A DevSecOps-based Assurance Process for Big Data Analytics

Marco Anisetti, Nicola Bena, Filippo Berto
Department of Computer Science
Università degli Studi di Milano
Milan, Italy
{firstname.lastname}@unimi.it

Gwanggil Jeon
Department of Embedded Systems Engineering
Incheon National Univerisity
Seoul, Korea
gjeon@inu.ac.kr

***Abstract*—Today big data pipelines are increasingly adopted by service applications representing a key enabler for enterprises to compete in the global market. However, the management of non-functional aspects of the big data pipeline (e.g., security, privacy) is still in its infancy. As a consequence, while functionally appealing, the big data pipeline does not provide a transparent environment, impairing the users' ability to evaluate its behavior. In this paper, we propose a security assurance methodology for big data pipelines grounded on the DevSecOps development paradigm to increase trustworthiness allowing reliable security and privacy by design. Our methodology models and annotates big data pipelines with non-functional requirements verified by assurance checks ensuring requirements to hold along with the pipeline lifecycle. The performance and quality of our methodology are evaluated in a real walkthrough analytics scenario.**

***Index Terms*—Assurance; Big Data; Trustworthiness; DevSecOps**

I. Introduction

To compete in the global market, enterprises must take advantage of big data, precisely and quickly analyzing the huge amount of data generated and collected every minute. Data analysis does not only carry challenges related to data analytics and processing, but also critical challenges in data preparation and management. Data in fact can be sensitive and need to be protected and secured once stored and during processing, following strict regulations such as the General Data Privacy Regulation (GDPR) in Europe.

Different ad hoc solutions have been provided to protect big data analytics infrastructures from threats [1]–[3]. They mostly focus on compliance with specific regulations [4], or protection according to traditional requirements such as confidentiality, integrity, availability, and privacy [5]–[7]. Furthermore, these solutions secure specific portions of a big data pipeline only, for instance, data preparation with anonymization techniques [8]. Big data security is instead a much larger and multi-facets problem, related to data protection techniques along with all phases of the pipeline as well as to the underlying big data infrastructure. The pipeline inevitably inherits non-functional leaks or strengths from the execution platform. In turn, the platform depends on the cooperation and security features of the (cloud) services composing the platform. For instance, services executed on a platform verified for confidentiality at rest inherit the support for confidentiality, if configured properly. Lastly, the development of big data pipelines is beginning to approach DevSecOps, a development process integrating security in all its phases. This adoption is however in its infancy, requiring the adaptation of traditional DevSecOps controls to big data pipelines, which are much more complex and difficult than traditional cloud-based services. To conclude, the lack of proper solutions for security and assurance of big data pipelines limits their trustworthiness, and represents a major barrier against high-quality big data computations. There is a clear need for a more holistic *security assurance solution* for big data pipelines, covering the underlying ecosystem of services, the pipeline itself, and its development process, in order to finally increase its trustworthiness.

The approach in this paper aims to fill in the above gap and puts forward the idea that the development of big data pipelines can benefit from a standardized paradigm inspired by DevSecOps, where security checks are considered at each point of the pipeline life cycle. Our approach builds on security assurance techniques evaluating whether the target big data pipeline behaves as expected in terms of given non-functional requirements (e.g., confidentiality). It enriches the functional descriptions of the target pipeline with the non-functional requirements that should hold at each step, and connects these requirements with assurance checks to be executed at each point of the corresponding DevSecOps development process, from planning to operations [9]. Checks vary from model checking against non-functional requirements to traditional infrastructure-level checks. In short, our approach takes advantage of DevSecOps development of big data pipelines to fully unleash their potential even in critical scenarios where many requirements must be honored, promoting the "big data-analytics enabled services" paradigm, where trustworthiness is of paramount importance.

The contribution of this paper is manifold. We first model big data pipelines and present a DevSecOps approach for their development. Second, we define and implement a novel assurance evaluation suitable for DevSecOps development of big data pipelines. We then apply it in the context of the development of a real big data pipeline demonstrating its usefulness and feasibility.

The remainder of this paper is organized as follows. Section II discusses the notion of non-functional assurance evaluation based on assurance checks. Section III introduces our modelling approach for big data pipelines, whose assurance evaluation methodology is presented in Section IV. Section V discusses our experimental evaluation. Section VI describes related work, and Section VII gives our conclusions.

II. Assurance Evaluation

Non-functional assurance models the degree of confidence to which a system supports non-functional requirements (e.g., "Confidentiality", "Inter node communication security"). Our assurance evaluation is implemented using assurance checks. For instance, let us consider a non-functional requirement r=*data confidentiality at rest*. Two different mechanisms of the target system can be considered as relevant for r: *i)* a mechanism for storage encryption, and *ii)* an access control mechanism protecting data access. The assurance evaluation measures, with a certain degree of confidence, whether the two mechanisms support the confidentiality defined in r. For instance, it can use two assurance checks: one controlling the strength of the encryption algorithm and the other the correctness of the access control system. Assurance checks are defined as follows.

Definition 2.1 (c): An assurance check c is a function of the form c: (t, r, *param*) $\rightarrow$ (*result*, *artifacts*), where t is the target of the check; $r \in R$ is the requirement to verify; *param* is the configuration of c in terms of, for instance, inputs, expected outputs, URI of the target; *result* is the outcome of the check, $\top$ if t supports r, $\perp$ otherwise; *artifacts* is a set of details motivating the outcome (e.g., the output of test cases used in the check).
Our assurance evaluation uses the outcomes of a set of assurance checks $\{c_1, \ldots, c_n\}$ to evaluate the assurance confidence.

Definition 2.2 ($L(\{c_1, \ldots, c_n\})$): The assurance confidence level is a function of the form L: $\{c_1, \ldots, c_n\} \rightarrow [0,1]$, where $\{c_1, \ldots, c_n\}$ is a set of assurance checks referred to the same requirement $r \in R$ and target t. It is computed as the ratio between the number of succeeded checks (i.e., assurance checks returning *result*=$\top$) and of all checks $|\{c_1, \ldots, c_n\}|$. For instance, assuming that the assurance check on storage encryption fails while the one on access control succeeds in the above example, the assurance confidence level returns 0.5.

In this paper, assurance checks are implemented as *probes* of the following types: *i) testing probes*, submitting test cases against a given target and inspecting the results; *ii) configuration probes*, evaluating configuration files looking for specific issues; and *iii) monitoring probes*, monitoring the behavior of the target system. We note that the specific activities implemented in probes depend on the peculiarities of target t and requirement r.

III. Big Data Pipeline

In this paper, we represent a big data pipeline p using a two-steps modelling approach (i.e., declarative and procedural) similar to the one used by TOREADOR project [10].

A. Declarative model

The declarative model describes a big data pipeline at a high level of abstraction in terms of *workflow* of steps and *infrastructure layer* of services supporting the workflow execution.

Definition 3.1 (T_p): A declarative model T_p for a given pipeline p is defined as $\langle[\theta_1, \ldots \theta_n], [\sigma_1, \ldots, \sigma_m]\rangle$ where

- $[\theta_1, \ldots \theta_n]$ is an ordered sequence of steps, each step θ_i of the form (A, J) describing the abstract processing jobs $j \in J$ (e.g., cleaning, dimension reduction, clustering);
- $[\sigma_1, \ldots, \sigma_m]$ is a set of abstract big data infrastructure layers σ_i of the form (A, S) describing the abstract ecosystem of services $s \in S$ supporting the pipeline execution (e.g., distributed filesystem, processing engine, workflow orchestrator).
- A represents a specific area of jobs or services and is taken from a controlled vocabulary (e.g., ingestion, preparation, analytics, visualization).

