Internet of Things 16 (2021) 100440

Contents lists available at ScienceDirect

Internet of Things

journal homepage: www.elsevier.com/locate/iot

Research article

Conceptualization and scalable execution of big data workflows using domain-specific languages and software containers

Nikolay Nikolov [a,*], Yared Dejene Dessalk [b], Akif Quddus Khan [c], Ahmet Soylu [d], Mihhail Matskin [b], Amir H. Payberah [b], Dumitru Roman [a]

[a] SINTEF AS, Forskningsveien 1, 0373 Oslo, Norway
[b] KTH Royal Institute of Technology, Brinellvägen 8, 114 28 Stockholm, Sweden
[c] Norwegian University of Science and Technology — NTNU, Teknologivegen 22, 2815 Gjøvik, Norway
[d] OsloMet — Oslo Metropolitan University, Pilestredet 46, 0167 Oslo, Norway

ARTICLE INFO

Keywords:
Big data workflows
Internet of Things
Domain-specific languages
Software containers

ABSTRACT

Big Data processing, especially with the increasing proliferation of Internet of Things (IoT) technologies and convergence of IoT, edge and cloud computing technologies, involves handling massive and complex data sets on heterogeneous resources and incorporating different tools, frameworks, and processes to help organizations make sense of their data collected from various sources. This set of operations, referred to as Big Data workflows, requires taking advantage of Cloud infrastructures' elasticity for scalability. In this article, we present the design and prototype implementation of a Big Data workflow approach based on the use of software container technologies, message-oriented middleware (MOM), and a domain-specific language (DSL) to enable highly scalable workflow execution and abstract workflow definition. We demonstrate our system in a use case and a set of experiments that show the practical applicability of the proposed approach for the specification and scalable execution of Big Data workflows. Furthermore, we compare our proposed approach's scalability with that of Argo Workflows – one of the most prominent tools in the area of Big Data workflows – and provide a qualitative evaluation of the proposed DSL and overall approach with respect to the existing literature.

1. Introduction

Massive amounts of data is being generated especially with the rise of Internet of Things (IoT) technologies creating new value creation opportunities through Big Data analysis. Accordingly, Big Data analysis has been a driving factor in revolutionizing major sectors, such as mobile services, finance, and scientific research. *Big Data workflows* are composed of multiple orchestrated steps, such as workflow activities that perform various data analytical tasks. They are different from business and scientific workflows since they are dynamic, process heterogeneous data, and are executed in parallel instead of a sequential set of scientific operators [1]. Although many organizations recognize the significance of Big Data analysis, they still face critical challenges when implementing data analytics into their process [2]. Firstly, multiple experts, ranging from technical to domain experts, need to be involved in specifying such complex workflows. Secondly, given the fact that IoT, Edge and Cloud technologies converge towards a computing continuum,

* Corresponding author.
E-mail addresses: nikolay.nikolov@sintef.no (N. Nikolov), yaredd@kth.se (Y.D. Dessalk), akif.q.khan@ntnu.no (A.Q. Khan), ahmet.soylu@oslomet.no (A. Soylu), misha@kth.se (M. Matskin), payberah@kth.se (A.H. Payberah), dumitru.roman@sintef.no (D. Roman).

https://doi.org/10.1016/j.iot.2021.100440
Received 14 February 2021; Received in revised form 27 July 2021; Accepted 29 July 2021
Available online 8 August 2021

workflow steps need to be mapped dynamically to heterogeneous computing and storage resources to ensure scalability [3,4]. Providing a scalable, general-purpose solution for Big Data workflows that a broad audience can use is an open research issue [2,4].

The challenges in devising an applicable generalized solution come from the fact that bottlenecks can occur on an individual workflow step level – for example, when the throughput of one step is lower than the others. Thus, scaling up the entire workflow does not address the scalability issues and needs to be done on the individual step level. This issue becomes worse by the fact that scalability needs to be organized and orchestrated over heterogeneous computing resources. Furthermore, scaling up individual steps introduces race conditions between step instances that attempt to process the same piece of data simultaneously. Another major challenge is achieving usability by multiple stakeholders as most Big Data processing solutions are focused on ad-hoc processing models that only trained professionals can use. However, organizations typically operate on specific software stacks, and getting experts in Big Data technology can introduce costs that are not affordable or practical. Even if an organization has the necessary technical personnel, data workflow steps pertain to specific domain-dependent knowledge, which is possessed by the domain experts rather than the data scientists who set up the data workflows. In this respect, this work aims to provide an approach that allows:

(a) Conceptualization of Big Data workflows using a domain-specific language (DSL) to support the high-level definition of complex data processing across multiple types of parameters, inputs, and outputs; and
(b) Scalable execution of Big Data workflows using software container technologies and message-oriented middleware solutions.

In this article, we present the design and implementation of a Big Data workflow approach based on the use of software container technologies, message-oriented middleware (MOM), and a DSL to enable highly scalable workflow execution and abstract workflow definition [5]. Our design allows for scaling up on the level of individual workflow steps on top of heterogeneous infrastructures while avoiding race conditions through a system of inter- and intra-step coordination. Furthermore, our container-based approach allows for the separation of concerns between the stakeholders by providing a flexible means of injecting domain-specific code involving any programming language and enabling the definition of workflows on a high level (i.e., without the step-specific code). Finally, the DSL allows easy specification of Big Data workflows by abstracting low-level technical aspects. We demonstrate our approach's applicability by implementing a prototype based on a real-world data workflow and multiple experiments showing satisfactory performance. Furthermore, a set of comparative experiments with Argo Workflows shows better performance due to concurrent workflow execution. A qualitative evaluation of the overall approach and the DSL with respect to the existing literature presents our approach's benefits. This article extends our previous work in [5] by providing (i) more insights and explanations about the motivation, approach and solution details, (ii) an extended presentation and analysis of the related work, (iii) an elaborate account and analysis of the requirements for enabling Big Data workflows on the Computing Continuum, (iv) a qualitative evaluation and discussion of the proposed DSL, and (v) a set of examples of real-life use cases of the approach.

The rest of the article is organized as follows. Section 2 sets the background, while Section 3 discusses related work. Section 4 describes the requirements and Section 5 presents the proposed approach. Section 6 discusses our proof-of-concept implementation based on the proposed design, Section 7 provides an evaluation, and, finally Section 8 concludes the article and discusses possible future work.

2. Background

In this section we briefly introduce technological background relevant in the context of scalable Big Data workflows execution.

2.1. Big data workflows

A Big Data workflow is the computerized modeling and automation of a process consisting of multiple orchestrated steps that perform various data analysis tasks [6]. In practice, most Big Data workflows are usually represented by a Directed Acyclic Graph (DAG) [7]. Various processing models can be applied for parallelizing data processing known as workflow data patterns [8]. In this context, Pipe and Filter (P&F) is a relevant architectural design to decompose a larger processing task into a series of smaller, separate processing steps (filters) that are connected by channels (pipes) [9]. The filters can then be integrated into a workflow, whereby each filter receives and sends data in a standardized way, thus implementing the "shared data passed by reference" pattern [8]. This pattern, given that different steps are loosely coupled, enables scalability at the workflow step level, but introduces the issue of handling concurrency control.

2.2. Message-oriented middleware

Achieving race-condition-free consistency and concurrency for scaling the homogeneous Big Data workflow steps requires using a synchronization mechanism across the different step instances. One approach for addressing such synchronization issues is to use Message-Oriented Middleware (MOM). MOM provides an infrastructure for loosely-coupled and asynchronous inter-process communication using messaging capabilities [10]. In a system integrated using MOM, a client can send messages to and receive from the other clients of the messaging system (without losses or message duplication) in a race-condition-free way through the use of a message queue. Thus, in the context of Big Data workflows, the middleware can act as a medium for communication, whereby step instances coordinate passing intermediate results through sending/receiving messages in MOM queues.

2.3. Container technology

A container is a packaged, standalone, deployable collection of program elements [11]. Containers provide an isolated virtual environment and include all the required dependencies of the provisioned tools. Docker is one of the most well-known platforms for organizing solutions based on container technologies. When executing multiple containers, a container orchestration system can manage their deployment, scaling, and networking. In particular, container technology is useful for deploying distributed scalable applications (such as Big Data workflows), as it provides transparent means for infrastructure management and easy scalability of individual application sub-components. Orchestration tools use a configuration file to define container images, network, and related deployment schemes of an application.

2.4. Domain-specific languages

DSLs are programming languages or specification languages that target a specific problem domain. DSLs are small, descriptive, and contain only the details needed for the desired domain. In general, there are two types of DSLs: internal and external [12]. An internal DSL is a specific form of Application Programming Interface (API) in a host general-purpose language (GPL). In contrast, an external DSL is a language that is parsed independently of the host GPL. Unlike GPLs, DSLs are limited in scope and cannot cover all aspects of a given problem. However, they are more effective than GPLs in providing expressiveness at the cost of generality. DSLs offer better domain-specificity and significantly improve collaboration between domain experts and developers [13].

3. Related work

In this section, related work is presented in terms of related scientific literature and commercial and non-commercial software tools. Aspects such as workflow resource scheduling and others, as described in [4,14], are complementary to our approach as the described concept uses container orchestration systems to specify them.

3.1. Related software tools

There is a large variety of Big Data workflow solutions that share similar design principles while fulfilling the needs of various groups of users and use cases. We carried out a comparative analysis of the most promising workflow tools (chosen based on their mass user base and relevance), including Pachyderm,[1] Apache Airflow,[2] Snakemake,[3] Apache NiFi,[4] Node-RED,[5] Argo Workflows,[6] NextFlow,[7] and Conductor.[8] An overview of the comparison is presented in Table 1 with respect to workflow type, usability for non-technical experts, run time container support, and generality of the solution.

