrect configurations not aligned with the team's objectives, which did not match the Integration Flow Anatomy. This situation—the difference between the actual configuration and the team's model—is fully addressed by our approach; hence, it poses itself as valuable to model different CI implementations. By using PM techniques, we can feed actual data to visualize the steps implemented in reality by teams. Some of the node attributes listed match the attributes available in our context and thus are also possible to monitor using PM techniques.

It is also worth referring in this work by Ståhl and Bosch [44] the mention of approaches used to communicate continuous integration failed status. Notifications via email are standard even if the email recipients may vary (last developer, core group of developers or all project team). Other

[27] https://www.redhat.com/en/topics/containers/what-is-clair.

communication platforms include RSS feeds, dedicated web pages or dashboards; a combination of methods can be more effective on the awareness of broken builds, the authors stated. These push notifications present themselves as the typical way of detecting that something is not OK with the CI/CD flow and are available in current platforms. The authors do not mention any study focusing on the concept of monitoring or observability.

Lehtonen et al. [32] proposed a suite of metrics to support continuous delivery and deployment for an actual case study—a web project—of a Finnish Software company; the authors relied on quantitative data and descriptions from the development process and the pipelines.

Those metrics target to help "eliminate waste, and find process improvements", which aligns with our motivation to use PM techniques. Their metrics suite contained two categories: *metrics on the implementation level*—development time, deployment time, activation time, and oldest feature done; and *metrics on the pipeline level*—features per month, releases per month and fastest possible feature lead time. Process mining analysis provides intrinsic visibility of such metrics. For example, using feature branch creation and merge requests as the start and end times for development. Unfortunately, this only applies to our projects in Gitlab/Git. For projects in SVN using *trunk-based* development, these metrics are not directly visible. This constraint agrees with Lehtonen et al. who referred to the tight coupling between data availability and the tools used, the pipelines configurations, and the usage itself.

Senapathi, Buchan and Osman [30] reported an exploratory case study from the industry; their goal was to "monitor and reflect" on the gradual implementation of the DevOps mindset and practices. In the end, the team delivered four times more releases than before, proving the value of the approach. This study highlights the value and importance of monitoring. In their case study, automated delivery processes and CI/CD pipelines emerged naturally as DevOps implementation was their primary goal. *Monitoring* of their releases was also of importance, and hence, the team created dashboards to visualize and share with others the status of their releases in real-time. Dashboards focusing on business monitoring included metrics such as the number of users in the system and the origin of the users. Teams also implemented technical dashboards; each team should have their dashboard targeting the infrastructure. Following a self-service philosophy, each team had to implement dedicated dashboards for their applications. Another goal of the company was to systematically collect metrics to track improvements on lead time and time to recover.

In a different perspective, Koopman [31] highlights the increasing importance of CI/CD, aligned with the increasing concern about security measures to avoid pipeline subversion [45]. Koopman proposes a security framework that allows developers to secure their pipelines systematically. Their work is in line with research that reveals that numerous developers are concerned that their pipelines are disrupted, leading to negative impacts on productivity and time to deliver a release. Any of these steps (or its absence) can be automatically flagged using our approach, as it relies upon evidence collected from the actual pipeline execution.

Conclusion and Ongoing Work

We have shown how the adoption of the Apache Kafka technology can solve a major process mining adoption problem, the collection, filtering, and normalization of the data into well-structured event logs ready to be fed into standard process mining algorithms.

Our next step is to integrate a full real-time, on-line conformance checking algorithm. We are considering several alternatives, specifically [46] whose method applies to any process mining algorithm, namely heuristic mining and fuzzy mining, two of the most common process discovery techniques. The streaming nature of Kafka seems particularly well suited to this type of on-line conformance checking. The goal is to have a dedicated process mining dashboard, and the deviations detected by the algorithm will trigger alerts—in real-time—to the related teams and QA managers. The visibility delivered by such alerts will provide early detection of potential defective components into the live system and shareable insights of situations where "rules should be broken".