Each area A is connected to specific non-functional requirements that should hold in A. In the following, we describe some of the most prominent areas and requirements.

- *Ingestion*. The technological connection to the source of data must be compliant with non-functional requirements. For instance, specific streaming technologies can be used to achieve specific performance requirements.
- *Preparation*. Ingested data should address non-functional requirements since the veracity of the origin is not enough (e.g., by filtering, cleansing, and enrichment classification). For instance, it requires fast data granulation techniques to bring large volumes of data to a granularity and detail level compatible with the privacy preferences and non-disclosure requirements of the data owners.
- *Analytics*. Big data computations should be modeled and processed following non-functional requirements. Analytics algorithms should be designed with security and privacy requirements in mind, resulting in security- and privacy-aware processing. For instance, an analytics process supporting confidentiality should not disclose processed information with external services (i.e., outside the big data perimeter).
- *Visualization*. The outcome of big data computations can be sensitive and need to be protected by inferences that might affect the privacy of data owners. This outcome is in general the result of aggregations that might reveal critical data. This scenario should be forbidden, still preserving a given quality of data.
- *Processing*. The ecosystem of service supporting distributed and parallel processing must be compliant with non-functional requirements. Any leakages in the distribution of the processing tasks or their orchestration should be considered, such as insecure distribution channels or untrusted orchestration.
- *Storage*. The ecosystem of services for distributed data storage must be verified against non-functional require-

ments. Similar to traditional storage systems, their weaknesses in protecting data at rest have to be considered.

- *Infrastructure*. It refers to the lowest layer of the big data infrastructure, including virtualization, operating system, or, in general, the environment where the services are executed. Weaknesses at the infrastructure level may have a severe impact on the entire system.

The declarative model can refer to a set of requirements R in form of annotations as follows.

Definition 3.2 (T_p^R): An annotated declarative model T_p^R is a declarative model T_p where each step θ_i and infrastructure layer σ_j can be annotated with one or more requirement $r_k \in R$, denoted as $\theta_i^{r_k}$ and $\sigma_j^{r_k}$.

The declarative model allows defining an abstract design plan for a big data pipeline p, leaving implementation details for further refinement.

Example 3.1 (Annotated Declarative Model): Let us consider a non-functional requirement r=*Confidentiality*. An annotated declarative model T_p^R can be defined as $\langle$[(*Ingestion*, {*batch*})^r, (*Preparation*, {*cleaning*})^r, (*Analytics*, {*regression model*})^r, (*Visualization*, {*save*})^r], [(*Processing*, {})^r, (*Storage*, {})^r, (*Infrastructure*, {})^r]$\rangle$.

The declarative model in the above example models a pipeline performing any kind of regression after batch ingestion and data cleaning, then saving the model on the storage. We note that every workflow step and infrastructure service is asked to satisfy a generic "Confidentiality" non-functional requirement.

B. Procedural Model

The procedural model represents the incarnation of a declarative model T_p into a more concrete big data pipeline model. It defines implementation details that are missing in the declarative model.

Definition 3.3 (I_T^R): Let T_p^R be an annotated declarative model. An annotated procedural model I_T^R instantiates T_p^R by replacing

- each abstract job $j \in J$ in θ_i with a concrete job of the form (j, f) where j is the abstract job in T_p^R and f is the concrete function implementing j;
- each abstract infrastructure layer $s \in S$ in σ_i with the concrete service supporting job execution;
- each annotated abstract requirement $r \in R$ with a more concrete and specialized one.

Example 3.2 (Annotated Procedural Model): Let us consider the annotated declarative model in Example 3.1 and two requirements r_1=*Confidentiality at rest* and r_2=*Confidentiality in transit* derived from abstract requirement r. A procedural model I_T^R with R=[r_1, r_2] can be defined as $\langle$[(*Ingestion*, {(*batch*, LoadHDFS())})^{r_1}, (*Preparation*, {(*cleaning*, Clean())})^{r_1}, (*Analytics*, {(*regression model*, m=Regression())})^{r_1}, (*Visualization*, {(*save*, SaveHDFS(m))})^{r_1}], [(*Processing*, {Spark, Airflow})^{r_2}, (*Storage*, {Hadoop HDFS})^{r_1}, (*Infrastructure*, {Linux Node})^{r_1,r_2}]$\rangle$. The annotations on I_T^R express the need to cope with specific requirements, for instance jobs in the *Ingestion* area (i.e., function LoadHDFS()) to support *confidentiality at rest* (r_1), while the services in the *Infrastructure* area (i.e., the Linux Node) to support both *confidentiality at rest* (r_1) and *confidentiality in transit* (r_2). The services in the infrastructure layers of I_T^R are:

- *Hadoop HDFS*: it is a distributed file system designed to support the *Hadoop Map-Reduce* paradigm. It is the core component of the Apache big data architecture. It also supports services such as *Apache Hive* in storing SQL-like datasets. It is the storing service of the pipeline.
- *Spark*: it is a framework for large-scale distributed processing, based on the concepts of in-memory caching and optimized query execution. It offers a set of Machine Learning libraries (*Mlib*) to be used for analytics development. Its tasks are defined as a Direct Acyclic Graph (DAG) of data transformations over a Resilient Distributed Dataset (RDD). It is one of the processing services of the pipeline.
- *Airflow*: it is a workflow manager orchestrating executions of different Spark tasks, according to DAG orchestration. It is one of the processing services of the pipeline, responsible for the orchestration of the whole workflow.
- *Linux Node*: it is the operating system of the computing and storage nodes where the pipeline is executed.

These infrastructure services support the following functions of the workflow jobs in I_T^R.

- *LoadHDFS*: it is the function of the ingestion job providing batch ingestion. It is implemented in Spark and loads raw data from *Hadoop HDFS*.
- *Clean*: it is the function of the preparation job providing data cleaning. It is implemented in Spark and cleans the raw data in order to make them available for analytics.
- *Regression*: it is the function of the analytics job providing regression modelling. It is implemented in *Spark Mlib* and creates a linear regression model out of the dataset. It returns a model called m.
- *SaveHDFS*: it is the function of the visualization job saving the generated model m for later usage (i.e., prediction). It is implemented in Spark and saves data on the *Hadoop HDFS*.

C. DevSecOps for big data pipeline

DevSecOps is a development process methodology focused on ensuring continuous integration (CI) and deployment (CD) on one side ("Dev" and "Ops"), and embedding security checks in the development process on the other ("Sec") [11]. It is a sequence of activities (*stages*) in a continuous loop, organized in two groups: *development* and *operation*. In the context of a big data pipeline, these groups of stages are as follows.

Development focuses on the design and implementation of the pipeline according to the following stages.