We consider two main types of workflows — *scientific* and *general-purpose*. *Scientific* workflow approaches are built to be applied in homogeneous infrastructures, such as High-Performance Computing (HPC) clusters, where resources are shared between multiple workflows and dedicated algorithms are used to optimize job scheduling and resource allocation. On the other hand, *general-purpose* workflow tools are applicable in varying domains/scenarios and can be executed in heterogeneous computing infrastructures. In terms of *usability* for non-technical experts, we consider three levels of support — *easy*, *medium*, and *difficult*. We regard approaches that provide a dedicated DSL/UI that caters to domain experts as *easy*. If the approach requires the use of coding of the Big Data workflows in a specific programming language, we consider the approach of *medium* usability. On the other hand, if the approach requires coding and the use of approach-specific concepts and libraries in order to declare a workflow, we consider the approach to be *hard* in terms of usability to non-technical experts. With respect to *container support*, we classify the approaches on whether or not they support the use of container technologies (e.g., Docker) for encapsulating the entire Big Data workflow or individual steps. We note that although some of the tools, such as Apache NiFi, are packaged and deployable using container technology,[9] but we do not consider them to provide *container support*. We consider the solutions in terms of whether or not they are generic or catered for a specific vertical domain of knowledge. Thereby, we regard approaches that are applicable in any vertical domain as *generic* and vice versa. Finally, in terms of monitoring capabilities, we classify workflow tools by the availability of monitoring execution of the workflow. Depending on how the execution can be monitored, it can be further classified into *logging* and *runtime*. We regard those tools that provide real-time monitoring as *runtime*. If the monitoring is based on logs at the end of execution, we classify the tools as *logging*. Tools that mainly provide logging-based monitoring, which also support limited real time monitoring capabilities, are regarded as *logging and (partial) runtime*.

[1] https://www.pachyderm.com.
[2] https://airflow.apache.org.
[3] https://snakemake.readthedocs.io.
[4] https://nifi.apache.org.
[5] https://nodered.org.
[6] https://argoproj.github.io/argo.
[7] https://www.nextflow.io.
[8] https://netflix.github.io/conductor.
[9] https://hub.docker.com/r/apache/nifi.

Table 1
Comparison matrix for tools supporting Big Data workflows.

Tool name	Workflow type	Usability	Container support	Generic solution	Monitoring
Airflow	General-purpose	Difficult	Docker	Yes	Logging and (partial) runtime
Argo	General-purpose	Difficult	Docker	Yes	Logging and (partial) runtime
Conductor	General-purpose	Difficult	No	Yes	Logging
Nextflow	Scientific	Medium	Docker	No	Runtime
NiFi	General-purpose	Medium	No	No	Runtime
Node-RED	General-purpose	Medium	No	No	Logging
Snakemake	General-purpose	Medium	Docker	Yes	Runtime
Pachyderm	General-purpose	Difficult	Docker	No	Runtime
Our approach	*General-purpose*	*Easy*	*Docker*	*Yes*	*Runtime*

Argo Workflows natively supports containers in workflows by implementing each step as a container. The workflow definition and automation are done by YAML templates based on a custom DSL.[10] The steps can be arranged either sequentially or in a DAG, making it possible to orchestrate and parallelize jobs. However, Argo Workflows does not have a middleware solution to handle inter-step communication, which may result in step instances running into race conditions when scaled horizontally. Additionally, individual steps cannot be scaled up in order to increase the workflow throughput (although they can be run in parallel).

Nextflow is a workflow framework based on the dataflow paradigm [15]. It uses a declarative processing model to execute parallel tasks and supports step-level scalability. Nextflow has built-in support for container technologies and the communication among processes is handled using channels, and asynchronous First-In-First-Out (FIFO) queues preventing race conditions. Nextflow provides a custom DSL to write complex workflows, which is an extension to Apache Groovy.[11] Nextflow does not provide a clear separation of concerns between workflow definition and implementation and relies on a specific software stack (through the DSL) for workflow step implementation.

Apache Airflow is a platform for the creation, scheduling, and monitoring of data workflows. Python scripts are used to describe DAG structured workflows. Airflow has a scheduler that executes workflows on a set of workers, but it lacks a mechanism to avoid race-conditions when scaled. Airflow has a rich user interface support allowing users to visualize workflows and monitor their execution. The workflow definition is done via programming. Pachyderm is a tool for managing data workflow and related input/output data that results in all the data workflows' reproducibility and scalability. Pachyderm is based on Docker and Kubernetes, and provides advanced features such as pluralization and incremental processing. Users need technical knowledge to define workflows.

Snakemake is a workflow system in which workflows are defined in terms of rules presenting the input–output conversion of files. Determining dependencies between rules, Snakemake automatically forms parallelizable DAG workflows. It provides a concise and readable DSL, an extension of Python programming language, for defining rules and workflow specific properties. It was initially developed for scalable workflows in bioinformatics. Apache NiFi is a project of Apache Software Foundations that allows automation of data flow between systems. It is based on a flow-based programming model for building scalable data workflows. Although it comes with a user interface, it requires technical knowledge to design workflows.

Node-RED is another flow-based programming workflow tool that is built based on Node.js run-time. It follows the event-driven and non-blocking model. It provides a Web-based visual editor for designing workflows, but it still requires technical knowledge from the user. Even though Node-RED was initially designed for Internet-of-Things (IoT) applications, it has evolved to develop various applications. Conductor is a workflow orchestration engine by Netflix that allows creating microservice-based business and process workflows. Workflows are composed of tasks that are executed by remote workers. A JSON-based DSL is used to define workflows.

3.2. Related studies

Many efforts have been made to use containers to address the challenges of scalability, resource provisioning, scheduling, orchestration, and data management of data workflows. Authors in [16] propose an approach for decomposing scientific workflows into micro-units that are containerized and contain sub-workflows that communicate through streaming middleware among each other and other applications and devices. This approach is specific to the domain of digital twins and addresses the job dependency, job scheduling, and streaming support issues, but does not provide means of independent scaling of steps and is domain-specific.

Another set of approaches relies on the use of container technology to deploy workers that execute jobs. Authors in [7] use Docker to deploy a software stack to homogenize the environment where tasks are run. However, their approach has limitations on the number of workflow step containers deployed on a single host and does not support long-running tasks (i.e., containers are shut down after executing). The approach in [17] uses containers to encapsulate workers that contain the workflow engine and execution environment but is applicable only in the context of workflows expressed in a specific stream-based dataflow DSL [18], which reduces applicability for general-purpose data workflows.

[10] https://argoproj.github.io/argo-workflows/fields.

[11] https://groovy-lang.org.

Another set of data workflow implementations in the area of HPC are built on top of the Shifter [19] framework. Authors in [20] use the framework to distribute jobs and entire workflows (encapsulated in a single container) over an HPC cluster. This approach does not support individual steps' scalability but views data workflows as entire units of work. Authors in [21] rely on Shifter for defining virtual HPC clusters to run jobs. Thereby, containers are used for creating worker nodes with the necessary HPC functionality and for managing the jobs, but their approach is not applicable for general-purpose data workflows.

One approach that comes close to the one described in this paper is presented in [1], whereby containers are used to wrap individual steps. The framework uses a TOSCA [22] to describe both the deployment and workflow steps, which containers can implement. However, the framework does not support dynamic workflows — tasks are executed in a sequential manner, whereby a task must finish for next to be deployed, which makes the approach not suitable for long-running workflows.

Finally, authors in [23] use Cloud orchestration for deploying distributed workflows in a similar manner as the one described here, although containers are not supported. However, the approach does not support run-time scalability and relies on the step definition to manage the input and output between the steps. Furthermore, the approach uses a DSL that provides no clear separation of concerns between the design- and run-time phases of data workflows.

4. Requirements

Commonly used data processing frameworks (such as Spark, Flink, Beam) are designed with ad-hoc processing models that technical experts on specific technology stacks can only use. The focus is on the programming and run-time aspects of workflows than the actual definition of workflows themselves. Even though some solutions, as discussed in related work, have demonstrated defining and executing Big Data workflows, they do not cater to the needs of domain experts. Solutions like Pachyderm, Snakemake, and Airflow are merely designed for technical experts, and the definition of workflows is done using high-level programming and scripting languages. To enable domain experts to participate in the process, some tools provide a user-friendly interface to define workflows. However, they either are made for a specific application domain (e.g., bioinformatics, computational chemistry, ecology, genomics, etc.) or require some level of technical knowledge to manipulate the data. Thus, although most approaches and tools use some form of DSL or UI, the abstraction level is not sufficient to allow for separation of concerns between definition and implementation, which is necessary to effectively involve domain experts in workflow definition.

Another major challenge in Big Data workflows is the dynamic mapping workflow steps to heterogeneous computing and storage resources to ensure scalability. This challenge comes from the fact that bottlenecks can occur on an individual workflow step level, e.g., the throughput of one step is lower than others. Thus, scaling up the entire workflow does not solve the scalability issues and needs to be done on the individual workflow step level. This issue is exacerbated by the fact that the scalability needs to be organized and orchestrated over heterogeneous computing resources. Achieving scalability in Big Data workflows has another dimension of challenges: exchanging data among workflow components and race conditions when multiple instances of workflow components try to modify a shared resource (e.g., a file) at the same time. Multiple workflow step instances can make up a large set of capabilities to deliver the workflow's needs. Some approaches and tools discussed in the related work attempt to address this issue through the use of containers and different types of middleware. However, no approach is able to unlock the full potential for achieving step-level scalability of workflows, which relies on workflow and step encapsulation.

Finally, in Big Data workflows, data need to be passed between the workflow components so that the communication overhead is minimal. A communication solution must decouple the communication between the steps to be scaled up while maintaining race-condition-free data access. Additionally, this communication module has to play a central role in determining data flow between workflow steps. Based on our analysis of the state of the art, we find there is no holistic Big Data workflow solution that can provide such a communication module along with high-level workflow definition and scalable execution.

Accordingly we extracted the following requirements for our Big Data Workflow solution:

(a) A workflow definition mechanism with a clear separation between design- and run-time aspects and not limited to a specific technology stack, application domain or ad-hoc processing models;
(b) A workflow run-time support that considers workflows as separate units, rather than as a single unit, for individual workflow steps; and
(c) A workflow enactment approach with event driven execution and support for race-condition-free parallel execution.

5. Proposed solution

In this section, we propose an approach for the workflow step design and inter-step communication. For the description of the Big Data Workflow we use the DSL from [24].

5.1. Overall architecture

The solution enables various stakeholders to be involved in the creation of Big Data workflows. The desired properties of the system are achieved by utilizing container and orchestration technologies and a DSL. The workflow system is composed of three components: *Workflow Modeling Manager*, *Deployment Service Runtime*, and *Data Storage/Sharing Ecosystem* (see Fig. 1).

