Secure and Polymorphic Composition of Service Workflows in the Cloud Continuum

Ruslan Bondaruc[1,2], Alberto Ovena[3], Mattia Perfumo[3], Mirko Gala[4], Ernesto Damiani[5], and Marco Anisetti[1,2]

[1] Dipartimento di Informatica, Università degli Studi di Milano, Milan, Italy
{first.last}@unimi.it
[2] Moon Cloud srl, Crema, Italy
{first.last}@moon-cloud.eu
[3] Dipartimento di Informatica, Università degli Studi di Milano, Milan, Italy
{first.last}@studenti.unimi.it
[4] DGcom srl, Brescia, Italy
mgala@dgcom.eu
[5] Center for Cyber-Physical Systems, Khalifa University, Abu Dhabi, UAE
ernesto.damiani@ku.ac.ae

Abstract. Modern applications are increasingly built as compositions of services deployed across diverse platforms. Ensuring the security and privacy of these service compositions remains a significant challenge, particularly within the dynamic and distributed environment of the Cloud Continuum. In such contexts, choreography, which avoids reliance on a centralized orchestrator, is often more suitable than traditional orchestration approaches. In this paper, we introduce a polymorphic workflow composition methodology that supports the design of choreographed workflows capable of adapting their structure (i.e., the polymorphism) and behavior based on specific security requirements and deployment contexts. Our approach generates deployment recipes that preserve critical non-functional properties, such as security and privacy. We evaluate our methodology in both traditional Cloud Continuum environments and high-performance computing (HPC) clusters, demonstrating its ability to maintain deployment independence from the underlying infrastructure.

Keywords: Workflow language · Secure deployment · Service choreography.

1 Introduction

Today, supported by the miniaturization of services and lambda functions, applications are often offered as compositions. Composing an application involves selecting from a pool or registry the services implementing the functional logic and arranging them in a graph pattern, thereby shaping the application as a workflow [1]. This paradigm brings several advantages, primarily the decomposition of the application structure into modular components based on functional (e.g., implemented operations) and non-functional (e.g., hardware or software

platform requirements) criteria. Composition services are treated as independent black-boxes, capable of being developed using various languages and frameworks, deployed across various computation models such as IaaS, PaaS and SaaS, and executed on heterogeneous physical nodes, provided that suitable interaction mechanisms are established [2]. Containerization technologies further boost service independence by encapsulating software dependencies within containers. This enables services to operate seamlessly across diverse environments, including the Cloud-Edge Continuum, an emergent paradigm that integrates existing computation paradigms (i.e., Cloud, Edge, and On-Premises) into a unified distributed computing and network infrastructure [3].

Although the Cloud-Edge Continuum offers the ability to host and dynamically migrate services with specialized requirements, existing approaches do not yet fully leverage this capability. In particular, there is a lack of standardized mechanisms to declare non-functional requirements associated with workflow services. Graph-based models are commonly employed to represent workflows, but they typically omit non-functional aspects and are designed for theoretical representation rather than practical user interaction. Moreover, no standardized approach exists for implementing security properties. Typically, enforcement mechanisms are controlled by the infrastructure provider, who determines the implementation strategy, the technologies employed, and the strength of security measures. Some properties, such as confidentiality, may be inherently guaranteed if a service is deployed on a node with an encrypted file system. However, other security properties require additional mechanisms, such as deploying support services to ensure compliance. For instance, data integrity may necessitate a separate service to sign the data before storage. As a result, each node in the Continuum can independently define its own set of properties as a list of capabilities. Nevertheless, these capabilities vary from node to node and may change over time, both in terms of their availability and implementation. Consequently, there is a clear and pressing need for a solution that enables: *i)* the description of workflows in terms of both functional and non-functional requirements, and *ii)* the deployment and execution of workflows as dynamic compositions of services in the Continuum.

Numerous solutions have been proposed to address these challenges, albeit independently and partially. Many, such as BPEL, BPMN, AWS Step Functions, and Workflow Description Language (WDL), focus on orchestration as a management and communication approach. However, orchestration presents several limitations within the Continuum: *i)* lifecycle monitoring of the entire application is resource-intensive, especially for long-running applications; *ii)* service migration increases management complexity; and *iii)* orchestration introduces a single point of failure. Choreographic approaches, such as Chor and WS-CDL, mitigate these issues by describing service interactions in a decentralized, peer-to-peer manner [4]. Existing solutions for both orchestration and choreography remain largely monomorphic, that is, they rely on static decision processes and lack the flexibility to dynamically adapt to varying deployment contexts. In contrast, deploying service workflows across the Continuum demands the ability to

tailor workflow behavior to both specific requirements and the changing deployment environment. In other words, a polymorphic solution is needed.