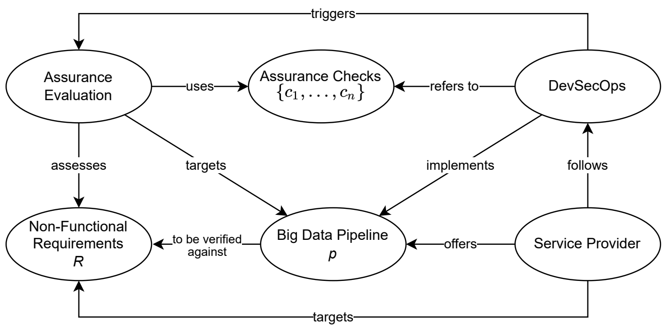

Fig. 1. Our reference scenario.

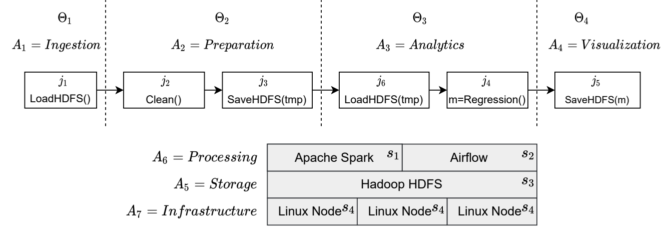

Fig. 2. The running example. Workflow jobs are presented in white boxes while infrastructure layer services in grey boxes.

- *Plan*: the service provider designs the pipeline at a high level identifying the areas for jobs and services and abstract job representations.
- *Code*: the service provider produces code for the jobs of the pipeline. It first identifies the jobs that are suitable for the pipeline for every area and then implements them.
- *Build*: the service provider defines the workflows of jobs and the supporting ecosystem of service.
- *Test*: the service provider prepares testing activities for each job of the pipeline.
- *Release*: the service provider identifies the operation or staging target and executes integration testing, that is, testing of the whole workflow.
- *Deploy*: the service provider deploys the pipeline on a big data engine in a staging or operation environment, that is, the pipeline is moved from a local platform to the final one (or similar to the final one). It includes also eventual details on the containerization and container orchestration solution adopted for the deployment (e.g., Kubernetes).

We note that stages *Plan*, *Code*, *Build*, and *Test* are normally carried out in the local development environment, while stage *Release* is meant to be ready for deployment in staging or operation environment. Testing activities in the local environment have to be adapted for the release of the entire pipeline in staging or operation. We also note that despite the deploy stage is typically considered part of the *operation* group, given the importance of the deployment environment for big data pipeline development, we consider it part of the *development* group.

Operation focuses on packaging and deploying the pipeline in staging/production environments according to the following stages.

- *Operate*: the service provider re-executes part of the verification carried out in the development stages in the operation environment, with more emphasis on the ecosystem of services. It also introduces infrastructure-levels checks.
- *Monitor*: the service provider monitors the pipeline behavior by inspecting execution traces.

In this paper, we embed our notion of assurance evaluation in Section II into the above DevSecOps process applied to big data pipelines, in order to evaluate pipelines against non-functional requirements, to increase their security and, in turn, trustworthiness.

D. Reference scenario

Figure 1 shows our reference scenario where a service provider wishes to develop a big data pipeline p ensuring a specific set of non-functional requirement R. Following a DevSecOps approach, our assurance evaluation and corresponding assurance checks are automatically triggered to continuously verify the requirements R during the pipeline development and operation lifecycle. Figure 2 shows our running example, underlining jobs, services, and the different areas. It is an extension of the pipeline in Example 3.2 having the following annotated procedural model.

Example 3.3 (Running Example): The annotated procedural model I_T^R of our running example is $\langle[(Ingestion, \{(batch, \text{LoadHDFS()})\})^{r_1}, (Preparation, \{(cleaning, \text{Clean()})\}, \{(save, \text{SaveHDFS(tmp)})\})^{r_1}, (Analytics, \{(batch, \text{LoadHDFS(tmp)})\}, \{(regression\ model, m{=}\text{Regression()})\})^{r_1}, (Visualization, (save, \{\text{SaveHDFS(m)}\}))^{r_1}], [(Processing, \{\text{Spark, Airflow}\})^{r_2}, (Storage, \{\text{Hadoop}\})^{r_1}, (Infrastructure, \{\text{Linux Node}\})^{r_1,r_2}]\rangle$. It extends the procedural model in Example 3.2 with jobs $(save, \text{SaveHDFS(tmp)})$ saving cleaned data on a temporary location *tmp*, and $(batch, \text{LoadHDFS(tmp)})$ loading cleaned data. r_1 and r_2 refer to requirements *Confidentiality at rest* and *Confidentiality in transit*, respectively, in Example 3.2.

IV. Assurance Evaluation Methodology for big data pipeline

Our assurance evaluation methodology for big data pipeline, depicted in Figure 3, follows the DevSecOps paradigm. It starts from the design of the big data pipeline p at stage *Plan* and the corresponding modeling and the annotations of requirements R as in Section III. It then introduces, during the "Development" DevSecOps stages, specific assurance checks to be executed by the CI/CD DevSecOps framework (*Development Assurance*). Prior to move to the "Operation" stages, our assurance methodology embeds relevant assurance checks within the big data pipeline (*Assurance Weaver*) to execute them in the operation environment. The embedded assurance checks are selectively executed during the "Operation" stages according to scheduling options added by the *Assurance*

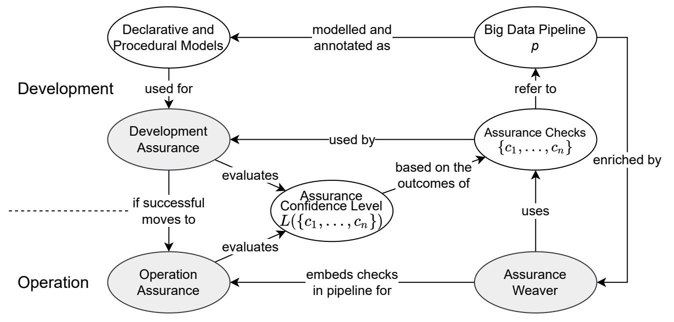

Fig. 3. Our Assurance Evaluation Methodology for big data pipeline. With grey background our methodology building blocks.

Weaver (e.g., executing the checks every time or just at the first deployment) (*Operation Assurance*).

We note that the operation environment is normally not under full control of the developers, therefore the presence of assurance checks in the pipeline enables to monitor the pipeline behavior while deployed in production.

Development Assurance and Operation Assurance collect assurance checks outcomes (in the form of (*result*, *artifacts*) according to Definition 2.1) at each iteration of the DevSecOps loop. They then evaluate the corresponding assurance confidence level $L(\{c_1, \ldots, c_n\})$ in Definition 2.2 (evaluating the set of *result*), to proceed in the DevSecOps stages and to support remediation and improvements (evaluating the set of *artifacts*). More specifically, the assurance confidence level is computed on the set of assurance checks that share the same target (step θ or infrastructure layer σ) and requirement r.

In the following, we first describe our Development Assurance addressing "Development" DevSecOps stages. We then describe the Assurance Weaver and the corresponding assurance checks execution within Operation Assurance.

A. Development Assurance

Development Assurance refers to the assurance activities carried out during the *Development* stages of DevSecOps. Type of checks to be executed at these stages are as follows.