Simultaneously, to leverage Kafka streams' full potential, other sources of log information will be added to start covering the whole software component life-cycle. For example, issue and project tracking—Jira[28] —and ticketing and change management—iTop.[29] In our current enterprise environment, we use these two in different contexts: Jira is used to track projects and report bugs detected in UAT, whereas iTop provides the means to report issues and incidents in production. One of the significant challenges of current process mining practices is the merging of new sources of information while ensuring traceability of the events reported with the components being created and deployed in the development pipeline. Using Kafka, that merging will be a straightforward task.

This work requires the involvement of the professionals that rely on this type of visualization to make decisions, including Engineering members such as Team Leads, Tech Leads, and Software Developers. Interviews with DevOps Engineers and Cloud Architectures are also in the backlog to

[28] https://www.atlassian.com/software/jira.

[29] https://www.combodo.com/itop-193.

have their feedback on the actual (not idealized) usage and configuration of the pipeline.

Our end goal is to devise a comprehensive development framework where the insights collected from an integrated adoption of process mining support an effective data-driven software development ecosystem.

Declarations

Conflict of interest The authors declare that they have no conflict of interest.

References

1. Accorsi R, Stocker T. On the exploitation of process mining for security audits: the conformance checking case. In: Ossowski S and Lecca P (eds) Proceedings of the ACM symposium on applied computing, SAC 2012, Riva, Trento, Italy, March 26–30, 2012. 2012;pp. 1709–16, ACM.
2. Giacalone M, Cusatelli C, Santarcangelo V. Big data compliance for innovative clinical models. Big Data Res. 2018;12:35–40.
3. Yang S, Sarcevic A, Farneth RA, Chen S, Ahmed OZ, Marsic I, Burd RS. An approach to automatic process deviation detection in a time-critical clinical process. J Biomed Informatics. 2018;85:155–67.
4. Alizadeh M, Lu X, Fahland D, Zannone N, van der Aalst WMP. Linking data and process perspectives for conformance analysis. Comput Secur. 2018;73:172–93.
5. Rubin VA, Mitsyuk AA, Lomazova IA, van der Aalst WMP. Process mining can be applied to software too!. In: Morisio M, Dybå T and Torchiano M (eds) 2014 ACM-IEEE international symposium on empirical software engineering and measurement, ESEM'14, Torino, Italy, September 18–19, 2014. 2014;pp. 57:1–57:8, ACM.
6. van der Aalst WMP. Process mining—data science in action. 2nd ed. Berlin: Springer; 2016.
7. Rubin VA. A workflow mining approach for deriving software process models, PhD thesis, Germany: University of Paderborn; 2007.
8. Neely S, Stolt S. Continuous delivery? easy! just change everything (well, maybe it is not that easy). In: Proceedings of the 2013 agile conference, AGILE '13, (USA), 2013;p. 121–8, IEEE Computer Society.
9. Mattila A-L, Lehtonen T, Terho H, Mikkonen T, Systä K. Mashing up software issue management, development, and usage data. In: Proceedings of the second international workshop on rapid continuous software engineering. RCoSE '15; 2015. pp. 26–9, IEEE Press.
10. Nogueira AF, Ribeiro JCB, Rela MZ, Craske A. Improving la redoute's CI/CD pipeline and devops processes by applying machine learning techniques. In: Bertolino A, Amaral V, Rupino P and Vieira M (eds) 11th International Conference on the Quality of Information and Communications Technology, QUATIC 2018, Coimbra, Portugal, September 4–7, 2018; 2018. pp. 282–6, IEEE Computer Society.