The Workflow Modeling Manager is the central element for defining workflow steps and composing them in workflows. It comprises a set of tools and configurations that allow the formation of deployable Big Data workflows. The component uses models that provide

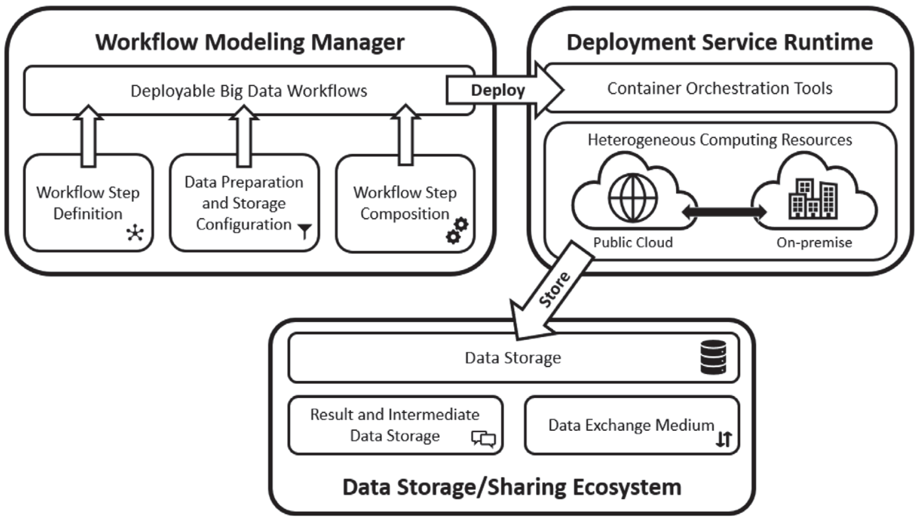
Workflow Modeling Manager
Deployable Big Data Workflows
Workflow Step Definition
Data Preparation and Storage Configuration
Workflow Step Composition
Deploy
Deployment Service Runtime
Container Orchestration Tools
Heterogeneous Computing Resources
Public Cloud
On-premise
Store
Data Storage
Result and Intermediate Data Storage
Data Exchange Medium
Data Storage/Sharing Ecosystem

Fig. 1. Overview of the our workflow system.

high-level descriptions of workflow steps and their dependencies. This component handles storage configurations, data preparation, and step-level data processing and transformation operations. The output of the component is a deployable data workflow. The *Deployment Service Runtime* is a component representing the collection of hybrid computing resources where workflows steps are deployed. As individual workflow steps are wrapped as containers, container orchestration tools play an important role in managing the heterogeneous resources and allowing workflows step containers to be deployed. The component also controls operations such as scaling and load balancing. *Data Storage/Sharing Ecosystem* is responsible for storing intermediate and output data and the data exchange mechanism during workflow execution. As workflow steps are deployed across heterogeneous distributed environments, the data exchange mechanism is an essential element that binds the workflow steps together by allowing them to pass data.

The system ensures the separation of design-time and run-time aspects of the workflows, i.e., workflow definition is done without considering the run-time execution. This allows having a separation of concerns among the involved stakeholders. Thereby, domain-experts can be responsible for extracting data processing requirements and structuring the high-level design of workflow steps. Technical experts provide the concrete programmatical implementations of the steps — for example, data scientists may provide workflow step-specific analytical models and data preparation code. Finally, DataOps experts are engaged in deploying and maintaining data workflows in production settings and monitoring data quality and related infrastructure status.

5.2. Big data workflow description

The DSL used in this work [24] for representing Big Data workflows is inspired by [25] and is shown in Fig. 2. The domain conceptualized by the DSL is container-based Big Data workflows, which is reflected in the choice of concepts. The main concepts of the metamodel are used to represent the different aspects of the *Workflow Modeling Manager* element shown in Fig. 1. *Workflow Step Definition* is implemented by the *Workflow* concept, which is comprised of a set of *Steps* in the DSL. Additionally, the element *Data Preparation and Storage Configuration* is implemented by the concepts *Parameter*, *Trigger* and *Communication Medium*, whereas the *Workflow Step Composition* element relates to the *Step Implementation* concept in the metamodel.

A *Workflow* is a sequence of steps that need to be executed in some order to process a set of data. It represents the conceptualized series of steps that perform different data ingestion, transformation, and analytics tasks. A given workflow can be defined by reusing another workflow. Besides the steps, a workflow is composed of a communication medium and a set of parameters. The *Steps* are the building blocks of a workflow, and each step corresponds to a single unit of data processing work in the workflow. The steps in a workflow are executed independently and are isolated from each other. Further, a step can have various options for its implementation and trigger mechanisms for executions.

A *Parameter* represents an input configuration value needed for the execution of a specific workflow instance. In addition to workflows, parameters can also be used to define configurations for workflow steps. The *Trigger* concept represents how the execution of a step instance is instantiated. In our metamodel, step execution can be triggered from a schedule that runs in a fixed interval of time, or it can be configured to run only once (e.g., during initialization of the workflow). Execution can also be triggered by an external event (e.g., invocation from a REST API or availability of input data in a message queue).

The *Step Implementation* is a concept that represents how the actual implementation of a workflow step is performed. A container-based implementation can be considered as an example of step implementation. In this way, the DSL provides explicit support for the container-based Big Data workflow approach described in the rest of this paper and additionally allows for other implementations that may not necessarily make use of containers. The *communication Medium* represents the mechanism in which workflow steps exchange data. A workflow step can pass data to another step using a message queue. Nevertheless, our approach allows that data exchange be performed using other means such as distributed file systems or Web services.

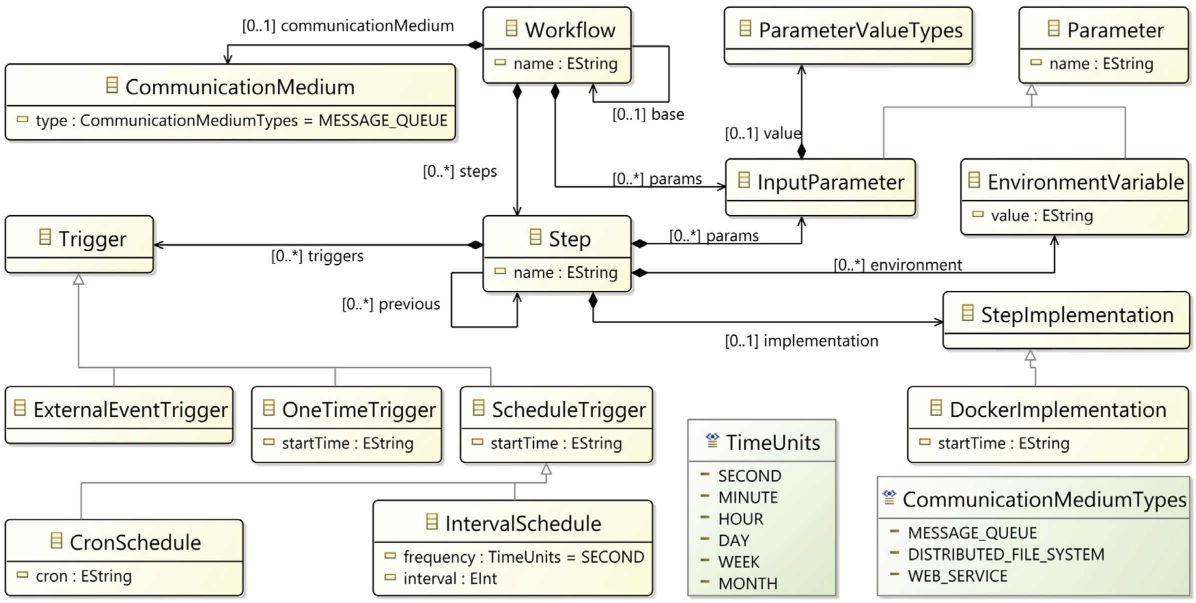

Fig. 2. Metamodel for the proposed DSL for representing Big Data workflows.

The design of the DSL emphasizes separation of concerns between the structural and implementation details of a workflow (e.g., through the clear separation between *Step* usage in workflow and *Step Implementation*). These aspects correspond to the domains of concerns of the two main groups of stakeholders — non-technical personnel (e.g., experts in a vertical domain) and technical experts (e.g., programmers). In that way, the DSL satisfies the requirement discussed in Section 4). Furthermore, we assume that the DSL should be used by non-technical experts and, therefore, should avoid complex constructions as much as possible. Therefore, another approach we apply in the language design is introduction of language concepts only if necessary. This means that only concepts used in our practical cases are supported in this first version of the DSL. Thus, constructs, such as loops and conditional steps have not been introduced into the DSL presented in this paper. The introduction of more advanced control structures is part of ongoing work in the context of the DataCloud project.[12]

5.3. Step design

Our approach takes advantage of container technology [26] in order to implement step encapsulation (in the DSL, this is reflected in the *Docker Implementation* sub-class of the *Step Implementation* concept shown in Fig. 2). The implementation of the approach implies wrapping workflow steps as containers and having step containers run independently of each other and in parallel. Moreover, step templates (container images) are downloaded once, and multiple instances of the same template can be deployed.

To enable the definition and deployment of workflow steps and their composition into a workflow, workflow steps are derived from a generic workflow step template supporting multiple programming languages. A workflow step can be prepared by customizing the generic template according to the need of the step and other settings. To achieve such flexibility, this template needs to be designed so that it is easy to introduce customization. The step template is composed of three main components: *Input Processing*, *Workflow Step Action*, and *Output Processing* (see Fig. 3).

The *Input Processing* is responsible for handling incoming data; this includes fetching data from remote sources (e.g., copying or downloading a file from shared file volumes and moving the data to the step workspace, where it will be processed). Based on the step's configuration, input data can be fetched once at a container startup or scheduled to poll for the data availability at a specific time interval. The Input Processing can be triggered when a piece of data is available at the source. The *Workflow Step Action* is a wrapper component for step-specific data processing code. This component allows the injection of custom code using different programming languages. The data fetched by the *Input Processing* component is processed using the step-specific code in the step. The *Output Processing* component is responsible for delivering the processed data to a specific destination (e.g., upload to a remote source or move it to a shared volume for further processing by next steps) and notifying that the processed data is available for the next steps. Output Processing also includes the clearing up of temporary and input data from the step workspace. Configuration and attributes of a workflow step can be expressed as parameters and injected at the deployment time. The step parameters are accessible only by the corresponding step, but workflow-level parameters can be defined as well.

[12] https://datacloudproject.eu.

