Towards a Constraint-Driven Deployment Process for Cloud-Native Applications

Ouail Derghal, Jean-Christophe Bach, Fabien Dagnat

▶ **To cite this version:**

Ouail Derghal, Jean-Christophe Bach, Fabien Dagnat. Towards a Constraint-Driven Deployment Process for Cloud-Native Applications. 13th IEEE International Conference on Cloud Engineering (IC2E 2025), Sep 2025, Rennes, France. pp.85-93, ⟨10.1109/IC2E65552.2025.00024⟩. ⟨hal-05206415⟩

HAL Id: hal-05206415

https://hal.science/hal-05206415v1

Submitted on 11 Aug 2025

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

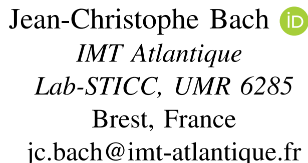
L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons CC BY 4.0 - Attribution - International License

Towards a Constraint-Driven Deployment Process for Cloud-Native Applications

Vision Paper

Ouail Derghal
IMT Atlantique
Lab-STICC, UMR 6285
Brest, France
ouail.derghal@imt-atlantique.fr

Jean-Christophe Bach 
IMT Atlantique
Lab-STICC, UMR 6285
Brest, France
jc.bach@imt-atlantique.fr

Fabien Dagnat
IMT Atlantique
Lab-STICC, UMR 6285
Brest, France
fabien.dagnat@imt-atlantique.fr

***Abstract*—Cloud-native deployments are typically ad-hoc and fragmented, leading to repeated misconfigurations and increased security risk. A challenge that arises often in modern deployment processes is the use of diverse heterogeneous technologies, such as Infrastructure-as-Code frameworks and cloud-specific tools. This causes a technological lock-in, making migration, adoption of alternative solutions, and supporting multi-cloud environments difficult. Another issue that further complicates current deployment processes is the collaboration of multiple actors during the process, such as developers, cloud architects, and business decision-makers whose roles are not well defined.**

To address these challenges, we advocate for a constraint-based approach of deployment. We emphasize security and compliance where constraints are declarative and treated as first-class inputs. This process would enable the specification of high-level constraints (*e.g.* security, cost, legal, etc.) and technical requirements in a unified and structured manner. This makes deployments portable across environments and cloud providers, auditable, and resilient to evolution. By introducing clear role separation and constraints validation, our approach promotes accountability while reducing the cognitive load on individual deployment actors.

We propose a framework that enables actors involved in the deployment to specify constraints at various stages (architecture level, implementation level, etc.). These constraints are evaluated and verified by a constraint solver, and results into a set of feasible deployment options. Deployment scripts are generated from one of the feasible deployments in order to deploy the application. It is worth stressing that our intention is not to replace existing deployment workflows, but rather to provide a complementary approach.

***Index Terms*—Constraint-Driven Deployment, Multi-Cloud Deployment, Infrastructure-as-Code**

I. Introduction

Cloud-native applications are of complex nature and their deployment requires the use and orchestration of a multitude of components, environments, tools, and configurations. Regardless of the tool maturity being used in today's deployment methodologies such as Infrastructure-as-Code (IaC) tools [1] and Continuous Integration and Continuous Deployment (CI/CD) pipelines [2], deployment processes remain complex, fractured and lacking standardization. Operation teams rely on several kinds of tools and practices which differ across organizations, and sometimes even within the same team, leading to misconfigurations, inconsistencies and security flaws [3].

Furthermore, the lack of standardization in the tools and practices not only leads to inconsistencies and increased risk of vulnerabilities but also introduces dependence on specific platforms and cloud vendors. Many teams deployment tools and APIs are tightly integrated with a particular provider's ecosystem whether for convenience or compatibility without a clear abstraction layer. Over time, this introduces a significant technological lock-in risk [4], where migrating workloads or even adopting alternative solutions becomes costly in terms of effort and downtime. As a result of the lack of standardization, actors involved into the deployment process are often locked into specific tool chains reducing their flexibility and increasing their long-term operational and security risks.