In this paper, we propose a polymorphic workflow composition methodology that allows to design choreographed workflows and enables workflows to adjust their behavior and structure based on security requirements and deployment decisions. Our methodology generates deployment recipes preserving given non-functional properties such as security and privacy. We focus on choreographed workflows instead of orchestration-driven ones to cope with the peculiarities of the Continuum, where centralized entities such as orchestrators are more complex to implement and manage.

The contribution of this paper is threefold. First, we extend WDL, a workflow definition language, to support choreography, resulting in the cWDL Language. Second, we introduce annotations in cWDL workflows to specify non-functional requirements (e.g., security, privacy, latency) at level of tasks, connections between tasks, and the workflow as a whole. Third, we develop a Polymorphic Deployment Engine (PDE) that instantiates the given annotated cWDL workflow by selecting suitable tasks and/or applying polymorphism to transform the workflow structure to comply with the annotated requirements, ultimately generating suitable deployment recipes.

The remainder of the paper is organized as follows. Section 2 presents our polymorphic service composition methodology. Section 3 illustrates the architecture implementing our methodology. Section 4 introduces cWDL, our polymorphic modeling language for choreographic workflows. Section 5 discusses the experimental evaluation of the proposed methodology. Section 6 reviews related work. Finally, Section 7 outlines the conclusions.

2 Polymorphic Methodology

In this work, we consider a scenario in which: *i)* users aim to design and deploy data processing workflows composed of various tasks, and *ii)* a Cloud Service Provider (CSP) offers its Continuum (CSP Continuum) for workflow deployment, supporting workflow design through a platform such as the one proposed by MUSA [5]. A workflow is defined as a composition of services organized in a graph structure, where each node represents a service. By definition, a workflow encompasses the concepts of sequentiality (i.e., a service must be executed before others), parallelization (i.e., multiple services can be executed concurrently), and inter-service communication, represented by the edges of the graph.

To compose their workflows, users generate a workflow deployment request by selecting a set of services from a registry, that is, an indexed catalog containing available services, their images, and functional descriptions. Services are selected based on the operations they implement, the manner of the implementation, and output formats. After selecting a service, users may annotate it with non-functional requirements that specify the resources and properties expected from the facility that will host the service. Non-functional requirements include QoS constraints (e.g., latency, execution deadlines, resource demands), as well as

privacy and security-related constraints, such as confidentiality, integrity, and authentication.

Once the workflow request is completed and annotated, it is submitted to the PDE, which computes the optimal placement for each service and orchestrates the deployment across the CSP Continuum. This Continuum consists of a set of computational nodes (hereafter referred to as facilities) arranged in a graph, with each node representing a facility. Due to the potential limitation in reachability across the infrastructure, the edges of the graph denote the available communication links between pairs of facilities. Each facility is annotated with its capabilities, i.e., the resources it hosts and the security properties it guarantees.

Non-functional properties can be provided by a facility in one of two ways: inheritance or overriding. Inheritance applies when deploying a service on a facility inherently satisfies the required property. In other words, no additional mechanism is required to ensure the requested property other than selecting that specific node for the deployment. For instance, confidentiality at rest is inherited when a facility employs an encrypted file system, as any service using that system automatically benefits from the property. In contrast, overriding is required when the desired property cannot be guaranteed solely by the deployment node and necessitates additional operations. This typically involves modifying the workflow with the injection of one or more support services, following defined patterns (discussed in Section 4.3). A support service is thus a service originally not part of the workflow, but is added by the PDE to enforce a property. An example of overriding is the enforcement of integrity, achieved by injecting a service that computes a cryptographic signature of the data before it is processed by the target service.

The ability to override the workflow endows our approach with polymorphism, enabling the workflow structure to adapt to varying deployment contexts. Specifically, our methodology supports: *i)* describing the workflow (Θ) by selecting tasks using our cWDL language; *ii)* annotating tasks, inter-task channels, and the entire workflow with non-functional requirements to yield an enriched description (Θ'); and *iii)* generating a deployable workflow instance ($\overline{\Theta}$) along with its corresponding deployment recipe ($\overline{\Theta}_r$) for execution on the given CSP Continuum.

Example 1 (Reference Scenario). Consider a pharmaceutical company aiming to perform molecular development via molecular-specific simulation workflows. Assume the company has an agreement with a given CSP implementing our PDE. According to these agreements, the CSP Continuum may include nodes located on the pharmaceutical premises (e.g., for data collection) and is utilized for deploying the company's workflows. Given the sensitive nature of pharmaceutical research, the following non-functional properties must be guaranteed: *i)* only authorized users are permitted to execute specific analytics on designated molecules; and *ii)* fine-grained control is enforced over the integrity, security and intellectual property of the developed molecules.