- Plan - *Pipeline annotation*: the big data pipeline is modeled and annotated with requirements R following the declarative and procedural models in Section III. Checks on the requirements are carried out manually at this stage.
- Code - *Instance check*: assurance checks verify the correspondence between the annotations on the declarative model T_p^R and requirements R, and between the annotated declarative and procedural models T_p^R and I_T^R, respectively. Specifically, checks fail in case a requirement is not annotated or wrongly annotated, or wrong correspondence between T_p^R and I_T^R (i.e., in terms of steps θ_i or infrastructure layers σ_i).
- Build - *Workflow verification*: assurance checks verify the workflow comparing I_T^R with the implemented pipeline p in terms of job correspondence and workflow structure. We note that this is the first stage where the implemented pipeline p is checked against the pipeline model.

Every assurance confidence computed at each stage must be higher than a given threshold to trigger the next stage in the DevSecOps loop. We note that the thresholds are chosen by experts according to the considered scenario. In our running example in Section III-D, when not specified differently, we consider an assurance confidence level threshold of 0.7.

Example 4.1 (Stages Code and Build): Let us consider the declarative model T_p^R in Example 3.1 and the corresponding procedural model I_T^R in Example 3.2. The procedural model I_T^R is a correct instantiation of T_p^R according to our *Instance check*, since all the elements (i.e., steps and infrastructure layers) in T_p^R are instantiated in I_T^R and with correct non-functional requirements deriving from the abstract one annotated on the corresponding elements in T_p^R (assurance check c_i). The check c_i is the only check of type *Instance check* and its *result* is $\top$ (Definition 2.1), therefore the assurance confidence level of the corresponding evaluation at stage *Code* is $L(\{c_i\})$=1. The DevSecOps process can then move on. Let us assume that in the next development iteration the pipeline p is updated with the inclusion of two more jobs like in Example 3.3 (i.e., our running example). Therefore, p differs from I_T^R in Example 3.3. An assurance check c_j of type *Workflow verification* during *Build* stage identifies this discrepancy by inspecting the coded pipeline p with respect to I_T^R, therefore returning $\bot$. As a consequence, the assurance confidence level of the corresponding evaluation is $L(\{c_j\})$=0. This failure prevents the DevSecOps process from moving on, requiring remediation. In fact, even if functionally equivalent, the fact the data are temporarily stored after the cleaning job can constitute a possible violation of r_1=*Confidentiality at rest*.

Other checks of Development Assurance are as follows.

- Test - *Job verification*: assurance checks verify the implementation of each job in p against the annotated requirements $r \in R$ separately. For instance, they search for code weaknesses relevant for each requirement r. As another example, they verify a that given preprocessing job does not introduce data leakage (e.g., not allowing re-identification of anonymized ingested data).
- Release - *Release validation*: assurance checks verify the packaging against requirements R. For instance, they verify that jobs packaged in containers are uploaded to a private container registry not accessible from the outside.
- Deploy - *Deployment validation*: assurance checks verify the deployed services against requirements R. For instance, they verify that services at infrastructure layers are correctly configured with respect to requirements.

Example 4.2 (Stages Test, Release, and Deploy): Let us consider procedural model I_T^R in Example 3.3 and the corresponding implementation p in Example 4.1. Each job needs to be verified against the annotated requirement. For instance, in case the function of job "Clean()" calls an external service for cleaning data, our *Job verification* check identifies this call as a possible violation for requirement r_1. Assuming assurance confidence level threshold equal to 1 for this singleton evaluation composed of check *Job verification*, this failure blocks from moving to stage *Release*. Assuming that the above issue is fixed, let us then consider that the entire pipeline is deployed with Docker. Checks of type *Release validation* validate the correctness of containerized solution by inspecting the *docker compose* file. In this case, it contains a correct containerization, allowing to deploy the pipeline as expected (assurance confidence level=1). Checks of type *Deployment validation* verify the correctness of all services' configurations in p with regards to r_1 and r_2. In this case, all the configurations allow for data protection via encryption and enable secure communication channels, therefore the assurance confidence level is equal to 1.

B. Assurance Weaver

The *assurance weaver* embeds the relevant assurance checks as concrete jobs of the pipeline workflow, mimicking the *Aspect Weaver* of the Aspect Oriented programming. The checks are then executed as part of the pipeline and can benefit from parallelization if realized accordingly. The assurance weaver is formally defined as follows.

Definition 4.1 ($W(I_T^R, p, \{c_1, \ldots, c_n\})$): Assurance weaver is a function of the form $W\colon (I_T^R, p, \{c_1, \ldots, c_n\}) \rightarrow \hat{p}$. It takes in input the procedural model I_T^R, the implemented pipeline p, the set of identified assurance checks $\{c_1, \ldots, c_n\}$ and produces as output a pipeline $\hat{p}$ with checks $\{c_1, \ldots, c_n\}$ added as jobs of the pipeline p.

First, the weaver adds auxiliary steps $\{\gamma_i\}$ to the procedural model I_T^R, generating a weaved procedural model $\widehat{I_T^R}$. Each auxiliary step γ_i models an assurance check c_i, and includes *i)* the abstract job j as the type of assurance check (e.g., *Job verification*, *Deployment validation*); *ii)* the job function f as c_i. The auxiliary steps are added before or afterward the workflow steps θ in $\widehat{I_T^R}$ according to the type of assurance checks. Finally, the weaver automatically modifies the implemented pipeline p code guided by the weaved procedural model $\widehat{I_T^R}$.

Example 4.3 (Assurance Checks Weaving): Let us consider pipeline p in Example 4.2. The assurance weaver weaves within p the assurance check c_k of area *Storage* and type *Deployment validation* verifying the HDFS configurations via function f=CheckHDFS(hook, conf). The weaved procedural model $\widehat{I_T^R}$ is defined as $\langle$[(*Ingestion*, {(*batch*, LoadHDFS())})^{r_1}, (*Preparation*, {(*cleaning*, Clean())}, {(*save*, SaveHDFS(tmp))})^{r_1}, (*Analytics*, {(*batch*, LoadHDFS(tmp))}, {(*regression model*, m=Regression())})^{r_1}, (*Visualization*, {(*save*, SaveHDFS(m))})^{r_1}, (*Storage*, {(*Deployment validation*, CheckHDFS(hook, conf))})], [(*Processing*, {Spark, Airflow})^{r_2}, (*Storage*, {Hadoop})^{r_1}, (*Infrastructure*, {Linux Node})^{r_1,r_2}]$\rangle$.

We note that the weaved assurance checks are subjected to scheduling, meaning that even if the check is weaved in the pipeline, it can be executed either every time the pipeline is executed or in case of a specific event. The scheduling option is added as parameter of the assurance checks during weaving. For instance, the HDFS check in Example 4.3 can be scheduled at every re-deployment.