The complexity of the currently used deployment processes appears in the involvement of multiple actors and stakeholders in the deployment life cycle. Developers, operators, security engineers, and compliance engineers all have different perspectives and tools in order to accomplish a specific set of tasks at different levels of the deployment process. While this specialization can enhance productivity, often different roles collaborate together making the boundaries of different roles ambiguous and not clear [5]. It can also introduce communication gaps, and operational misalignments that make the deployments slower, less reliable and more error-prone.

Another challenge that arises in current deployment processes is the integration of constraints checking, such as security policies, compliance rules and reliability checks at various steps of the process. The way that these constraints are expressed, enforced and verified is often ad-hoc and loosely coupled with the deployment logic. This leads to a tension between deployment speed and deployment safety.

We argue for a constraint-driven deployment model that treats constraints as first-class, declarative inputs to the deployment process. By separating the specification of the de-

This work was supported partially by the France 2030 ANR Project ANR-23-PECL-0009 TRUSTINCloudS.

ployment from the mechanics of its execution, our vision aims to enable portable, auditable, and reproducible deployments that are independent of specific tool chains or cloud-provider ecosystems. Our proposed process also promotes a clear distinction between the multiple actors involved in the deployment lifecycle, ensuring accountability and role clarity across development, operations, security, and compliance domains.

We aim to provide an integration for both software deployment and infrastructure deployment aspects within a unified deployment process. In existing deployment processes, the definition of infrastructure and the deployment of the application are addressed as separate tasks, often handled independently using different tools and specifications. This separation creates a gap, when defining constraints, it is typically not possible to reference infrastructure properties within the application deployment model, or the other way around, to express deployment requirements that depend on the underlying infrastructure.

This paper is structured as follows: Section II presents the context and motivation for our work, highlighting the key challenges in modern deployment processes. Section III introduces our proposed constraint-driven deployment process and provides a detailed description of each of its steps. Section IV reviews three contributions that form the foundation of our approach and align closely with our vision. Finally, Section V summarizes our proposal, outlines the challenges involved in implementing the approach, and future evaluation.

II. Context and Motivations

A. Cloud-Native Application Deployment

Cloud-native applications are software solutions designed using a combination of specific practices and technologies to take advantages of cloud computing environments. Such environments promote elasticity, resilience, and rapid iteration in a distributed and dynamic environment [6].

Deploying cloud-native applications presents several challenges. While micro-services and component-based systems offer agility and flexibility, managing numerous independently deployed services increase the complexity in deployment, monitoring, and inter-service communication. Another challenge is regarding the security, with multiple components and increased exposure due to cloud environments, maintaining security across all levels of the application becomes challenging. Security measures need to be integrated deeply into the application architecture rather than added as an afterthought [7].

Compliance with data protection requirements such as GDPR[1] [9] or HIPAA[2] is a significant challenge for cloud-native applications. These requirements impose strict data privacy and secure storage requirements, which have the effect of making deployment processes more complex particularly in hybrid or multi-cloud environments [11].

[1]General Data Protection Regulation (European Union) [8]

[2]Health Insurance Portability and Accountability Act (United States) [10]

B. IaC and CI/CD Pipelines

The implementation of modern deployment processes requires the use of Infrastructure-as-Code tools and CI/CD pipelines [12]. IaC tools such as Terraform [13] or Ansible [14] replace traditional deployment scripts, enabling users to directly connect to cloud APIs in order to provision and configure the infrastructure required to operate applications. These tools have multiple uses and can be combined to carry out deployment processes.

The heterogeneity of these tools makes it more difficult for teams to learn and adapt to new technologies and frameworks. As different toolkits require specialized knowledge, it can be difficult for organizations to remain efficient and consistent across various deployment environments. This complexity may slow down the deployment process and make it more difficult for team members to collaborate [2].

These difficulties highlight the need for a unified strategy that captures the complexities of different IaC tools. Organizations can lower the level of technical expertise needed, accelerate the deployment process, and increase accessibility for non-technical stakeholders by making use of a system that can generate deployment scripts compatible with multiple IaC frameworks [15].

C. Multi-Actor Deployment Pipelines

A deployment process involves various stakeholders, each bringing distinct tasks, expertise, and responsibilities. Software architects focus on application structure and design, cloud architects manage infrastructure, while business actors contribute constraints. Despite their shared goal of delivering reliable and secure applications, these stakeholders often use different tools and express requirements in different formats. This may lead to misunderstandings, duplicated efforts, and misaligned decisions during deployment planning.