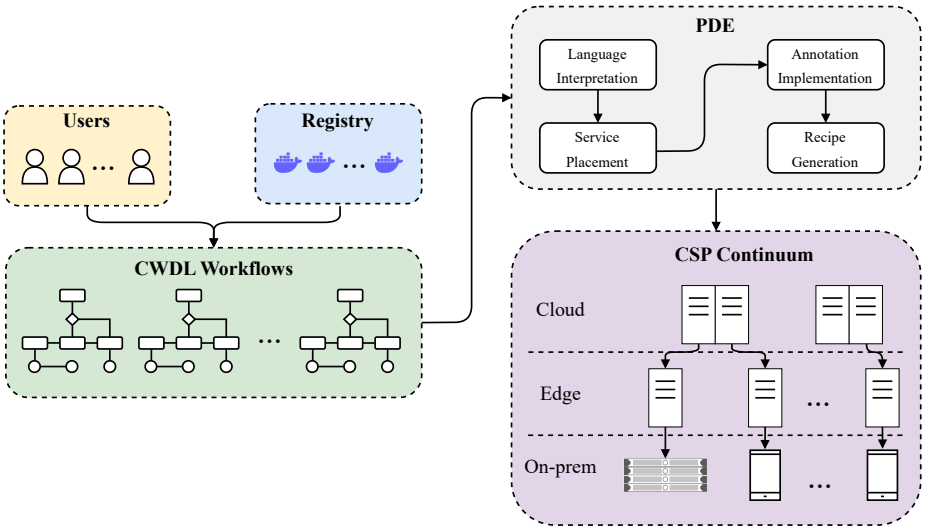

Fig. 1. Deployment architecture including a Polymorphic Deployment Engine (PDE) interpreting annotated workflows and deploying them in the Cloud-Edge Continuum.

3 Architecture

Fig. 1 illustrates the architecture implementing the methodology introduced in Section 2. It is composed of: *i)* a registry that stores the set of available tasks for workflow composition; *ii)* a high-level description of the workflow Θ and its annotated version Θ' to be deployed; and *iii)* the PDE engine, responsible for generating, deploying, and executing the workflow on the CSP Continuum using the underlying CSP deployment technology.

The PDE engine is the core component of the architecture and operates through four main phases. First, in the language interpretation phase, it parses the annotated workflow Θ' expressed in cWDL to verify its syntactic and semantic correctness (e.g., ensuring the workflow is connected and well-formed). Second, it performs service placement, computing an execution plan that satisfies both functional and non-functional requirements. Third, during the annotation implementation phase, it applies polymorphic transformations to the workflow to enforce non-functional requirements that cannot be fulfilled by existing tasks. For example, if authentication is required but not natively supported by any task, the PDE engine injects an appropriate authorization service, thereby producing the workflow instance $\overline{\Theta}$. Finally, in the recipe generation phase, it creates a deployment artifact $\overline{\Theta}_r$ tailored to the target environment (e.g., Docker Compose, Kubernetes, AWS Step Functions, HPC clusters), enabling automated deployment and execution.

Example 2 (Θ and Θ'). Consider the scenario introduced in Example 1, where a pharmaceutical company designs a molecular docking workflow. The pro-

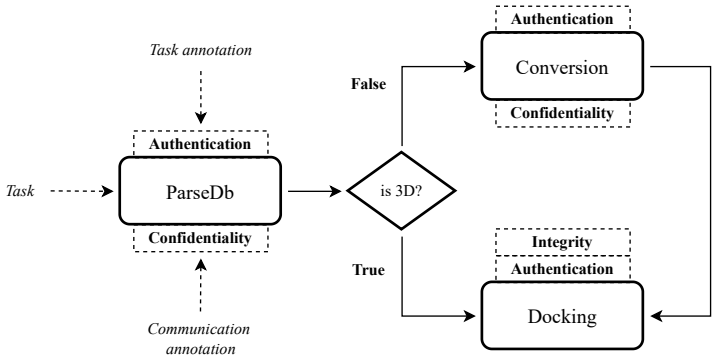

Fig. 2. Θ' workflow for our scenario in Example 2. The workflow is represented in a graphical form, with annotations depicted as dotted rectangle attached to the relevant tasks.

cess begins with the selection of tasks to construct the workflow Θ. First, the *ParseDb* task retrieves data from the companys databases. Next, the *Conversion* task transforms two-dimensional molecular representations into three-dimensional structures. Finally, the *Docking* task executes the molecular docking process. The client then annotates the entire workflow with an authentication requirement, adds channel confidentiality annotations to *ParseDb* and *Conversion*, and applies an integrity annotation to *Docking*, thus producing the annotated workflow Θ', illustrated in Fig. 2.