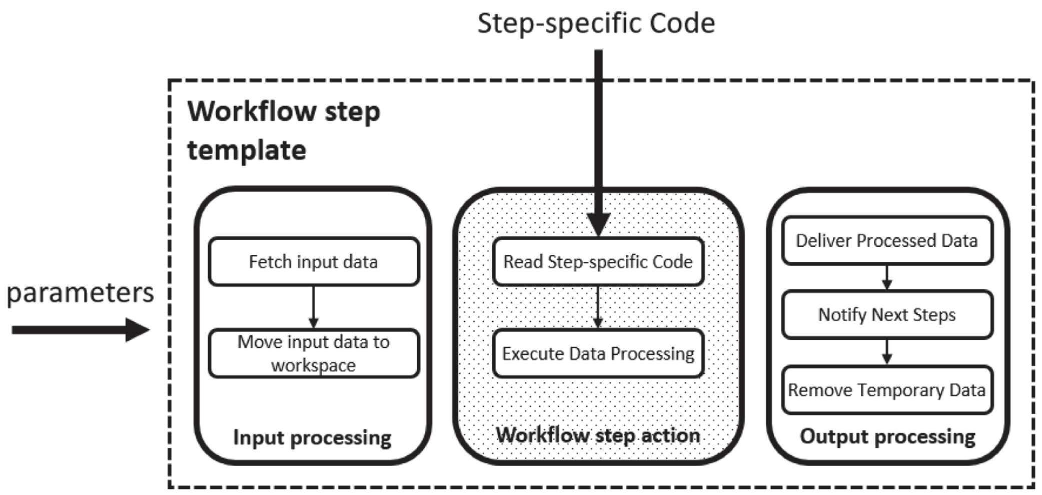

Fig. 3. Components of workflow step template.

5.4. Inter-step communication

The step design approach described in the previous section implements loose coupling of steps that comprise a workflow. However, in order to ensure that data between steps are transmitted consistently and correctly, consecutive steps need to communicate to notify each other of data availability. This communication corresponds to the *Fetch input data* and *Notify Next Steps* sub-processes of the *Input and Output processing* components in Fig. 3. In the proposed workflow approach, MOM serves as a medium for workflow step communication. Two inter-dependent workflow steps communicate by passing data through MOM without direct interaction. The sender step pushes data to the MOM so that it is consumed by the receiver step at any time after the data becomes available in the MOM. The two steps do not need to run simultaneously for interaction, ensuring temporal decoupling. The space decoupling is also achieved since none of the sequential steps needs to know the other nor how many other steps are in the workflow. Since workflow steps are loosely coupled, they can be scaled independently. Therefore, it is possible to assign more instances to bottleneck workflow steps that are, for example, more computationally heavy and reduce the overall processing time.

Specifically, message queues are used as a communication medium so that two inter-dependent workflow steps can share a queue to exchange data asynchronously. Both workflow step processes do not necessarily need to be running simultaneously to interact with the queue. Additionally, the messaging system is capable of providing an exactly-once message delivery guarantee. Furthermore, MOM-based communication ensures that the workflow does not run into race conditions. These can be seen in two scenarios:

- A workflow step that receives data does not access the data while the predecessor step is writing it. Message queues ensure that sending a message to a queue is independent of when it enters the queue and when the receiver reads it; and
- When a workflow step is scaled up, a step instance does not process a piece of data already being processed. This problem is avoided since a message is delivered to only a single step instance, i.e., data access is only for a specific instance.

The data patterns of the designed workflow are characterized according to the major workflow data patterns described in [27]. From a data visibility perspective, data elements are accessible from all workflow steps (e.g., data can be stored in shared file storage). Though a data element is accessible for all, it can only be used by a single workflow step instance at a time (e.g., when a reference of a file is transferred to the step from a message queue). Internally, data interaction happens only between two workflow step instances. Depending on the workflow step's purpose, it can also interact with an external source (e.g., by downloading a file from a remote source or invoking an external API endpoint). Data transfer is undertaken by the reference to the data element in some shared location (e.g., a file can be stored in a shared location, and its reference is exchanged over a message queue). In this case, data locking is not required; message queues restrict concurrent access to the data element. In terms of data-based routing, a given workflow step is executed whenever data are available at a network location.

6. Prototype implementation

To demonstrate the applicability of the proposed solution, we developed a prototype Big Data workflow (see Fig. 4) that implements the design choices described in Section 5 (available on GitHub[13] including a small anatomized sample of the used data). The prototype workflow comes from the domain of digital marketing (see [28] for a detailed description) and was chosen

[13] https://github.com/SINTEF-9012/ebw-prototype.

such that some of its steps have higher compute requirements than others. Those steps need to be assigned with more computing resources than other workflow steps to enable faster data processing. This makes it possible to demonstrate the applicability of step-level scalability. The prototype workflow demonstrates the implementation of Big Data workflows using container technology (encapsulating individual steps of the workflow), a messaging system (Message Queue), shared file system volumes (can be local or distributed file system), and includes the following steps: (i) extract tab-separated values (TSV) files from an archive file stored on a volume in a shared file system, (ii) convert TSV files to comma-separated values (CSV) files, (iii) split CSV files into smaller pieces if the number of rows in the files is above a certain number, (iv) clean and pre-process CSV files, (v) and convert tabular CSV files to JSON collections for further storage. The DSL description of the prototype workflow using an Xtext-based[14] grammar specification over our DSL model is given in Listing 1.

```
workflow prototypeWorkflow {
    communicationMedium: medium MESSAGE_QUEUE
    parameters: MQ_HOST = kubemq
    steps:
        - step unzip
            triggers: external-event
            implementation:
                docker-implementation image: '/ebw-prototype-00-unzip'
            environment:
                STEP_NAME='00-unzip'

        - step tsv2csv
            triggers: external-event
            implementation:
                docker-implementation image: '/ebw-prototype-01-tsv2csv'
            environment:
                STEP_NAME='01-tsv2csv'

        - step split
            triggers: external-event
            implementation:
                docker-implementation image: '/ebw-prototype-02-split'
            environment:
                STEP_NAME='02-split'

        - step transform
            triggers: external-event
            implementation:
                docker-implementation image: 'ebw-prototype-03-transform'
            parameters: tranformationJar = '/transformation/transformation.jar'
            environment:
                STEP_NAME='03-transform'

        - step toarango
            triggers: external-event
            implementation:
                docker-implementation image: '/ebw-prototype-04-toarango'
            parameters: tranformationJson = '/transformation/transformation.json'
            environment:
                STEP_NAME='04-toarango' }
```

Listing 1: The DSL description of the prototype workflow.

6.1. Inter-step communication

Asynchronous FIFO message queues are used to implement inter-step communication and the KubeMQ[15] messaging system was chosen for this purpose. KubeMQ provides multiple message queues with a guarantee of exactly-once message delivery. A communication link is established when a workflow step is configured with a message queue as its output channel, and the same queue is used as an input channel for another step. In this way, the latter step waits for the output of the former and its execution is triggered immediately when the shared queues have content. This communication mechanism enables the two step instances to run concurrently during execution while maintaining race-condition-free data access. Such workflow execution follows the P&F architecture, i.e., step instances as filters and message queues as pipes. In the example prototype, a containerized instance of KubeMQ is used to handle the communication between the steps and is made accessible to all workflow steps. Each consecutive workflow step is configured to share a message queue (except the first and last steps). Thereby, steps in the workflow use two message queues: one for retrieving information about available input data from the previous step and one for signaling that data have been made available for the next step. There is no direct link between two consecutive steps; instead, the message queue they share creates a logical connection.

[14] https://www.eclipse.org/Xtext/.
[15] https://kubemq.io.

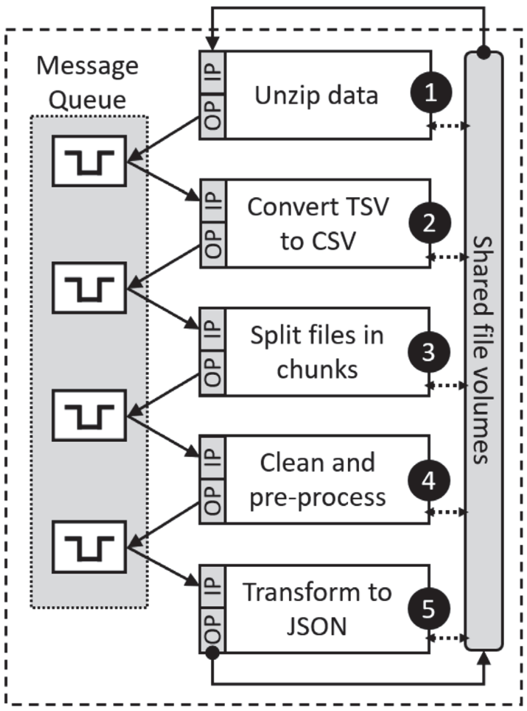

Fig. 4. Prototype Big Data workflow.

Even though the message queues in the prototype workflow serve as a communication mechanism, the actual data for processing are not stored in the message queues but are stored on shared volumes. Only references to the data are placed in the message queues. To organize the processing of the data, a step container allocates at least four different file storage volumes: *in*, *work*, *out*, and *sandbox*. When a file reference becomes available in the input message queue of a step container, it reads the file reference from the queue and accesses the file from its *in* volume. Then, the file is moved to the *work* volume for processing. When file processing is completed, the result is stored in the *out* volume, and the reference of the output file is published in the output message queue so that the next step can use it for further processing. The *sandbox* volume is used to store any files that are not processed successfully.

6.2. Container orchestration

The individual steps in a workflow are wrapped as Docker containers. The Docker images of the workflow steps are derived from a generic step template that implements the input processing, output processing, and communication logic. The template is modified by adding the installation script for any necessary software libraries in the Docker image building configuration and injecting relevant code scripts (step processing code) in a specific place in the template logic. Container orchestration systems provide effective means to deploy distributed applications across a heterogeneous cluster of resources. To use these tools, it is necessary to prepare a deployment configuration file either in JSON or YAML, which describes the location of the container images, network setups, and other configurations. In the case of Big Data workflows, the configuration needs to include the description of different workflow steps and the communication medium. Scalability and other constraints can also be stated in the configuration. When deploying a workflow, the orchestration tool will automatically schedule the deployment of each workflow step to a cluster and pick the right host, taking into account any stated requirements or constraints.

The prototype workflow deployment was done by composing individual workflow step Docker containers using a Docker-compose file. A workflow step description in the file includes shared volumes, environment values, and message queue assignments. The communication medium, KubeMQ, is also included as a separate service in addition to the workflow steps. Rancher[16] is used as an orchestration tool to deploy the workflow on a private cloud environment running Docker containers. In Rancher, the number of instances for each step can be defined in a separate YAML file. When this YAML file, together with the Docker compose file, is supplied, Rancher handles the deployment by taking the scalability requirement into account and finding the right host for each container.

[16] https://rancher.com.