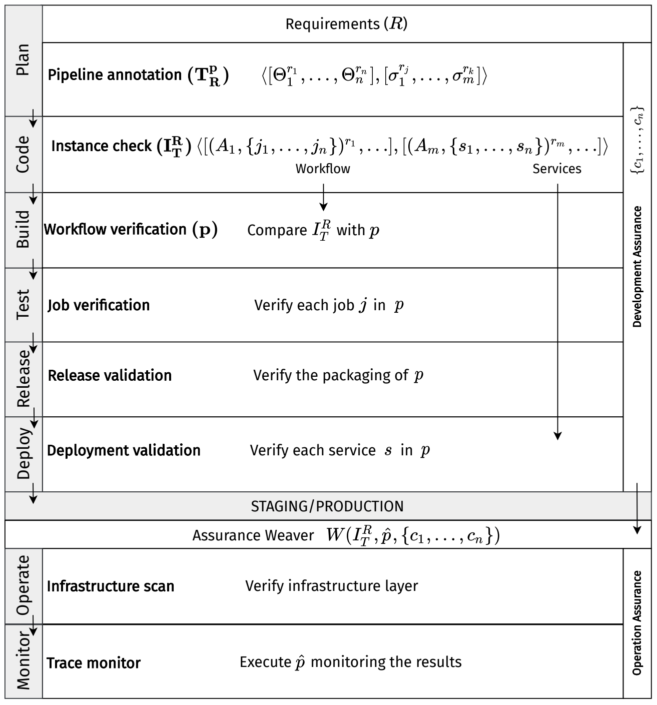
Requirements (R)
Plan
Pipeline annotation ($\mathbf{T_R^p}$) $\langle[\Theta_1^{r_1}, \ldots, \Theta_n^{r_n}], [\sigma_1^{r_j}, \ldots, \sigma_m^{r_k}]\rangle$
Code
Instance check ($\mathbf{I_T^R}$) $\langle[(A_1, \{j_1, \ldots, j_n\})^{r_1}, \ldots], [(A_m, \{s_1, \ldots, s_n\})^{r_m}, \ldots]\rangle$
Workflow
Services
$\{c_1, \ldots, c_n\}$
Build
Workflow verification ($\mathbf{p}$)
Compare I_T^R with p
Test
Job verification
Verify each job j in p
Release
Release validation
Verify the packaging of p
Deploy
Deployment validation
Verify each service s in p
Development Assurance
STAGING/PRODUCTION
Assurance Weaver $W(I_T^R, \hat{p}, \{c_1, \ldots, c_n\})$
Operate
Infrastructure scan
Verify infrastructure layer
Monitor
Trace monitor
Execute $\hat{p}$ monitoring the results
Operation Assurance

Fig. 4. Relation between the DevSecOps and our methodology. Development Assurance and Operation Assurance together with Assurance Weaver constitute the building blocks of our assurance evaluation methodology.

C. Operation Assurance

Operation assurance refers to the assurance activities carried out during the *Operation* stages of DevSecOps. It includes two types of assurance checks driven by the assurance weaver.

- Operation - *Infrastructure scan*: assurance checks verify the infrastructure underlying the pipeline with regards to requirements R. For instance, it includes checks against cloud provider' configurations or against the operating system of private servers.
- Monitor - *Trace monitor*: assurance checks verify the execution of each part of the pipeline with regards to requirements R, reusing the assurance checks weaved in the pipeline. We note that inputs of the weaved checks can involve values that are only available in these stages [12]. For instance, the path of HDFS storage input of an assurance check to verify its configuration is available only in the operation environment (i.e., it can differ from the one used in staging).

We note that, during operation assurance, the assurance confidence level is retrieved as in development assurance in Section IV-A but it does not block any DevSecOps stages.

Example 4.4 (Operate and Monitor stages): Let us consider the weaved pipeline $\hat{p}$ in Example 4.3. A check of type *Infrastructure scan* inspects the Linux nodes of our cluster looking for vulnerabilities. Assuming that some nodes host a Linux distribution with some well-known vulnerabilities, the assurance check returns a negative *result* ($\perp$), implying that some remediation has to be carried out on the infrastructure.

Let us then assume that a check of type *Trace monitor* verifying the working of job “Clean()” has been weaved in the pipeline. The check requires information on the cleaning parameters, which are defined at run time. Our weaver is able to retrieve and weave this information, allowing the check to work properly.

Figure 4 shows the relation between our assurance methodology for big data pipelines and DevSecOps stages.

V. Experimental Evaluation

We evaluated our approach in terms of soundness and usefulness, by presenting a walkthrough of our reference example in Section III-D and evaluating its performance.

A. Experimental Setup

Our experimental setup is based on GitLab CI/CD where we implemented, annotated, and evaluated the big data pipeline in Section III-D according to our methodology. Our big data platform is based on the Apache ecosystem with Hadoop as the distributed filesystem, Spark as the computation framework, and Airflow for pipeline definition and scheduling. GitLab CI/CD orchestrates the jobs linked to the development and deployment of the pipeline, including tests, assurance checks, and operation monitoring. The *GitLab runner* is deployed on a bare-metal configuration of NixOS with an AMD 5900x processor, 32 GB of memory, and Linux kernel 5.17.5. The running example we implemented was applied to the “Titanic” dataset[1] with the goal of producing a linear regression model predicting whether a passenger will survive the sinking. We consider a portion of the dataset for testing during *development* stages, and the entire dataset for the deployment in *operation*.

B. Walkthrough Evaluation

In the following, we present a set of concrete checks and their results on the first complete run of DevSecOps applied to the above pipeline, whose modeling is shown in Table I. For simplicity, we set the Confidence Level threshold to 0.7 for all the evaluations, although we note that the threshold can vary (Definition 2.2). For brevity, we present the assurance checks' parameters *param* only, while target t and requirements r for each check c are listed in Table I.

1) Development Assurance: Figure 5 shows a portion of the YAML file describing the different checks to be carried out in the different *development* stages. We note that the YAML file includes a stage called *operation* where the assurance weaver is executed. We also note that the declarative model T_p^R is specified in the YAML as `MODEL`, while the implemented pipeline in the procedural model I_p^R is specified as `DAG_FILE`. Table II shows the pseudo code of probes implementing the assurance checks described in the following.

At stage *Code*, two assurance checks of type *Instance check* are applied checking: *i)* the correctness of annotations of requirements R in the declarative model T_p^R ($c_1(T_p^R, R)$); and *ii)* I_T^R with respect to T_p^R ($c_2(T_p^R, I_T^R)$). Both checks succeeded, leading to assurance confidence level $L(\{c_1, c_2\})$=1.

[1]https://www.kaggle.com/c/titanic

TABLE I
Walkthrough of our running example in Example 3.3

Worflow in I_T^R		
A_1=Ingestion	j_1=(*batch*, LoadHDFS())	r_1
A_2=Preparation	j_2=(*cleaning*, Clean()) j_3=(*save*, SaveHDFS(tmp))	r_1
A_3=Analytics	j_6=(*batch*, LoadHDFS(tmp)) j_4=(*regression model*, m=Regression())	r_1
A_4=Visualization	j_5=(*save*, SaveHDFS(m))	r_1

Services in I_T^R		
A_6=Processing	$\{s_1$=Spark, s_2=Airflow$\}$	r_2
A_5=Storage	$\{s_3$=Hadoop$\}$	r_1
A_7=Infrastructure	$\{s_4$=Linux Node$\}$	r_1, r_2

Assurance Checks c

	t	**r**
c_1	T_p^R	r_1, r_2
c_2	I_T^R	
c_3	p	
c_4	$j_1, \ldots, j_6$	r_1
c_5	$j_1, \ldots, j_6$	r_1
c_6	$j_1, \ldots, j_6$	r_1
c_7	j_3, j_5, j_6	r_1

	t	**r**
c_8	p	r_1, r_2
c_9	s_1	r_2
c_{10}	s_2	r_2
c_{11}	s_3	r_1
c_{12}	s_4	r_1, r_2
c_{13}	s_4	r_1, r_2