4 Choreography-based WDL Language

In this section, we introduce cWDL, our proposed language for defining choreographic workflows of services within the polymorphic methodology. cWDL builds upon WDL by adopting a choreographic execution model and incorporating support for security annotations. Unlike orchestration, choreography eliminates the need for a centralized entity managing the data flow; instead, the control is distributed among the services themselves. The grammar of cWDL is derived from the OpenWDL standard, from which it inherits most structures and keywords [6]. The main differences lie in the definition of workflows and the mechanism used to connect tasks. In cWDL, tasks are modeled as services that not only implement the functional logic but also manage communication with subsequent tasks. Thus, workflows are expressed as chains of tasks in which each task explicitly defines the next step upon completion.

A choreographic workflow consists of two sections: definitions and tasks. The definitions section serves as the workflow header and includes metadata necessary for workflow execution, such as version, options, dependencies, data structures, and the starting task. Data structures are specified in a C-like syntax, enabling precise definition of the data exchanged between tasks. The starting task is

declared in this section using the *start* construct. The tasks section describes how each service is built and executed, and supports security annotations.

4.1 Tasks

A task is composed of several sections: input, command, runtime, and next. The input section defines the task inputs and constants, specifying their data types and names, as well as any constant values. The command section lists the shell commands to be executed once the inputs are provided, and supports an escape syntax for referencing variables. The runtime section contains key-value pairs specifying the execution environment, including container runtime and image details. The next section determines the subsequent task to invoke after execution, using the call construct to map outputs to the inputs of the next task. Variables, constants, and conditional logic can also be used via the *if* construct to define conditional workflows. If a task produces the final output of the workflow, it includes a result section specifying the result structure, which is returned only if no subsequent task is invoked.

Example 3 (Task definition). In the Θ workflow from Example 2 expressed in cWDL, the *ParseDb* task takes a File input named Db, and its output is aligned with the requirements of the subsequent task, to which it is forwarded as input. Additionally, the molecules struct (defined in the workflow header) is instantiated as data. This variable is passed to the next task (*Conversion* or *Docking*) via the *call* construct, alongside *Db*. The logic of *ParseDb* is defined in the command section, where data is extracted from Db using the syntax $\sim \{Db\}$. The WDL code for the examples discussed in this paper is provided in Appendix A.

4.2 Annotations

cWDL supports two types of annotations on a workflow Θ: transformation annotations and constraint annotations. Transformation annotations, declared using the *with* keyword, modify the workflow structure (e.g., by adding tasks to enforce polymorphic behaviors). Constraint annotations, specified via the *runtime* or *comms* keywords, define execution parameters (e.g., performance or resource requirements) or communication properties (e.g., secure channels between tasks).

Example 4 (Task annotation). Authentication is enforced globally using the *with auth* annotation on the *start* call. Encryption between tasks is configured through the *comms* section, where confidentiality is set and HTTPS is specified as the secure protocol. Furthermore, the *ParseDb* task includes constraint annotations in its runtime section, detailing the container version, engine, node type, and RAM allocation.

4.3 Workflow Instance Generation

The annotations specified within the workflow are processed by the PDE engine, which handles both constraint annotations and transformation annotations.

Constraint annotations are evaluated in a subsequent stage, immediately prior to recipe generation, to guide deployment decisions. Transformation annotations, instead, are translated by the PDE into predefined patterns, classified as follows: *i)* global patterns, which affect all tasks in the workflow (e.g., the auth pattern in Example 4); *ii)* intra-task patterns, which modify only the annotated tasks; and *iii)* inter-task patterns, which affect interactions between the annotated task and one or more downstream tasks (e.g., the HTTPS pattern in Example 4).

As an example of an intra-task pattern, an availability pattern introduces task replication within the workflow. This approach is fully transparent to other tasks, thereby preserving a black-box model in which intra-task modifications remain encapsulated and isolated from the broader composition.

Example 5 ($\overline{\Theta}$). Consider the annotated workflow Θ' described in Example 2. The PDE engine applies the annotations to generate the instantiated workflow $\overline{\Theta}$. In this case, the global authentication annotation in Θ' is implemented by adding an authentication task at the beginning of the workflow to provide an authentication mechanism (e.g., certificate-based or OAuth-based). Additionally, each task is augmented with authentication features by injecting authentication logic into their container images, enabling them to authenticate against the authentication service. The PDE engine also enforces channel encryption for the *ParseDb* task and appends a signature-verification task after *Docking* to ensure data integrity.