Table 2
Summary of feature-based comparison.

Workflow tool	Step-level containerization	Communication mechanism	Parallelism	DSL for workflow	Separation of concerns
Airflow	Yes	Yes	No	Yes	Yes
Argo	Yes	No	Yes	No	Yes
Conductor	No	Yes	Yes	Yes	No
Nextflow	Yes	Yes	Yes	Yes	No
NiFi	No	Yes	No	Yes	No
Node-RED	No	No	No	No	Yes
Pachyderm	Yes	No	No	No	Yes
SnakeMake	Yes	No	Yes	No	Yes
Our approach	*Yes*	*Yes*	*Yes*	*Yes*	*Yes*

6.3. Fault tolerance

To address faults in Big Data workflows, it is necessary to keep track of each file's status while it is being processed. Having this information stored in a structured way helps for monitoring and debugging purposes. To this end, a centralized logging functionality writing structured logs in JSON format is incorporated. These logs are stored in scalable message queues, such as Kafka [29], and contain important execution statuses, warnings, and errors. A centralized logging system can be a bottleneck when the system grows. This is because the logging unit gets overwhelmed by the flow of log data coming from scaled up workflow step instances. This slows down the overall performance of the workflow. However, our implementation is not significantly affected by this problem as the centralized logging approach is inspired by the state management system in well-known scalable platforms, such as Flink [30] and MillWheel [31]. In addition, the bottleneck on the logging mechanism can also be alleviated by using decentralized queuing systems. Errors in the common and step-specific scripts are handled differently by using a separate exception handler. This is done by checking exit codes of program calls and operating system control flow constructs. Using such constructs, important program calls in the scripts are bound together with an error handler function. The error handler will be invoked when the program exits with an error code and logs error information, including the file causing the error, the line number that caused the error, and the input file's name being processed.

To make the workflow more resilient to intermittent failures, we implemented retrying processing of failed input files. Whenever an error occurs, the input file is moved to the respective *sandbox* volume to be processed later or to be archived. Since the message queue implements exactly-once delivery, the reference of the file will not be available in the queue for retrials. In this case, all the failed files can be accessed directly from the sandbox without the need to access the message queue. When a step becomes free, i.e., when there are no more files in the message queue (or input directory for the first step), it tries to re-process files from the sandbox. With the current implementation, retrial is done only once.

7. Evaluation

In this section, we present the evaluation of the proposed solution. It includes a feature-based comparison of the workflow solution against existing workflow tools and performance experiments based on the prototype implementation.

7.1. Feature-based comparison

To compare the proposed workflow solution with other similar tools, we selected five main implementation features: (i) the ability to wrap workflow steps as containers (isolated units); (ii) communication mechanism between steps and their activation trigger; (iii) parallel execution of workflow steps; (iv) inclusion of DSL for workflow definition; and (v) separation of concerns. (see Table 2).

Granular containerization (i.e., step-level containerization) is not provided out of the box for Big Data workflows containing multiple steps. Hence, step-level scalability cannot be achieved when faster processing is needed for an individual workflow step. Argo Workflows, Nextflow, Pachyderm, and SnakeMake are instances of the few Big Data workflow tools supporting step-level containerization. Nextflow also has an additional feature that allows deploying the entire workflow as a single container. The majority of existing data workflow tools lack such a separate communication link and instead, steps in these tools are tightly coupled. Thereby, the logic that determines the flow of step execution is embedded as part of the step implementation. Even though Argo Workflows provides both sequential and parallel workflow step definitions, this type of communication mechanism is missing.

Apache NiFi and Nextflow use a queuing system for inter-step communication. Nextflow provides advanced data binding and publish/subscribe features. Conductor has a special type of workflow step (i.e., event step) to enable event-based dependencies for steps by publishing events internally or an external message queuing system like Amazon SQS. Apache Airflow uses a feature called XCom to communicate small messages between steps and larger data are exchanged using remote storage such as S3 and HDFS.

Workflow tools such as Apache Airflow, Conductor, Nextflow, and Apache NiFi provide parallel execution of step instances from different workflow steps. In Apache Airflow and Conductor, special operators are used to define a parallel set of workflow steps. Parallelism support in Argo Workflows is limited to all workflows in the system (i.e., it is not granular to a class of workflows, or steps within them).

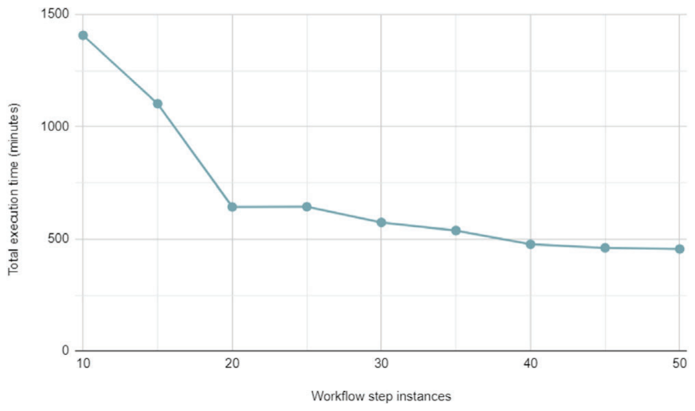

Fig. 5. Results of the horizontal scalability test.

Defining workflows in existing data workflow tools requires knowledge of general-purpose programming and scripting languages such as Java, Python, Scala, or R. Consequently, domain-experts face a significant learning curve to master these languages. The need for DSLs is indispensable in this regard. Only a few Big Data workflow tools, e.g., Argo Workflows, Conductor, Nextflow, and SnakeMake, support a custom-made DSL for workflow definitions.

Separation of concerns is not a focus in tools like Conductor, Nextflow and Apache NiFi. Hence, there is no mechanism for the separation of high-level workflow definition concerns from step-specific implementation and deployment details. Therefore, such tools do not ensure the separation of design- and run-time aspects of workflows.

7.2. Performance evaluation

We evaluated our workflow approach's horizontal scalability using the prototype workflow and compared it with the Argo Workflows. We used eight heterogeneous physical hosts configured to form our distributed testbed. Three of them have 12-core Intel CPUs and 64 GB RAM each; the other five — four-core AMD CPUs and 16 GB RAM each. The hosts are connected in a Gigabit Ethernet network, share a distributed file system, and run the Docker engine connected to Rancher.

7.2.1. Scalability evaluation

Our workflow solution allows us to scale individual workflow steps. Hence, it is worth investigating the impact of horizontal scalability on the performance of the workflow. Therefore, we designed an experiment to determine how the number of workflow step instances (i.e., containers) affects workflow execution performance. The prototype workflow, shown in Listing 1, was used for this experiment. We defined an increasing number of workflow step instances and measured the time it takes to complete processing input files. Input data size is kept constant over all iterations. The results of this experiment are shown in Fig. 5.

We performed the scalability experiment by processing approximately 100 GB of compressed TSV files in nine rounds. These are historical data from the use case described in [28] that comprises a large volume of extracts from the Google Ads platform[17] and made available for this experiment by a digital marketing company. For each round, we increased the number of workflow step instances by five (starting from 10), and the increased number was distributed among the steps based on the previous step execution time. Hence, the step that takes the longest time was allocated a higher number of instances. We stopped adding more step instances after nine rounds since we noticed the minimal effect in the last two rounds. Fig. 5 shows that with four times increase in the number of instances, the total execution time decreases from 1406 to 455 min, i.e., approximately a three-times decrease. There is a significant reduction of execution time up to 20 containers. However, from 25 containers upwards, the reduction in execution time gets smaller. This reduction in performance gains is due to resource bottlenecks with the addition of more instances. In our experimental setup, resources available to the containers are shared. They are split among the containers by availability time, size or processing power and increasing the number of instances beyond 50 does not improve the performance further. Even though such an arrangement helps to utilize the resources effectively, it can cause resource contention among the containers. As the number of instances increases, it leads them to compete for the resources. Further increase of instances can result in degraded performance due to CPU, memory, and I/O bottlenecks. Measuring the usage of such resources (CPU, memory, I/O, Network Receive/Transmit Throughput, etc.) would be interesting but is considered out of scope for this evaluation as the container orchestration system does not allow to easily change or customize them.

[17] https://ads.google.com/home.

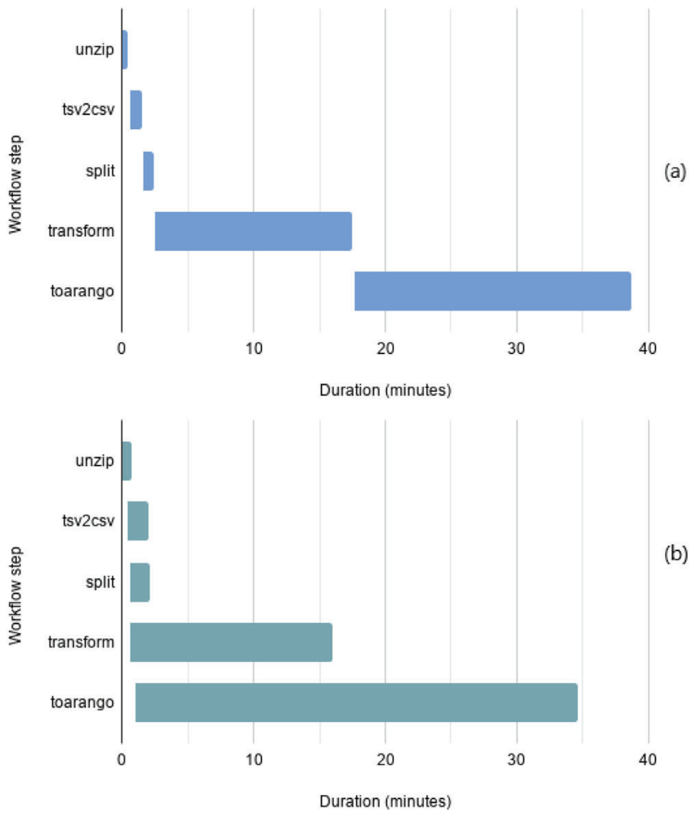

Fig. 6. Workflow execution times (a) using Argo Workflows and (b) using the proposed approach.

7.2.2. *Comparison with Argo Workflows*

The implemented workflow approach allows individual workflow steps to run independently. In this experiment, we investigated the effect of concurrent workflow execution on the workflow's performance by comparing our approach with another workflow tool that does not have built-in concurrency support. For the comparison, we chose Argo Workflows since it does not have built-in concurrency support. The composition of a workflow can be done quickly using its custom DSL that is similar to traditional YAML. Together with the step template invocator, the container template was used to compose the sequential workflow with a single instance assigned for each step. The experiment had two parts: one using fixed input data size, and the other is by using increasing input data size.