R = r_1= Confidentiality at rest
r_2= Confidentiality in transit

$$I_T^R = \langle [(A_1, \{j_1\})^{r_1}, (A_2, \{j_2, j_3\})^{r_1}, (A_3, \{j_6, j_4\})^{r_1}, (A_4, \{j_5\})^{r_1}], [(A_6, \{s_1, s_2\})^{r_2}, (A_5, \{s_3\})^{r_1}, (A_7, \{s_4\})^{r_1, r_2}] \rangle$$

```
...
variables:
  DAG_FILE: ./dags/classification.py
  MODEL: ./model.yml

stages:
  - code
  - build
  - test
  - release
  - deploy
  - operation

# CODE
c1:
  stage: code
  script:
    - python3 ./tests/c1.py -r r1 -r r2
  rules:
    - if: $SKIP_C1 != "true"
c2:
  stage: code
  script:
    - python3 ./tests/c2.py
  rules:
    - if: $SKIP_C2 != "true"
...
```

Fig. 5. Portion of the YAML description for Development Assurance checks.

At stage *Build*, one assurance check of type *Workflow verification* is applied checking I_T^R against p ($c_3(I_T^R, p)$) with success, leading to assurance confidence level $L(\{c_3\})$=1.

At stage *Test*, four assurance checks of type *Job verification* are applied targeting the jobs of the pipeline's workflow: *i)* one check looking for vulnerable code specifically impacting r_1 ($c_4(p)$); *ii)* two checks for code inspection aimed at finding hidden temporary storage where data can be stored during the processing impacting r_1 ($c_5(p, \textit{uris})$), and finding connection to external services allowing data exfiltration ($c_6(p, \textit{urls})$);

iii) one check verifying the correctness of the endpoints' configurations (*expected_args*) in relation to requirement r_1 (e.g., correct storage endpoint) ($c_7(p, \textit{expected\_args})$). Check c_4 raised some warnings related to r_1 that are severe enough to consider the check failed, leading to assurance confidence level $L(\{c_4,\ c_5,\ c_6,\ c_7\})$=0.75. This level is above the threshold allowing to move to the next DevSecOps stage.

At stage *Release*, one assurance check of type *Release validation* was applied to check the Docker files (*docker_file*) and Docker Compose file (*docker_compose_file*) to check the composition ($c_8(\textit{docker\_file}, \textit{docker\_compose\_file})$). These files allow for a correct deployment of our pipeline, leading to an assurance confidence level $L(\{c_8\})$=1.

At stage *Deploy*, three assurance checks of type *Deployment validation* are applied analyzing the configurations of Spark, Airflow, and HDFS in terms of acceptable threshold of warnings ($c_9(\textit{threshold}, r_2)$, $c_{10}(\textit{threshold}, r_2)$ and $c_{11}(\textit{threshold}, r_1)$). Checks c_9 against Spark and c_{11} against HDFS failed, revealing wrong configurations in relation to encryption, leading to $L(\{c_9,\ c_{10}\})$=0.3. Being below our threshold, it prevents other stages.

After fixing the above issues, the CI/CD moves to *operation*, where the assurance weaver embeds assurance checks within the Big Data pipeline. In particular, it introduces c_4 and c_7 prior to pipeline stage θ_1; c_9, c_{10}, and c_{11} after pipeline stage θ_4. It also adds scheduling parameter to c_4, c_9, c_{10}, c_{11} to trigger them just once, while c_7 has not scheduling restrictions.

2) Operation Assurance: At stage *Operate*, we applied two additional assurance check of type *Infrastructure scan* to check operating system-level vulnerabilities against the Linux cluster: *i)* c_{12}(*threshold*, *openscap_conf*, *environment*) based on OpenSCAP; *ii)* c_{13}(*threshold*, *openvas_conf*, *environment*) based on OpenVAS for requirements r_1 and r_2. No severe vulnerabilities were identified, leading to an assurance confidence level $L(\{c_{12},\ c_{13}\})$=1.

At stage *Monitor*, we applied assurance checks of type *Trace monitor*, corresponding to those embedded by the assurance weaver (Section V-B1). They discovered no issues, leading to a Confidence Level $L(\{c_4,\ c_7,\ c_9,\ c_{10},\ c_{11}\})$=1.

C. Performance

We evaluated the performance of our approach in terms of the additional computational time requested by the application of the assurance methodology in Section IV on the running example in the above walkthrough. For the sake of simplicity, we consider the following performance scenarios, based on the assumption that assurance checks are triggered when needed due to relevant changes (e.g., changes on the target of assurance, assurance checks, or environment).

- First Deployment (FD): it includes all the checks requested for the first complete round of DevSecOps.
- Pipeline Updates (PU): it includes changes at pipeline level due to updates on jobs or services. This scenario forces the re-evaluation of Development Assurance.

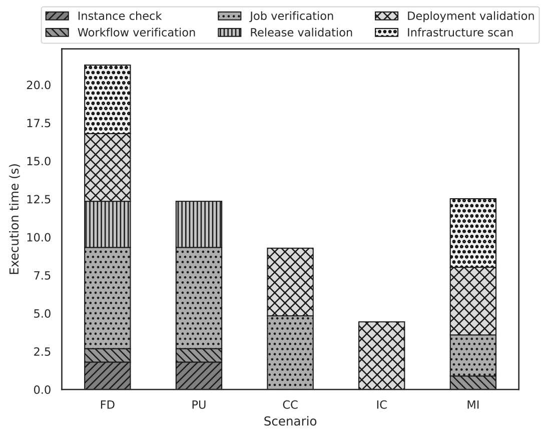

Fig. 6. Performance of our walkthrough in different scenarios.

- Checks Changes (CC): it considers updates of assurance checks, in particular those used in operation assurance. This scenario forces the re-execution of the updated checks both in Development and Operation.
- Infrastructure Changes (IC): it considers changes on the operation infrastructure or, in general, the environment, including the discovery of new weaknesses potentially impacting the infrastructure. It forces the re-evaluation of *Infrastructure scan* during Operation Assurance.
- Monitoring Issues (MI): it considers situations where issues are found during monitoring. Once the corresponding jobs are fixed, assurance checks insisting on such jobs are re-executed.

Figure 6 shows the results of our performance evaluation in the different scenarios. We note that *i)* the histogram does not include check c_{13} for plotting reasons since its time of execution (available in Table III) is significantly larger then the other checks; *ii)* the times of execution are stacked assuming a sequential execution, although most checks can be parallelized. Table III shows the checks included in each scenario and their averaged execution time. c_1, c_2, c_3, c_5, c_6, c_7, c_{10} and c_{11} terminated in approximately 1 second without raising any warnings. c_4 executed in 3.9 seconds raising only minor warnings related to code style and readability. c_8 terminated in 3 seconds, with the Docker images already built and retrieved from cache. Check c_9 executed in 2.1 seconds while c_{11} in 1.2, and they both raised warnings regarding misconfigurations in the encryption settings of the respective target services. Finally, c_{12} and c_{13} terminated respectively in 4.5 and 93.3 seconds, without raising any issues.