5 Experimental Evaluation

We evaluated the proposed language along two primary dimensions: workflow representation efficiency (measured by the length of the workflow description) and runtime performance (measured by total execution time). This evaluation highlights the benefits of our choreography-based PDE in comparison to existing solutions built on WDL and the Common Workflow Language (CWL) [7]. For this purpose, we selected three representative workflows with distinct complexity and operational characteristics: *i)* a molecular docking workflow following the reference scenario in Example 1, *ii)* a code pipeline simulating a typical DevOps automation process, *iii)* an MLP data pipeline for model training and evaluation. The code pipeline involves three services to download data from a source, process it, and upload it to a destination, respectively. The MLP data pipeline features five services to download the training dataset, perform preprocessing, train the model, evaluate it, and upload the results. These workflows encompass realistic use cases with diverse computational structures, data dependencies, and security requirements, allowing us to assess the flexibility and generality of our approach. Before evaluating runtime performance, we illustrate the process of recipe generation in a Docker-based deployment environment. Additionally, we report on preliminary experiments evaluating the PDE engines ability to generate deployment artifacts for HPC environments, thereby demonstrating the platform-agnostic nature of our methodology. In this context, we consider MLP-

based data-intensive workflows derived from the UAERep project [8, 9] as a representative benchmark for HPC execution.

5.1 Experimental settings

All experiments were conducted on a virtual machine provisioned with 4 vCores and 6 GB of RAM, hosted on a system equipped with an Intel® Core i5-10400 CPU running at 2.90 GHz and 3200 MHz RAM. Each workflow was executed 100 times to ensure statistical robustness. Execution times were measured from workflow submission to completion, encompassing all orchestration and task execution phases. To mitigate the influence of anomalies, outlier values exceeding two standard deviations from the mean were discarded prior to analysis, ensuring that the reported results reflect stable and representative performance trends.

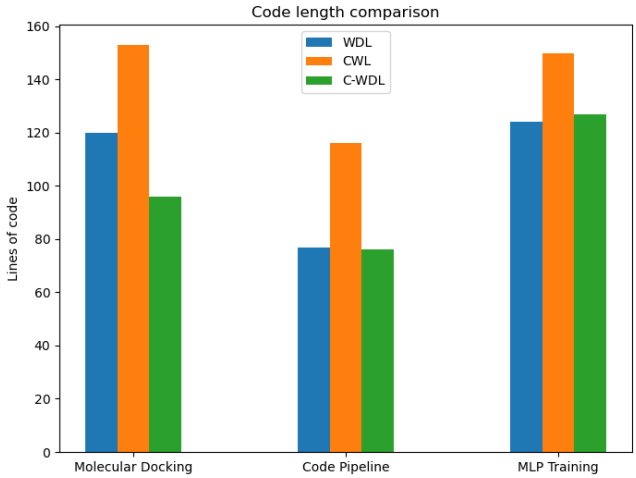

Fig. 3. Code length comparison.

5.2 Representation efficiency

Fig. 3 reports the number of lines of code required to describe the three workflows using WDL, CWL, and cWDL. Overall, code length across the three workflows ranges from approximately 75 to 150 lines. It can be observed that CWL consistently results in the most verbose implementations, reflecting its higher syntactic overhead and lower level of abstraction. In contrast, WDL and cWDL exhibit comparable conciseness, which is expected given that cWDL extends WDL by adapting it to a choreographic and pattern-based paradigm.

In the first experiment, with *Molecular Docking* workflow, CWL required approximately 150 lines of code, making it the most verbose among the three. WDL reduced this to around 120 lines, offering a more compact specification. Notably, cWDL produced the shortest implementation, with approximately 95 lines. This reduction is primarily attributed to the use of predefined patterns (e.g., the authentication pattern), which eliminate the need for developers to manually implement recurring non-functional features such as authentication. By abstracting these concerns, cWDL simplifies workflow design, standardizes the handling of cross-cutting properties, and reduces the potential for errors or inconsistencies.

In the second experiment, *Code Pipeline*, which involves a simpler workflow, WDL and cWDL resulted in nearly identical code lengths. This outcome reflects their shared syntactic foundation: cWDL preserves the readability and simplicity of WDL while offering additional expressiveness through optional patterns. Conversely, CWL again required substantially more lines of code, reinforcing its verbosity even for straightforward workflows.

Finally, in the third experiment, *MLP Training*, we evaluated the languages in a more complex, data-intensive context. Here, cWDL was implemented without the use of patterns, resulting in a code length similar to that of WDL. This demonstrates that cWDL is capable of maintaining the core advantages of WDL even in pattern-free scenarios. More importantly, it underscores cWDLs flexibility: developers can selectively apply patterns when beneficial, while still leveraging a syntax that remains concise and consistent for workflows that do not require them.