11. Nogueira AF, Sergeant E, Craske A, Ribeiro JCB, Zenha-Rela MA. Collecting data from continuous practices: an infrastructure to support team development. In: Perkusich A (ed) The 31st international conference on software engineering and knowledge engineering, SEKE 2019, Hotel Tivoli, Lisbon, Portugal, July 10–12, 2019; 2019. pp. 687–777, KSI Research Inc. and Knowledge Systems Institute Graduate School.
12. van der Aalst WMP, Weijters T, Maruster L. Workflow mining: discovering process models from event logs. IEEE Trans Knowl Data Eng. 2004;16(9):1128–42.
13. van der Aalst WMP. Exploring the CSCW spectrum using process mining. Adv Eng Informatics. 2007;21(2):191–9.
14. van der Aalst WMP, Rubin VA, Verbeek HMW, van Dongen BF, Kindler E, Günther CW. Process mining: a two-step approach to balance between underfitting and overfitting. Softw Syst Model. 2010;9(1):87–111.
15. de Medeiros AKA, Weijters AJMM, van der Aalst WMP. Genetic process mining: an experimental evaluation. Data Min Knowl Discov. 2007;14(2):245–304.
16. Günther CW, van der Aalst WMP. Fuzzy mining—adaptive process simplification based on multi-perspective metrics. In: Alonso G, Dadam P and Rosemann M (eds) Business process management, 5th international conference, BPM 2007, Brisbane, Australia, September 24–28, 2007, Proceedings, vol. 4714 of Lecture notes in computer science; 2007. pp. 328–43, Springer.
17. Weijters AJMM, van der Aalst WMP. Rediscovering workflow models from event-based data using little thumb. Integr Comput Aided Eng. 2003;10(2):151–62.
18. Cerberus. Cerberus—an open source, user friendly, automated testing tool. 2011–2021. https://cerberus-testing.com/.
19. Weijters A, Aalst W, van der, De Medeiros A. Alves. Process mining with the HeuristicsMiner algorithm. BETA publicatie: working papers. Technische Universiteit Eindhoven. 2006.
20. Leemans SJJ, Poppe E, Wynn MT. Directly follows-based process mining: exploration and a case study. In: International conference on process mining, ICPM 2019, Aachen, Germany, June 24–26, 2019; 2019. pp. 25–32, IEEE.
21. Ajayi LK, Azeta AA, Owolabi IT, Damilola OO, Chidozie F, Azeta AE, Amosu O. Current trends in workflow mining. J Phys. 2019;1299:012036.
22. Suriadi S, Andrews R, ter Hofstede AHM, Wynn MT. Event log imperfection patterns for process mining: towards a systematic approach to cleaning event logs. Inf Syst. 2017;64:132–50.
23. Nooijen EHJ, van Dongen BF, Fahland D. Automatic discovery of data-centric and artifact-centric processes. In: Rosa ML and Soffer P (eds) Business process management workshops—BPM 2012 international workshops, Tallinn, Estonia, September 3, 2012. Revised papers, vol. 132 of lecture notes in business information processing; 2012. pp. 316–27, Springer.
24. Fox F, Aggarwal VR, Whelton H, Johnson OA. A data quality framework for process mining of electronic health record data. In: IEEE international conference on healthcare informatics, ICHI 2018, New York City, NY, USA, June 4–7, 2018; 2018. pp. 12–21, IEEE Computer Society.
25. Günther CW, van der Aalst WMP. A generic import framework for process event logs. In: Eder J and Dustdar S (eds) Business process management workshops, BPM 2006 international workshops, BPD, BPI, ENEI, GPWW, DPM, semantics4ws, Vienna, Austria, September 4–7, 2006, Proceedings, vol. 4103 of lecture notes in computer science; 2006. pp. 81–92, Springer.
26. Li G, de Murillas EGL, de Carvalho RM, van der Aalst WMP. Extracting object-centric event logs to support process mining on databases. In: Mendling J and Mouratidis H (eds) Information systems in the big data era - CAiSE Forum 2018, Tallinn, Estonia, June 11–15, 2018, Proceedings, vol. 317 of lecture notes in business information processing; 2018. pp. 182–99, Springer.
27. Calvanese D, Montali M, Syamsiyah A, van der Aalst WMP. Ontology-driven extraction of event logs from relational databases. In: Reichert M and Reijers HA (eds) Business process management workshops—BPM 2015, 13th international workshops,