To conclude, the execution time of the checks in each stage did not vary much between the different scenarios, with the total execution time in the most complete scenario (FD) being less than 21 seconds, which is a far shorter than the typical time required to execute a big data pipeline. In fact, our pipeline executes in 108 seconds on our experimental setup based on a small dataset (i.e., almost 1300 rows), therefore the entire assurance overhead is approximately 19%.

TABLE II
ASSURANCE CHECK PROBES: PSEUDO CODE

Instance check

```
def c1(model, expected_requirements):
model = Model.parse_file(model)
model_requirements = {
  r for s in model.tasks.values()
    for r in s.requirements
}.union({
  r for s in model.services.values()
    for r in s.requirements
})
missing_requirements = expected_requirements.
  difference(model_requirements)
if missing_requirements:
  print(f"Missing: {missing_requirements}")
  sys.exit(1)

def c2(model, expected_requirements):
model = Model.parse_file(model)
dags = load_dags_from_file(dag_file)
for dag_val in dags.values():
  for t_name, t_def in dag_val.task_dict.items():
    assert t_def.area == model.tasks[t_name].area
    for req in t_def.requirements:
      assert set(t_def.requirements) ==
        model.tasks[t_name].requirements
```

Workflow verification

```
def c3(model, dag_file):
model = Model.parse_file(model)
dags = load_dags_from_file(dag_file)
for dag_val in dags.values():
  for t_name, t_def in dag_val.task_dict.items():
    assert t_def.downstream_task_ids ==
      model.tasks[t_name].next
```

Release validation

```
def c8(docker_file, docker_compose_file):
if docker_file:
  subprocess.check_output([
    "docker", "build", "--file",
    docker_file, "-t", "test_image", "."])
  subprocess.check_output([
    "docker", "scan", "test_image"])
if docker_compose_file:
  subprocess.check_output([
    "docker-compose", "config", "-f",
     docker_compose_file])
```

Job verification

```
def c4(dag_file):
warnings = run_pylint(target_file=dag_file)
types = {warning["type"] for warning in warnings}
if "error" in types:
  pprint([ w for w in warnings
           if w["type"] == "error"])
  sys.exit(1)

def c5(dag_file, expected_urls):
dags = load_dags_from_file(dag_file)
sources = []
for dag_val in dags.values():
  for t_name, t_def in dag_val.task_dict.items():
    sources += get_sources(t_def)
urls = {url for src in sources
  for url in hdfs_paths_probe(src)}
unexpected_urls = expected_urls.differendce(urls)
if unexpected_urls:
  print("Unexpected URLs:")
  pprint(unexpected_urls)
  sys.exit(1)

def c6(dag_file, expected_urls):
dags = load_dags_from_file(dag_file)
regex = URL_REGEX
sources = []
for dag_val in dags.values():
  for t_name, t_def in dag_val.task_dict.items():
    sources += get_sources(t_def)
sources = list(filter(lambda e: e is not [], sources))
urls = {url for src in sources
  for url in re.findall(regex, src)
  if url is not []}
unexpected_urls = expected_urls.differendce(urls)
if unexpected_urls:
  print("Unexpected URLs:")
  pprint(unexpected_urls)
  sys.exit(1)

def c7(dag_file, dag_id, task_id, expected_args):
expected_args = dict(expected_args)
dags = load_dags_from_file(dag_file)
args = operator_args_extractor(dags[dag_id], task_id)
args = {k: str(v) for k, v in args.items()}
for k, v in expected_args.items():
  if args.get(k) != v:
    print(f"expected {k} to be {v}"
          f" but found {args.get(k)}")
    sys.exit(1)
```

Deployment validation

```
def c9(requirements, threshold):
kerberos_login("bertof/my.engine", "eng.keytab")
spark = spark_config_check(requirements)
print("Spark warnings:")
pprint(spark["warnings"])
if score < threshold:
  sys.exit(1)

def c10(requirements, threshold):
kerberos_login("bertof/my.engine", "eng.keytab")
airflow = airflow_config_check()
print("Airflow warnings:")
pprint(airflow["warnings"])
if score < threshold:
  sys.exit(1)

def c11(requirements, threshold):
kerberos_login("bertof/my.engine", "eng.keytab")
enc = hadoop_config_check_encryption(requirements)
sec = hadoop_config_check_security(requirements)
hadoop = merge_dicts(enc, sec)
hadoop["score"] = min(enc["score"], sec["score"])
print("Hadoop warnings:")
pprint(hadoop["warnings"])
if score < threshold:
  sys.exit(1)
print("Ok")
```

Infrastructure scan

```
def c12(threshold, openscap_conf, environment):
environment = dict(environment)
config = yaml.load(openscap_conf, yaml.FullLoader)
openscap_res = openscap_check(
  config=openscap_conf, environment=environment)
print("Openscap warnings:")
pprint(openscap_res["warnings"])
if openscap_res["score"] < threshold:
  sys.exit(1)

def c13(threshold, openvas_conf, environment):
environment = dict(environment)
config = yaml.load(openvas_conf, yaml.FullLoader)
openvas_res = openvas_check(
  config=openvas_conf, environment=environment)
print("Openvas warnings:")
pprint(openvas_res["warnings"])
if openvas_res["score"] < threshold:
  sys.exit(1)
```

TABLE III
CHECKS INCLUDED IN EACH PERFORMANCE SCENARIO

Type	Id	Time (s)	FD	PU	CC	IC	MI
Instance check	c_1	0.900	●	●			
	c_2	0.898	●	●			
Workflow verification	c_3	0.884	●	●			●
Job verification	c_4	3.946	●	●	●		
	c_5	0.905	●	●			●
	c_6	0.902	●	●			●
	c_7	0.888	●	●	●		●
Release validation	c_8	3.028	●	●			
Deployment validation	c_9	2.130	●		●	●	●
	c_{10}	1.130	●		●	●	●
	c_{11}	1.180	●		●	●	●
Infrastructure scan	c_{12}	4.500	●				●
	c_{13}	93.347	●				●

VI. RELATED WORK

Recent years have been characterized by intense research on big data and its protection, and on security assurance in general. Big data security concerns enriching and protecting big data computations with security techniques. Research on this topic emerged around 2012 [13], and it immediately become clear that, first and foremost, big data security must adequately protect stored data [13]. It then moved towards a larger definition of security embracing pipelines themselves with approaches still centered on data [14]. This translates into securing big data pipelines according to the CIA (*Confidentiality*, *Integrity*, *Availability*) triad [6], [7]. Scenarios involving high-sensitive data (e.g., healthcare) and the emergence of legal regulations such as General data Protection Regulation (GDPR) and California Consumer Privacy Act (CCPA) called for the inclusion of *privacy* in the triad [5], [7]. In general, security issues can occur and be counteracted at any step of the pipeline [15], and protection techniques must find an appropriate tradeoff between the quality of the analytics, the degree of security, as well as the performance of the analytics [5], and enforce legal compliance [4].

Security assurance is a suitable approach to address security and privacy concerns in big data. It follows the application of security countermeasures and aims to verify whether such countermeasures work in practice, enabling the big data pipeline to demonstrate a (set of) non-functional requirements [16]. The main techniques at the basis of non-functional requirements verification range from *testing* [17], [18] to *monitoring* [19], [20]. Other approaches include *Trusted Platform Modules* [21] and *security assurance cases* [22], to name but the most relevant.