5.3 Runtime performance

Execution time is a critical metric for evaluating workflow systems, particularly in the Cloud-Edge Continuum, where latency, resource constraints, and overall completion time significantly impact performance. While our previous analysis examined code length as a measure of language readability and maintainability, runtime performance directly determines the feasibility of deploying workflow solutions in production environments. The choreographed execution model of PDE, in contrast to the orchestration models of WDL and CWL, introduces a fundamentally different paradigm whose performance warrants careful evaluation. This experiment aims at evaluating the impact of the choreographed model on runtime performance compared to orchestration-based alternatives. An additional distinction is that PDE generates a Docker Compose file that can be executed manually, offering operators the flexibility to inspect and modify it prior to execution. For comparability in our experiments, we used a Bash script to automate the execution of the PDE-generated composition, thereby aligning it with the execution methods used for WDL and CWL. The three benchmark workflows employed in our study reflect common real-world computational patterns. The *MLP Data* and *Code Pipeline* workflows share elements such as file transfers via *curl*, whereas the *Molecular Docking* workflow emphasizes compute-intensive tasks without network transfer overhead.

Table 1. Comparison of execution times across different languages.

Experiment	Metric	cWDL	WDL	CWL
Code Pipeline	Mean (ms)	9863.57	38668.97	7588.72
	Min (ms)	8240.00	33235.00	6680.00
	Max (ms)	13416.00	46351.00	10505.00
	CV (%)	11.57	7.37	13.10
Molecular Docking	Mean (ms)	8460.66	6764.70	7203.21
	Min (ms)	3753.00	4024.00	6002.00
	Max (ms)	22290.00	11235.00	13173.00
	CV (%)	33.58	23.21	21.96
MLP Data	Mean (ms)	8260.62	43626.79	10630.31
	Min (ms)	6476.00	37397.00	8820.00
	Max (ms)	13586.00	51033.00	22512.00
	CV (%)	18.43	7.03	21.02

CV (%) represents the Coefficient of Variation, calculated as the standard deviation divided by the mean, expressed as a percentage.

Table 1 presents the mean, maximum, and minimum execution time statistics for the three workflows across all tested languages. Several notable patterns emerge from this data. Overall, it can be seen that cWDL and CWL perform similarly across all experiments, while WDL shows a significant slowdown compared to the others. This anomaly, only present in Code Pipeline and MLP Data, could be caused by the use of web requests, as the task that took the longest to complete was *DownloadFile*. Given that identical commands were used across all languages, this bottleneck highlights differences in execution handling between the paradigms. With reference to the code pipeline, cWDL and CWL significantly outperform WDL, due to the limitations discussed above. In particular, cWDL and CWL yield comparable results, with CWL demonstrating marginally superior performance. In the Molecular Docking experiment, which operates without web requests, WDL achieves the best performance with the fastest execution times. CWL and cWDL maintain equivalent performance characteristics throughout these tests. In the case of MLP Data experiments, the results align with the findings on code pipeline, with cWDL being the fastest.

To preliminary showcase the versatility of our PDE methodology in handling complex pipelines across diverse deployment platforms, we designed and implemented the advanced data-intensive pipelines of the UAEREP project for execution on HPC infrastructure. The UAEREP pipelines are dedicated to rainfall estimation and forecasting, leveraging a variety of heterogeneous data sources, including radar observations, satellite imagery, and precipitation gauge measurements, to generate accurate predictions. In this preliminary evaluation, for the sake of simplicity, the HPC environment is equipped with a specific configuration

of Docker, Docker Compose, and the necessary libraries to support the execution of pipeline tasks by authorized users in advance. The UAEREP pipelines were specified using cWDL and executed through the PDE framework, resulting in two deployment recipes: one targeting our local experimental environment, and the other tailored for the UAEREP HPC cluster. Although the two deployment recipes are largely similar, they differ in the data gathering phase. In the HPC cluster, data access is restricted to authorized users and confined to a designated directory, necessitating a specialized retrieval procedure. To accommodate these constraints, the data gathering tasks were polymorphically adapted within the PDE framework, ensuring full compliance with the access mechanism.

6 Related Work

Many workflow modeling languages have been proposed in the literature, with most focusing on orchestration. Some of them, such as BPEL, CWL, and WDL, have reached significant adoption. Business Process Execution Language (BPEL) [10] is a specification and execution language to describe business processes. It supports both orchestration and choreography paradigms, but it presents a series of criticalities. First, deep XPath and XSLT knowledge is required, which brings other issues such as difficult debugging and error handling, verbosity, and poor human-readability. Second, the BPEL process engine causes an additional load to services, which is suboptimal for data-intensive operations. CWL [7] is a language developed to unify the style, principles, and standards of coding pipelines, in a way that is agnostic of the hardware. It focuses heavily on making orchestrated workflows reproducible and portable, which means that it requires explicit and detailed parameter definitions, making it verbose. In contrast, WDL was developed with human readability and ease of use as core functionalities, with the downside of a less expressive syntax and poor data structures available [11]. In addition, all the above examples lack security by design. cWDL implements annotations on each task to specify security requirements, which are automatically integrated into the workflow. Deriving from WDL, cWDL inherits human readability and concise syntax. This approach, combined with the choreography methodology, results in more efficient data-intensive pipelines since tasks do not have to communicate with an orchestrator.