Experiment with fixed input data size. In this experiment, we compose workflows in both approaches and measure the execution time of each step with a fixed volume of input data. We measure each the start and end time of each step to observe the performance difference between concurrent and sequential execution modes. To achieve this, we used the workflow in Listing 1 and followed the same workflow composition as in the scalability evaluation from Section 7.2.1, but with a single container instance assigned to each step. A similar workflow structure was composed in Argo Workflows using the same step container images. Both workflows were executed using 100 MB of input data on a Kubernetes cluster (Argo Workflows runs only on Kubernetes). We measured the execution time for each step. To minimize systematic errors, we repeated each experiment 20 times and took the arithmetic mean of execution time to compare the two modes. The repetitions were done in a cool start fashion without caching result or intermediate data from previous rounds.

The sequential execution mode in Argo Workflows takes 38.7 min to complete, whereas the concurrent mode takes 34.6 min, which shows a slight performance gain when using our approach. Fig. 6(a), shows the execution time for each step in Argo Workflows. Since the workflow is in sequential execution mode, step container execution starts after the previous completes processing all the input files. In this mode, step containers are not initialized at the time of workflow submission, and initialization occurs right before step execution, which results in a short gap in between consecutive steps. The concurrent workflow execution based on our workflow implementation is shown in Fig. 6(b). We observe a slight performance gain due to the concurrent execution of steps. Concurrency is enabled by the P&F architectural design which allows workflow step container to continue processing the next input file right after passing the processed file to the next step. Moreover, concurrent step containers share the host machine's resources, which explains the longer execution time of the last step (compared to sequential mode). Additionally, we do not observe positive effects of the concurrency in the first three steps since their execution times are relatively short (they are completed in a couple of minutes).

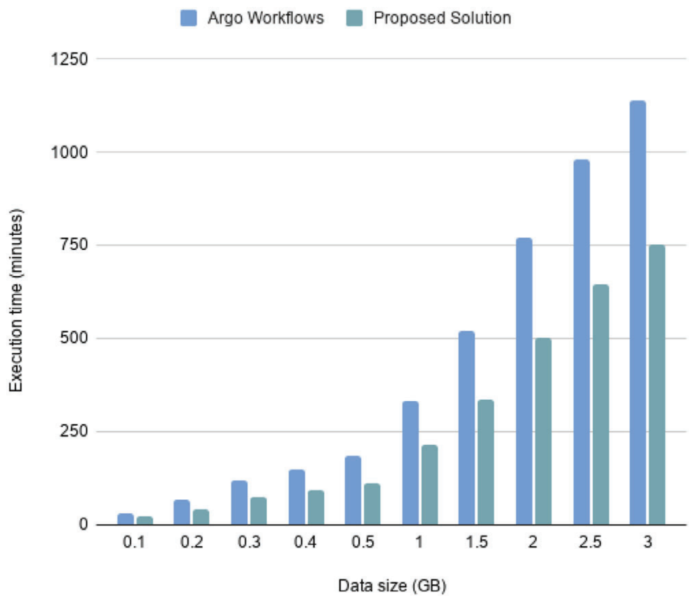

Fig. 7. Performance with increasing input size.

Experiment with increasing input data size. The previous experiment is repeated to evaluate which of the two workflow approaches performs better with increasing input data of compressed CSV files. To achieve this, we used the workflow from Listing 1 once again and measured execution time for both approaches using input data starting from 0.1 up to 3 GB. Fig. 7 shows the results of the experiment. It is evident from these results that our workflow approach performs better than Argo Workflows in all cases, with an approximate performance gain of 36%. However, we did not observe a significant performance gain when input data are further increased. This is because the last two steps are computationally intensive and they determine the overall speed of the workflow. Having a single container instance for each step, the addition of more input data does not bring any significant performance gain as the execution rate of steps is (relatively) constant independent of the size of the input data.

7.3. DSL evaluation

We evaluated the proposed DSL qualitatively in what follows from a development and quality perspectives from multiple dimensions.

7.3.1. Development perspective

One of the benefits of using DSL over general-purpose programming language is to reduce code complexity and increase development efficiency. However, it is challenging to measure the level of code abstraction introduced by using DSL. A simple quantitative evaluation metric can be used by measuring and comparing the amount of code required to define a workflow with and without DSL usage [32]. The measurement can be expressed in lines of code (LOC). Another metric is to compare the number of concepts or technologies involved with and without the DSL. This metric shows the number of concepts abstracted by the DSL.

For example, to implement the example workflow presented without using the DSL, it is necessary to follow some specific steps; however, the important work in this process is to compose all the workflow steps into a single deployment file. Then, assuming we have all the step images ready, it is required to set up the MOM and shared volumes. Also, input/output parameters, message queues, and environment variables must be adequately assigned for each workflow step. By using the DSL, composing a workflow requires fewer configurations since the DSL abstracts some concepts. For example, it is not required to set up MOM and assign message queues for workflow steps. Intermediate storage volumes and settings are not specified either. In other words, workflow definition using the DSL does not necessarily require knowing technologies like shared volume management, MOM, and message queues. To provide a quantitative comparison, we examined the LOC required to define workflow shown in Fig. 4 with and without the DSL. Without using the DSL, the Docker compose file has 133 LOC.[18] Whereas the same workflow can only be expressed using 36 LOC by the DSL on a higher level of abstraction — see Listing 1. The DSL provides the capability to define the example workflow in a short LOC because of the abstraction of key concepts and technologies needed to define a workflow.

To test the functional ability of our DSL in the context of an IoT scenario, we mapped a third-party data preparation workflow used at Bosch as described in [33]. The mapping of the workflow to our approach is shown in Fig. 8. The IoT scenario represents

[18] https://github.com/SINTEF-9012/ebw-prototype/blob/master/docker-compose.yml.

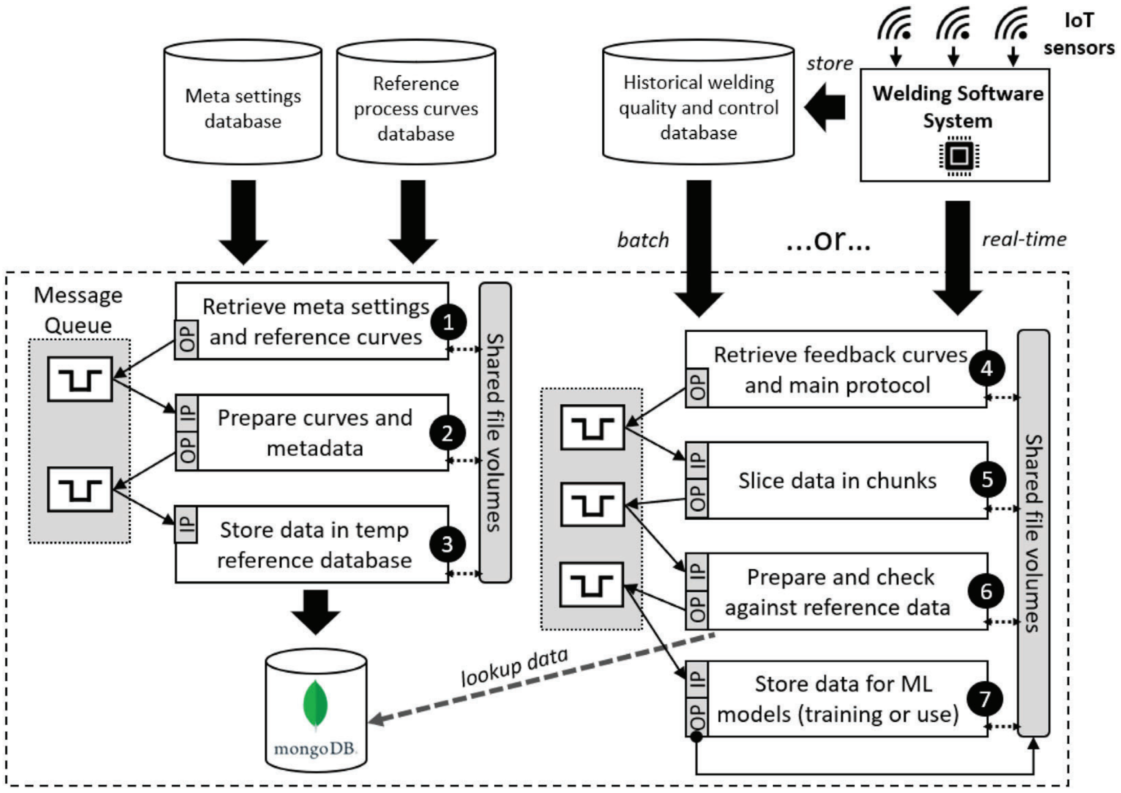
Meta settings database
Reference process curves database
Historical welding quality and control database
store
IoT sensors
Welding Software System
batch
...or...
real-time
Message Queue
Retrieve meta settings and reference curves
Prepare curves and metadata
Store data in temp reference database
Shared file volumes
Retrieve feedback curves and main protocol
Slice data in chunks
Prepare and check against reference data
Store data for ML models (training or use)
lookup data
mongoDB
OP
IP

Fig. 8. Mapping of approach to an IoT Big Data workflow about welding data preparation for machine learning.

a use case at Bosch[19] of quality monitoring for resistance spot welding. Data about the conducting of process are continuously generated at high velocity by sensors attached to the welding equipment and are managed by a welding software system. These data are stored in a database for and used by the software to signal for the current process status and potential issues. During the welding process, signals need to be generated at near-real time as any necessary response by the system or operators at the factory needs to be quickly addressed to avoid delays on the production lines. In addition to the sensor data from the welding machines, the use case involves reference data that provide information about target parameters of the equipment as well as the settings of the individual machines and welding programs. The overall goal of the work is to create a workflow for offline *batch* preparation of data for machine learning as well as online scenario for *real-time* monitoring using trained ML models.