To the best of our knowledge, very few solutions have been presented for assurance in big data-based systems. They in fact mostly address SLA compliance (e.g., [23], [24]), specific requirements (e.g., privacy [4]), DevOps integration without "Sec" (e.g., [25]), or DevSecOps integration without adequate emphasis on assurance (e.g. [26]). The approach in this paper addresses the confluence of big data pipelines developed in a DevSecOps fashion.

VII. Conclusions

Big data pipelines process and gain insights from a vast amount of critical data, whose scale poses additional challenges to the application of security and privacy countermeasures. For this reason, it is of paramount importance to ensure that such countermeasures and the whole pipeline work as expected. The assurance methodology in this paper provides the first boost in this direction, integrating assurance checks within a DevSecOps-based big data pipeline where the pipeline behavior in operation is continuously verified thanks to inserted assurance checks.

Acknowledgments

Some of the work reported here was part of the project "One Health Action Hub: University Task Force for the resilience of territorial ecosystems", funded by Università degli Studi di Milano - PSR 2021 - GSA - Linea 6 and by the program "piano sostegno alla ricerca". This research is also part of the project "MIND FoodS HUB (Milano Innovation District Food System Hub): Innovative concept for the eco-intensification of agricultural production and for the promotion of dietary patterns for human health and longevity through the creation in MIND of a digital Food System Hub", cofunded by POR FESR 2014-2020_BANDO Call HUB Ricerca e Innovazione, Regione Lombardia.

References

[1] G. D'Acquisto, J. Domingo-Ferrer, P. Kikiras, V. Torra, Y.-A. de Montjoye, and A. Bourka, "Privacy by design in big data: an overview of privacy enhancing technologies in the era of big data analytics," *ENISA*, 2015.

[2] W. L. Chang, "Big data interoperability framework: Volume 4, security and privacy," NIST, Tech. Rep., 2015.

[3] CSA, *Big Data Security and Privacy Handbook: 100 Best Practices in Big Data Security and Privacy*, 2016, https://cloudsecurityalliance.org/download/big-data-security-and-privacy-handbook/.

[4] L. Wang, J. P. Near, N. Somani, P. Gao, A. Low, D. Dao, and D. Song, "Data Capsule: A New Paradigm for Automatic Compliance with Data Privacy Regulations," in *Heterogeneous Data Management, Polystores, and Analytics for Healthcare*, 2019.

[5] H. Kupwade Patil and R. Seshadri, "Big Data Security and Privacy Issues in Healthcare," in *Proc. of IEEE Big Data 2014*, Anchorage, AK, USA, Jun. 2014.

[6] L. Zhou, A. Fu, S. Yu, M. Su, and B. Kuang, "Data integrity verification of the outsourced big data in the cloud environment: A survey," *Journal of Network and Computer Applications*, vol. 122, 2018.

[7] E. Bertino and E. Ferrari, *Big Data Security and Privacy*. Springer, 2018.

[8] X. Zhang, L. Qi, W. Dou, Q. He, C. Leckie, K. Ramamohanarao, and Z. Salcic, "Mrmondrian: Scalable multidimensional anonymisation for big data privacy preservation," *IEEE TBD*, 2017.

[9] M. Anisetti, C. A. Ardagna, F. Gaudenzi, and E. Damiani, "A continuous certification methodology for devops," in *Proc. of ACM MEDES 2019*, Limassol, Cyprus, November 2019.

[10] C. Ardagna, V. Bellandi, M. Bezzi, P. Ceravolo, E. Damiani, and C. Hebert, "Model-based big data analytics-as-a-service: Take big data to the next level," *IEEE TSC*, 2018.

[11] H. Myrbakken and R. Colomo-Palacios, "DevSecOps: A Multivocal Literature Review," in *Proc. of SPICE 2017*, Palma de Mallorca, Spain, October 2017.

[12] M. Anisetti, C. A. Ardagna, E. Damiani, N. E. Ioini, and F. Gaudenzi, "Modeling time, probability, and configuration constraints for continuous cloud service certification," *Computers & Security*, vol. 72, pp. 234–254, 2018.

[13] C. Tankard, "Big data security," *Network Security*, vol. 2012, no. 7, Jul. 2012.

[14] Y. Demchenko, C. Ngo, C. de Laat, P. Membrey, and D. Gordijenko, "Big Security for Big Data: Addressing Security Challenges for the Big Data Infrastructure," in *Proc. of SDM 2013*, Trento, Italy, August 2013, workshop at VLDB 2013.

[15] A. Mehmood, I. Natgunanathan, Y. Xiang, G. Hua, and S. Guo, "Protection of Big Data Privacy," *IEEE Access*, vol. 4, 2016.

[16] C. Ardagna, R. Asal, E. Damiani, and Q. Vu, "From security to assurance in the cloud: A survey," *ACM CSUR*, vol. 48, no. 1, August 2015.

[17] P. Stephanow, G. Srivastava, and J. Schütte, "Test-Based Cloud Service Certification of Opportunistic Providers," in *Proc. of IEEE CLOUD 2016*, San Francisco, CA, USA, June, July 2016.

[18] M. Anisetti, C. Ardagna, E. Damiani, and F. Gaudenzi, "A semi-automatic and trustworthy scheme for continuous cloud service certification," *IEEE Transactions on Services Computing (TSC)*, vol. 13, no. 1, 2020.

[19] A. Ciuffoletti, "Application level interface for a cloud monitoring service," *Computer Standards & Interfaces*, vol. 46, 2016.

[20] S. Lins, S. Schneider, J. Szefer, S. Ibraheem, and A. Ali, "Designing Monitoring Systems for Continuous Certification of Cloud Services: Deriving Meta-requirements and Design Guidelines," *Comm. of the Association for Information Systems*, vol. 44, 2019.

[21] M. Aslam, B. Mohsin, A. Nasir, and S. Raza, "FoNAC - An automated Fog Node Audit and Certification scheme," *Computers & Security*, vol. 93, June 2020.

[22] R. Calinescu, D. Weyns, S. Gerasimou, M. U. Iftikhar, I. Habli, and T. Kelly, "Engineering Trustworthy Self-Adaptive Software with Dynamic Assurance Cases," *IEEE Transactions on Software Engineering*, vol. 44, no. 11, 2018.

[23] C. A. Ardagna, E. Damiani, M. Krotsiani, C. Kloukinas, and G. Spanoudakis, "Big data assurance evaluation: An sla-based approach," in *Proc. of IEEE SCC 2018*, San Francisco, CA, USA, July 2018.

[24] X. Zeng, S. Garg, M. Barika, A. Y. Zomaya, L. Wang, M. Villari, D. Chen, and R. Ranjan, "SLA Management for Big Data Analytical Applications in Clouds: A Taxonomy Study," *ACM Computing Surveys*, vol. 53, no. 3, Jun. 2020.

[25] C. Castellanos, C. A. Varela, and D. Correal, "ACCORDANT: A domain specific-model and DevOps approach for big data analytics architectures," *Journal of Systems and Software*, vol. 172, 2021.

[26] R. Kumar and R. Goyal, "When Security Meets Velocity: Modeling Continuous Security for Cloud Applications using DevSecOps," in *Proc. of ICIDCA 2020*, Coimbatore, India, September 2020.