Other workflow modeling languages have also been formulated. The work in [12] presents Yet Another Workflow Language (YAWL), a workflow language based on Petri nets extended with specific constructs to directly support a comprehensive set of workflow patterns identified in prior research. YAWL is implemented as a hierarchy of extended workflow nets (EWF-nets) with formal semantics, introducing features like OR-joins, explicit multiple-instance handling, token removal for cancellation, and compositional correctness verification. In [13] Partially Ordered Workflow Language (POWL) is introduced, providing a structured, hierarchical framework for modeling workflows as compositions of smaller process blocks connected through sequence, choice, and partial order relations. POWL is implemented with a formal definition, compositional semantics, and a

translation to Petri nets, enabling formal analysis while maintaining an intuitive and compact modeling style .

In the service deployment domain, there exist a limited number of works that aim to intersect workflow modeling with security. Among these, the authors of [14] introduce SecFog, a methodology for securely deploying IoT applications across Cloud-Edge infrastructures by assessing security and trust. They use declarative programming to optimize deployment strategies, providing explainable security assessments. The same authors also introduce in [15] a declarative method for securely deploying FaaS orchestrations in the Edge-Cloud Continuum, focusing on preventing data leaks. The approach, implemented in the SecFaaS2Fog tool, effectively balances security, execution times, and energy consumption. In [16], authors present an automatic QoS-aware deployment solution for composed services in the Edge-Cloud Continuum. It compares security requirements on the service composition with the capabilities of a given Continuum in order to find, generate and execute suitable deployment recipes.

7 Conclusions

In this paper, we introduced a polymorphic workflow composition methodology for choreographed workflows that dynamically adapts to specific security requirements and deployment contexts. We proposed PDE, an engine generating deployment recipes that preserve non-functional properties such as security and privacy. We evaluated PDE against existing workflow execution environments across both traditional cloud platforms and high-performance computing (HPC) clusters, demonstrating its ability to perform better in terms of complexity and remain infrastructure-independent.

Acknowledgement

Research supported, in parts, by MUSA – Multilayered Urban Sustainability Action – project, funded by the European Union – NextGenerationEU, under the National Recovery and Resilience Plan (NRRP) Mission 4 Component 2 Investment Line 1.5: Strengthening of research structures and creation of R&D "innovation ecosystems", set up of "territorial leaders in R&D" (CUP G43C22001370007, Code ECS00000037); project SERICS (PE00000014) under the NRRP MUR program funded by the EU – NextGenerationEU; 1H-HUB and SOV-EDGE-HUB funded by Università degli Studi di Milano – PSR 2021/2022 – GSA – Linea 6; and Università degli Studi di Milano under the program "Piano di Sostegno alla Ricerca"; and National Center of Meteorology, Abu Dhabi, UAE under the UAE Research Program for Rain Enhancement Science (UAEREP award no. 30128758).

Views and opinions expressed are however those of the authors only and do not necessarily reflect those of the European Union or the Italian MUR. Neither the European Union nor the Italian MUR can be held responsible for them.