The workflow consists of two parallel dependent sub-workflows that are involved in processing the different parts of the data. We use a gray shade to represent components specific to our approach that are necessary for the encapsulation of the workflow steps. Data transmission in this workflow and the rest of the examples in this section is done through shared file volumes of a (possibly distributed) file system, whereas the coordination of the steps (i.e., informing the next step of available data for processing) is performed using a Message Queue that is dedicated for each pair of steps. The output of each step is passed by the *Output Processing (OP)* script to the *Input Processing (IP)* script of the next step, both of which are part of the wrapper of the respective images that wrap the step implementations. The left side of Fig. 8 represents the sub-workflow for processing the reference datasets. In the first step, data are retrieved from the meta settings and reference process curves databases and sent for further processing. The curves and metadata are integrated according to a common data model and are also re-formatted and prepared to be referenced by the larger scale sub-workflow. After the preparation data are stored in a reference database (implemented using MongoDB[20]) where they can be accessed for lookups. The reference datasets are continuously but infrequently updated as new machine settings or reference data become available. The large-scale IoT data sub-workflow is displayed on the right side of Fig. 8. This sub-workflow processes either very large volumes of historical welding quality and control data from a database (in the offline setting), or direct signals from the welding software system in real-time (in the online setting). The data retrieval step (numbered as step 4) fetches the data from the respective source and sends it to a dedicated slicing step. In the offline scenario, this step is used to chunk the data in a specific way, whereby the data coming from the software system is packaged together with the exact sensor data that relates to it in chunks of a pre-determined size (this correspondence is needed during the preparation step). The chunk size is calculated so that when the workflow is scaled, there are enough hardware resources (esp. RAM and CPU) to process them are available. The chunks

[19] https://www.bosch.com.
[20] https://www.mongodb.com.

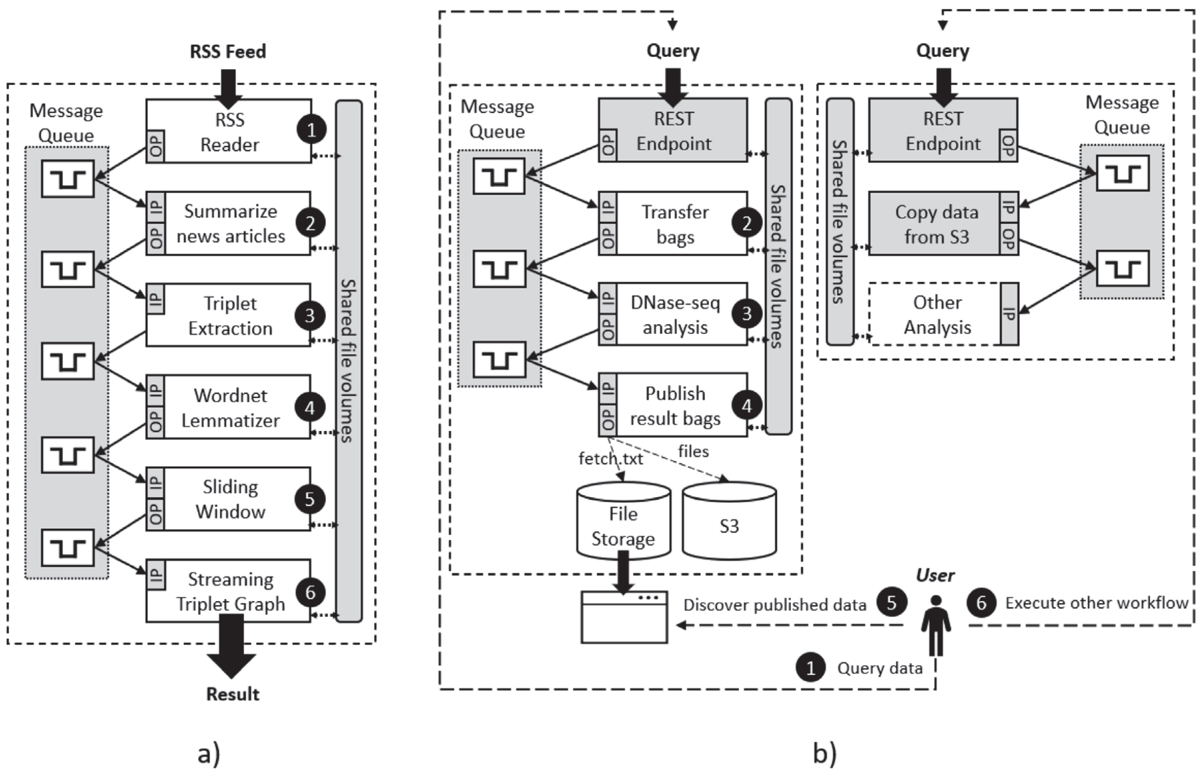

Fig. 9. Mapping of approach to existing Big Data workflows — (a) CloudFlows data mining and (b) ENCODE Data access workflows.

are then sent for preparation during which the data are integrated, reformatted and packaged together with the specific relevant reference data through a lookup from a reference database. The result is then stored on disk for further training of or use by the machine learning models for welding quality prediction.

Another case study we tried to express in terms of our DSL is of the workflow *CloudFlows, A data mining workflow platform* [34]. Fig. 9a shows a semantic triplet graph workflow built on the ClowdFlows framework from an RSS feed workflow.

The first step of the workflow implements an RSS Reader that takes an RSS feed URL as input. The second step summarizes news articles based on the generated read data from the first step. The text from the news feed is then passed to the next step of the workflow that performs triplet extraction. This step implements NLP techniques to extract *subject–verb–object* triples from the news articles. Step four of the workflow uses a WordNet[21] Lemmatizer on the resulting triples, and step five performs a sliding window that takes the number of triplets as input. Finally, the data window is passed to the Streaming triplet graph, which is the last step of this workflow.

Another case study is of *ENCODE Data Access* [35] shown in Fig. 9b. In the first step of the workflow, the user enters a REST-format ENCODE query or uploads an ENCODE metadata file that represents a dataset array corresponding to the so-called *bags* of data. The selected data are then transferred for analysis to the DNase-seq sub-workflow that implements a set of analyses on the data. After the analysis is performed, the result (also serialized into bags) is published on Amazon S3, and a metadata file (named "fetch.txt") is created and stored for serving to the users of the result. A user may then discover these new bags and perform further analysis using the same type of querying as in step 1, but with another analysis set.

7.3.2. Quality characteristics

We used the Framework for Qualitative Assessment of DSLs (FQAD) [36] that defines sets of quality characteristics to evaluate DSLs. The quality characteristics are mainly determined from ISO/IEC 25010:2011 standard but tailored for DSLs. FQAD is helpful for evaluating the requisite quality characteristics at the outset of DSL development, as well as assessing the end product of the DSL development process (i.e., the language). It defines a set of assessment goals and DSL characteristics to fulfill them, which are derived from the ISO/IEC 25010:2011 international systems and software standards model.

Several works in literature have used this framework to evaluate their DSL. For example, Florian et al. [37] used this framework to evaluate a DSL in the business domain for creating test case specification. They did not evaluate the test language itself but only the output and, therefore, concentrated only on the expressiveness, usability and productivity characteristics using the sub-characteristics of completeness, correctness, comprehensibility, reading flow, and reproducibility. Aleksandar et al. [38] employed

[21] https://wordnet.princeton.edu.

the full FQAD framework to perform a quality assessment of their DSL for modeling application-specific functionalities of business applications. Sadiq and Geylani developed DSML4DT - a domain-specific modeling language for device tree software [39]. For qualitative assessment of the language, they prepared a comprehensive questionnaire for the users. To prepare the scoring part of the questionnaire, they employed FQAD and customized with respect to DSML4DT specifications.

Below, we outline the evaluated FQAD characteristics for our DSL and the respective assessments.

- *Functional suitability*: This refers to the degree to which a DSL is completely developed. Functional suitability further has two sub-characteristics:
  - Completeness: The ability of a DSL to express all concepts and scenarios of the domain is termed as completeness. We successfully implemented four Big Data and IoT workflows from different domains with our DSL. The language was expressive enough to describe the workflows efficiently.
  - Appropriateness: The scale to which the DSL is appropriate for the particular applications of the domain is called appropriateness. While implementing the Encode workflow, we faced the challenge of incorporating all elements in one workflow. However, by breaking it down into two separate workflows based on the functionality and with additional REST services, we achieved the required functionality with our DSL.

  The functional suitability analysis shows that all the important functionality is included in the DSL. In other words, the DSL should not contain functionalities that are not part of the domain. In this regard, based on the workflows that were mapped to our DSL, we claim that the DSL implemented in this work covers the core functionality required to define a given data workflow.
- *Usability*: This aspect refers to the degree to which a group of users can use a DSL to achieve specific goals. A DSL must be as simple as possible to express the domain concepts and support its users. As presented, our DSL provides the option to define concise workflows with minimal technical knowledge.
- *Reliability*: It defines the characteristics of the language that help to produce reliable code. This includes functionalities to prevent errors and support for model checking. The DSL was implemented using the Eclipse environment that supports languages designed with precise semantics based on well-defined principles. Additionally, the Eclipse IDE also has essential features to debug and handle code errors. The reliability characteristics has two sub-characteristics:
  - Model Checking: This concerns whether the DSL reduces user error rates. In comparison to writing YAML files, our DSL-based approach has lower error rate. This is, on the one hand due to the conceptual schema that is defined using the Ecore metamodel[22] and Xtext grammar on top of it. When a user is defining a model of our DSL, the Xtext editor uses the Ecore definitions to check for erroneous values or concept instances and highlights incorrectly defined values.
  - Correctness: The term correctness concerns whether appropriate elements, as well as the correct relation between them, are provided, i.e., any unexpected interactions are prevented. Ecore alongside the Xtext framework allow for checking of the validity of each DSL model class instance and attribute value types thus ensuring the correct elements and relations are chosen by the users.
- *Maintainability*: This characteristic refers to the degree to which it is easy to maintain a DSL. A DSL needs to be easy to modify or introduce new concepts. Modularity falls under this characteristic as well. The DSL was designed based on the separation of concerns principle. This makes it easily understandable and maintainable.
  - Modifiability: This characteristic refers to the DSL's ability to incorporate new functionality by as little modification as possible. The modular approach of our DSL makes it easier to add new sub-vocabularies (which is also part of our future work) that can be used to describe Big data workflows with more details.
  - Low coupling: This means how discrete the elements of the DSL are. At this stage, each element of the DSL plays an integral part in describing a workflow. Therefore, the change in each element will have an impact on other elements.
- *Productivity*: This is mainly related to the number of resources required by a user to achieve specific goals. Productivity can be improved using our DSL because of two main reasons. First, the DSL incorporates high-level concepts of Big Data workflows. Hence, designing a workflow using the DSL is simplified and can be done quickly. Second, the language provides automatic generation of template workflow code and configurations. This saves a lot of time compared to when the process is done manually from the ground up.
- *Extensibility*: This characteristic expresses the degree to which a language has mechanisms for users to add new features. The language we have presented currently has a low degree of extensibility since it does not provide users a means (e.g., separate packages in the Ecore model) to extend the language with new functionalities.
- *Compatibility*: This characteristic measures degrees to which a DSL is compatible with the domain and development process. Our DSL was developed iteratively and modified to cover concepts relevant in the domain.
- *Expressiveness*: Expressiveness is defined as the degree to which a problem-solving strategy can naturally be mapped into a DSL program. The DSL was developed after a thorough domain analysis in which each concept has mapped to its corresponding metamodels to represent only core elements and does not cover complex structures such as cyclic workflows. The sub-characteristics of the expressiveness are as follows:

[22] http://www.eclipse.org/modeling/emft/search/concepts/subtopic.html.