Innsbruck, Austria, August 31–September 3, 2015, Revised Papers, vol. 256 of lecture notes in business information processing; 2015. pp. 140–53, Springer.
28. Andrews R, van Dun CGJ, Wynn MT, Kratsch W, Röglinger M, ter Hofstede AHM. Quality-informed semi-automated event log generation for process mining. Decis Support Syst. 2020;132:113265.
29. Rodríguez P, Haghighatkhah A, Lwakatare LE, Teppola S, Suomalainen T, Eskeli J, Karvonen T, Kuvaja P, Verner JM, Oivo M. Continuous deployment of software intensive products and services: a systematic mapping study. J Syst Softw. 2017;123:263–91.
30. Senapathi M, Buchan J, Osman H. Devops capabilities, practices, and challenges: Insights from a case study. In: Proceedings of the 22nd international conference on evaluation and assessment in software engineering 2018, EASE'18, (New York, NY, USA); 2018. pp. 57–67, Association for Computing Machinery.
31. Koopman M. A framework for detecting and preventing security vulnerabilities in continuous integration/continuous delivery pipelines. 2019. http://essay.utwente.nl/78048/.
32. Lehtonen T, Suonsyrjä S, Kilamo T, Mikkonen T. Defining metrics for continuous delivery and deployment pipeline. In: Nummenmaa J, Sievi-Korte O and Mäkinen E (eds) Proceedings of the 14th symposium on programming languages and software tools (SPLST'15), Tampere, Finland, October 9–10, 2015, vol. 1525 of CEUR workshop proceedings; 2015. pp. 16–30, CEUR-WS.org.
33. Whittingham M. Revdebug—building a ci/cd pipeline [guide]. 2020.
34. Fedeczko D. Continuous monitoring and observability in ci/cd. 2020. https://codilime.com/blog/continuous-monitoring-and-observability-in-devops.
35. Viggiato M, Oliveira J, Figueiredo E, Jamshidi P, Kästner C. Understanding similarities and differences in software development practices across domains. In: 2019 ACM/IEEE 14th international conference on global software engineering (ICGSE); 2019. pp. 84–94.
36. Brandt CE, Panichella A, Zaidman A, Beller M. Logchunks: a data set for build log analysis. In: Proceedings of the 17th International Conference on Mining Software Repositories, MSR '20, (New York, NY, USA); 2020. pp. 583–7, Association for Computing Machinery.
37. Zhang C, Chen B, Chen L, Peng X, Zhao W. A large-scale empirical study of compiler errors in continuous integration. In: Proceedings of the 2019 27th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering, ESEC/FSE 2019, (New York, NY, USA); 2019. pp. 176–87, Association for Computing Machinery.
38. Rausch T, Hummer W, Leitner P, Schulte S. An empirical analysis of build failures in the continuous integration workflows of java-based open-source software. In: 2017 IEEE/ACM 14th international conference on mining software repositories (MSR); 2017. pp. 345–55.
39. Zampetti F, Scalabrino S, Oliveto R, Canfora G, Di Penta M. How open source projects use static code analysis tools in continuous integration pipelines. In: 2017 IEEE/ACM 14th international conference on mining software repositories (MSR); 2017. pp. 334–44.
40. Martin K, Osterling M. Value stream mapping: How to visualize work and align leadership for organizational transformation. New York: McGraw-Hill Education; 2013.
41. PAFnow. Process mining is the (r)evolutionary progression from value stream analysis. 2020.
42. Mertens K, Bernerstätter R, Biedermann H. Value stream mapping and process mining: a lean method supported by data analytics. In: Proceedings of the 1st Conference on Production Systems and Logistics (CPSL 2020), 2020, S.119–26. https://doi.org/10.15488/9653.
43. Knoll D, Reinhart G, Prüglmeier M. Enabling value stream mapping for internal logistics using multidimensional process mining. Expert Syst Appl. 2019;124:130–42.
44. Ståhl D, Bosch J. Modeling continuous integration practice differences in industry software development. J Syst Softw. 2014;87:48–59.
45. Bass L, Holz R, Rimba P, Tran AB, Zhu L. Securing a deployment pipeline. In: Proceedings of the third international workshop on release engineering, RELENG '15; 2015. pp. 4–7, IEEE Press.
46. Sim S, Bae H, Choi Y, Liu L. Statistical verification of process model conformance to execution log considering model abstraction. Int J Cooperative Inf Syst. 2018;27(2):1850002:1–1850002:23.