References

1. A. Jula, E. Sundararajan, and Z. Othman, "Cloud computing service composition: A systematic literature review," *Expert Systems with Applications*, vol. 41, no. 8, pp. 3809–3824, Jun. 2014.
2. A. Hedhli and H. Mezni, "A Survey of Service Placement in Cloud Environments," *Journal of Grid Computing*, vol. 19, no. 3, p. 23, Sep. 2021.
3. S. Moreschini, F. Pecorelli, X. Li, S. Naz, D. Hästbacka, and D. Taibi, "Cloud Continuum: The Definition," *IEEE Access*, vol. 10, pp. 131 876–131 886, 2022.
4. M. S. Nikoo, Ö. Babur, and M. Van Den Brand, "A survey on service composition languages," in *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*. Virtual Event Canada: ACM, Oct. 2020, pp. 1–5.
5. M. Anisetti, C. A. Ardagna, M. Banzi, F. Berto, R. Bondaruc, E. Damiani, A. Pedretti, A. Pisati, and A. Retico, "MUSA: A Platform for Data-Intensive Services in Edge-Cloud Continuum," in *Advanced Information Networking and Applications*, L. Barolli, Ed. Cham: Springer Nature Switzerland, 2024, pp. 327–337.
6. K. Voss, G. V. der Auwera, and J. Gentry, "Full-stack genomics pipelining with GATK4 + WDL + Cromwell," *F1000Research*, vol. 6, Aug. 2017.
7. M. R. Crusoe, S. Abeln, A. Iosup, P. Amstutz, J. Chilton, N. Tijanić, H. Ménager, S. Soiland-Reyes, B. Gavrilović, C. Goble, and T. C. Community, "Methods included: standardizing computational reuse and portability with the common workflow language," *Commun. ACM*, vol. 65, no. 6, p. 5463, May 2022. [Online]. Available: https://doi.org/10.1145/3486897
8. B. Banimfreg, E. Damiani, V. A. Gorooh, D. Axisa, L. D. Monache, and Y. Wehbe, "Innovative approach for gauge-based qpe in arid climates: comparing neural networks and traditional methods," *Journal of Big Data*, vol. 12, no. 1, pp. 1–36, 2025.
9. V. Afzali Gorooh, M. Ghazvinian, W. Hu, Q. Cao, M. Pan, B. Hassan Banimfreg, E. Damiani, A. Sengupta, D. Axisa, and L. Delle Monache, "High spatiotemporal resolution quantitative precipitation estimation over the united arab emirates," in *104th Annual AMS Meeting 2024*, vol. 104, 2024, p. 429038.
10. "Web Services Business Process Execution Language," https://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html, accessed: 2025-08-10.
11. A. E. Ahmed, J. M. Allen, T. Bhat, P. Burra, C. E. Fliege, S. N. Hart, J. R. Heldenbrand, M. E. Hudson, D. D. Istanto, M. T. Kalmbach *et al.*, "Design considerations for workflow management systems use in production genomics research and the clinic," *Scientific reports*, vol. 11, no. 1, p. 21680, 2021.
12. W. M. P. van der Aalst and A. H. M. ter Hofstede, "YAWL: Yet another workflow language," *Information Systems*, vol. 30, no. 4, pp. 245–275, Jun. 2005.
13. H. Kourani and S. J. van Zelst, "POWL: Partially Ordered Workflow Language," in *Business Process Management*, C. Di Francescomarino, A. Burattin, C. Janiesch, and S. Sadiq, Eds. Cham: Springer Nature Switzerland, 2023, pp. 92–108.
14. S. Forti, G.-L. Ferrari, and A. Brogi, "Secure Cloud-Edge Deployments, with Trust," *Future Generation Computer Systems*, vol. 102, pp. 775–788, Jan. 2020.
15. A. Bocci, S. Forti, G.-L. Ferrari, and A. Brogi, "Declarative Secure Placement of FaaS Orchestrations in the Cloud-Edge Continuum," *Electronics*, vol. 12, no. 6, p. 1332, Jan. 2023.
16. M. Anisetti, F. Berto, and R. Bondaruc, "Qos-aware deployment of service compositions in 5g-empowered edge-cloud continuum," in *2023 IEEE 16th International Conference on Cloud Computing (CLOUD)*, 2023, pp. 471–478.

Appendix A Listings

```
struct molecules {
  String x
  String y
  String z
}

start {
  call parseDb
}

task parseDb {
  input {
    File Db
  }
  command <<<
    cat ~{Db} | grep "^x" | awk '{print $2}'
    cat ~{Db} | grep "^y" | awk '{print $2}'
    cat ~{Db} | grep "^z" | awk '{print $2}'
  >>>
  runtime {
    container: "parsedb:latest",
    engine: "docker",
    dbtype: "SQL"
  }
  next {
    molecules data = molecules {
      x : read_lines(stdout())[0]
      y : read_lines(stdout())[1]
      z : read_lines(stdout())[2]
    }
    if (data.z == "0") {
      call conversion {
        In = data
      }
    }
    if (data.z != "0") {
      call docking {
        data = data
      }
    }
    result {
      String out="Db_Error"
    }
  }
}
```

Fig. A1. Variable and task definition in cWDL.

```
start {
    call parseDb with auth
}

task parseDb {
    ...

    comms {
        confidentiality: "https"
    }
    ...
    runtime {
        container: "parsedb:latest",
        engine: "docker",
        nodetype: "edge",
        ram: "16gb"
   }
}
```

Fig. A2. Application of auth and encryption annotations and constraints to a cWDL task.

```
runtime {
   container: "parsedb:latest",
   engine: "docker",
   nodetype: "edge",
   ram: "16gb"
}

comms {
   trasmission-medium: "optical_fibre",
   trasmission-technology: "single-mode"
}
```

Fig. A3. cWDL annotations in runtime and comms.

```
task docking {
   ...

   runtime {
       container: "parsedb:latest",
       engine: "docker",
       replicas: "2"
   }
   ...
}
```

Fig. A4. cWDL annotations for availability request.