  - Mind to program mapping: Concepts are designed appropriately and named so that they are intuitive in order to accommodate problem-solving tasks for the domain.
  - Uniqueness: Because of its simplicity, our DSL provides one and only one good way to express every concept of interest.
  - Orthogonality: Each construct in our Big Data workflow DSL is used to represent exactly one distinct concept in the domain.
  - Correspondence to important domain concepts: DSL constructs correspond to important domain concepts and the language does not include trivial domain concepts. Our DSL only includes highly relevant and non-trivial Big Data workflow concepts.
  - Conflicting elements: This refers to the absence of conflicts between the DSL elements. Our DSL concepts have no conflicts as each element serves a unique purpose for data workflows specification.
  - Right abstraction level: This refers to whether the DSL is at the correct abstraction level that it is not more complex or detailed than necessary. Our DSL provides only the most important elements of the domain at the moment. A few more elements are planned to be added to enhance the functionality — for example, concepts for aspects of data transmission and resource requirements.

- *Reusability*: This characteristic refers to how a language construct can be used in more than one language. The grammar specification of our DSL can be imported and reused in the Eclipse environment. Furthermore, the language is conceptualized so that some elements can be reused in the workflow definition. For example, a workflow step from one workflow can be copied and used as part of another workflow.
- *Integrability*: This characteristic measures how a DSL can be integrated with other languages and modeling tools. The DSL that we have provided can be exported as a plug-in within the Eclipse environment. Integrability of a DSL is largely dependent on the technical space in which it was defined. As shown in [40], the modelware technical space, which is the one used for defining our DSL, provides a high level of integrability both within the technical space and across technical spaces. Furthermore, the Eclipse modeling environment provides integrations with other languages and frameworks such as Javascript,[23] which increases the integrability of our DSL. Finally, as demonstrated in [40], the EMF framework provides automated migration of models through the Edapt framework.[24] Therefore, we conclude that the DSL has a high degree of integrability.

8. Conclusions

In this article, we described a Big Data workflow approach that allows specification of Big Data workflows at a high level of abstraction and enables separation of design- and run-time aspects while maintaining scalable workflow execution. Scalable workflow execution entails parallel data processing that requires workflow fragments to run separately on different computing resources. In the future, we plan to implement comprehensive provenance support. Another limitation to be addressed is the use of a centralized message-oriented communication, which could become a bottleneck for large-scale execution and is a single point failure. Decentralized communication media, Web services, or distributed file systems can potentially be used to address this issue. The DSL that is used in this work is limited to expressing core workflow structures. Thus, the experiments on scalable workflow execution can be performed using an extended DSL, e.g., including infrastructure requirements or with different workflow (sub-) structures. Finally, a visual language for workflow definition, composition, and run-time monitoring would be essential to support domain-experts' participation in workflow design.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was partly funded by the EC H2020 projects *"DataCloud: Enabling The Big Data Pipeline Lifecycle on the Computing Continuum"* (Grant nr. 101016835) and *"COGNITWIN: Cognitive plants through proactive self-learning hybrid digital twins"* (Grant nr. 870130), and the NFR project *"BigDataMine"* (Grant nr. 309691).

References

[1] R. Qasha, et al., Dynamic deployment of scientific workflows in the cloud using container virtualization, in: Proc. of the CloudCom 2016, 2016, pp. 269–276.
[2] M. Barika, et al., Orchestrating big data analysis workflows in the cloud: Research challenges, survey, and future directions, ACM Comput. Surv. 52 (5) (2019).
[3] R. Ranjan, et al., Orchestrating big data analysis workflows, IEEE Cloud Comput. 4 (3) (2017) 20–28.
[4] R. Buyya, et al., A manifesto for future generation cloud computing: Research directions for the next decade, ACM Comput. Surv. 51 (5) (2018) 1–38.

[23] https://wiki.eclipse.org/JS4EMF.
[24] https://www.eclipse.org/edapt/.

[5] Y.D. Dessalk, et al., Scalable execution of big data workflows using software containers, in: Proc. of the MEDES 2020, 2020, pp. 76–83.
[6] A. Kashlev, et al., Big data workflows: A reference architecture and the DATAVIEW system, Serv. Trans. Big Data 4 (1) (2017).
[7] W. Gerlach, et al., Skyport - Container-based execution environment management for multi-cloud scientific workflows, in: Proc. of the DataCloud 2014, 2014, pp. 25–32.
[8] N. Russell, et al., Workflow data patterns: Identification, representation and tool support, in: Proc. of the ER 2005, 2005, pp. 353–368.
[9] C. Wulf, et al., Increasing the throughput of pipe-and-filter architectures by integrating the task farm parallelization pattern, in: Proc. of the CBSE 2016, 2016, pp. 13–22.
[10] E. Curry, Message-oriented middleware, in: Middleware for Communications, John Wiley & Sons, Ltd, 2005, pp. 1–28.
[11] N. Naik, Docker container-based big data processing system in multiple clouds for everyone, in: Proc. of the ISSE 2017, 2017, pp. 1–7.
[12] S. Junsawang, Y. Limpiyakorn, A domain specific language for scripting ETL process, in: Proc. of the WCSE 2017, 2017, pp. 239–243.
[13] M. Mernik, et al., When and how to develop domain-specific languages, ACM Comput. Surv. 37 (4) (2005) 316–344.
[14] M. Adhikari, et al., A survey on scheduling strategies for workflows in cloud environment and emerging trends, ACM Comput. Surv. 52 (4) (2019) 1–36.
[15] D.C. Arvind, et al., The Tagged Token Dataflow Architecture, Technical Report, MIT Laboratory for Computer Science, 1984.
[16] A. Alaasam, et al., Scientific micro-workflows: Where event-driven approach meets workflows to support digital twins, in: Proc. of the RuSCDays 2018, 2018, pp. 489–495.
[17] R. Filgueira, et al., Asterism: Pegasus and dispel4py hybrid workflows for data-intensive science, in: Proc. of the DataCloud 2016, 2016, pp. 1–8.
[18] R. Filguiera, et al., dispel4py: A Python framework for data-intensive scientific computing, Int. J. High Perform. Comput. Appl. 31 (4) (2017) 316–334.
[19] L. Gerhardt, et al., Shifter: Containers for HPC, J. Phys. Conf. Ser. 898 (8) (2017) 082021.
[20] M. Belkin, et al., Container solutions for HPC systems: a case study of using Shifter on Blue Waters, in: Proc. of the PEARC 2018, 2018, pp. 1–8.
[21] L. Bryant, et al., VC3: A virtual cluster service for community computation, in: Proc. of the PEARC 2018, 2018, pp. 1–8.
[22] D. Palma, T. Spatzier, Topology and Orchestration Specification for Cloud Applications Version 1.0, OASIS Standard, 2013.
[23] P. Kacsuk, et al., The flowbster cloud-oriented workflow system to process large scientific data sets, J. Grid Comput. 16 (1) (2018) 55–83.
[24] Y.D. Dessalk, Big Data Workflows: DSL-based Specification and Software Containers for Scalable Execution, The Royal Institute of Technology, 2020.
[25] T. Fernando, et al., WorkflowDSL: scalable workflow execution with provenance for data analysis applications, in: Proc. of the COMPSAC 2018, 2018, pp. 774–779.
[26] C. Zheng, D. Thain, Integrating containers into workflows: A case study using makeflow, work queue, and docker, in: Proc. of the VTDC 2015, 2015, pp. 31–38.
[27] S. Migliorini1, et al., Pattern-Based Evaluation of Scientific Workflow Management Systems, Tech. Rep., Queensland University of Technology, 2011.
[28] V. Cutrona, et al., Semantically-Enabled Optimization of Digital Marketing Campaigns, in: Proc. of the ISWC 2019, 2019, pp. 345–362.
[29] J. Kreps, et al., Kafka: A distributed messaging system for log processing, in: Proc. of the NetDB 2011, 2011, pp. 1–7.
[30] P. Carbone, et al., Apache flink: Stream and batch processing in a single engine, Bull. IEEE Comput. Soc. Tech. Committee Data Eng. 36 (4) (2015).
[31] T. Akidau, et al., Millwheel: Fault-tolerant stream processing at internet scale, Proc. VLDB Endowment 6 (11) (2013) 1033–1044.
[32] T. Wegeler, et al., Evaluating the benefits of using domain-specific modeling languages: An experience report, in: Proc. of the DSM 2013, 2013, pp. 7–12.
[33] B. Zhou, et al., SemFE: Facilitating ML pipeline development with semantics, in: Proc. of the CIKM 2020, 2020, pp. 3489–3492.
[34] J. Kranjc, et al., Clowdflows: Online workflows for distributed big data mining, Future Gener. Comput. Syst. 68 (2017) 38–58.
[35] K. Chard, et al., I'll take that to go: Big data bags and minimal identifiers for exchange of large, complex datasets, in: Proc. of the BigData 2016, 2016, pp. 319–328.
[36] G. Kahraman, S. Bilgen, A framework for qualitative assessment of domain-specific languages, Softw. Syst. Model. 14 (4) (2015) 1505–1526.
[37] F. Häser, et al., Is business domain language support beneficial for creating test case specifications: A controlled experiment, Inf. Softw. Technol. 79 (2016) 52–62.
[38] A. Popovic, et al., A DSL for modeling application-specific functionalities of business applications, Comput. Lang. Syst. Struct. 43 (2015) 69–95.
[39] S. Arslan, G. Kardas, DSML4DT: A domain-specific modeling language for device tree software, Comput. Ind. 115 (2020) 103179.
[40] N. Nikolov, et al., Integration of DSLs and migration of models: a case study in the cloud computing domain, Procedia Comput. Sci. 68 (2015) 53–66.