One of the core motivations behind this work is to address this complexity and bridge the gap between these diverse roles. By enabling all actors to express their requirements in a common language and integrating their inputs as early as possible in the deployment process. Using a structured, constraint-based model, where each stakeholder's input is formalized and validated, transforms the deployment process into a more reliable, inclusive, and collaborative task. This methodology aligns with the core principles of DevOps, which prioritize eliminating barriers and promoting cross-functional collaboration to enhance software delivery.

D. Technological Lock-In

Technological lock-in is a common challenge in modern cloud-native deployment processes. Technological lock-in occurs when organizations become dependent on a single cloud provider's proprietary technologies, making it difficult and costly to switch to alternative providers due to incompatible APIs, data formats, and service interfaces. This dependency increases costs, and limits the flexibility [16] [17].

Consequently, organizations may find themselves constrained in their ability to adapt to changing business needs.

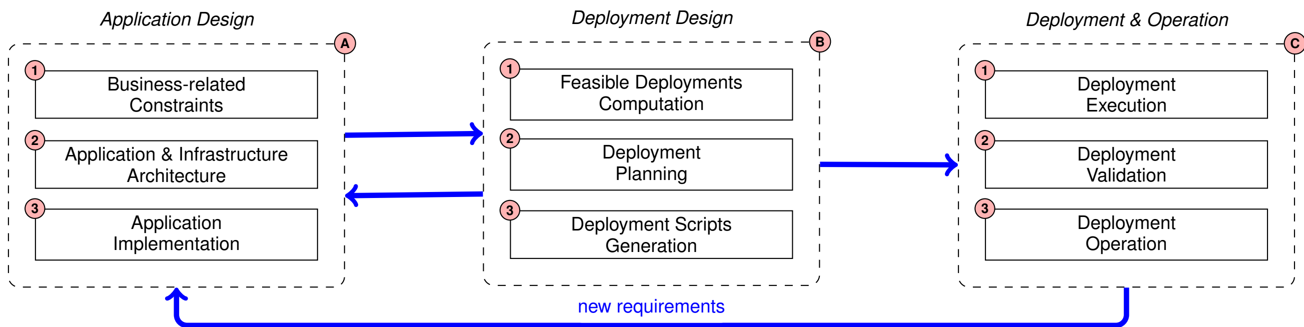


Fig. 1. Overall Steps of the Proposed Deployment Process

Addressing the technological lock-in issue requires the adoption of standardized formats and tools. Implementing these strategies would mitigate the risks associated with the lock-in and enhance the overall experience of cloud-native applications deployment.

E. Ever-Changing Requirements

The multiplicity and heterogeneity of services, providers and technologies make cloud systems complex. Many factors, such as security, business needs or external changes, can lead to redefinitions of requirements and hence to redeployments, making these systems highly dynamic.

In the context of security, maintaining systems in operational condition naturally leads to reassessment and redefinition of requirements. For example, new vulnerabilities are constantly being discovered and need to be corrected[3]. This can lead to changes in the applications deployed (deletion or drop of a vulnerable library, deployment of an updated component, introduction of new components and services, redefinition of the architecture) and therefore to changes in requirements.

Business needs may also change. An organization may decide to launch new services, discontinue existing ones, or even adapt existing infrastructures, necessitating a review of initial requirements. For instance, the introduction and generalization of home working may conflict with security constraints ("prohibition of remote access") and force the company to modify its organization and processes (*e.g.* introduction of a VPN to secure remote access).

Deployment requirements may also evolve due to external constraints such as changes in legislation (*e.g.* implementation of the GDPR) or a change at a cloud provider (addition/removal of technology support, API change or even changes in pricing). This can force the organization to review its services, infrastructures and associated deployments.

F. An Ideal Deployment Approach

An ideal deployment should allow the expression and enforcement of a wide range of requirements (technical, legal, business, and operational). It should also ensure that all stakeholders, from software and cloud architects to business experts, can contribute constraints in a unified, structured, and tool-agnostic way. The resulting plan should be compliant by design, optimized according to business priorities (cost, performance, compliance), and portable across multiple cloud providers. It should enable the automatic generation of deployment scripts using interchangeable IaC backends, reducing the technical burden and making the process accessible to non-technical contributors. In a context of continuous evolution of requirements and security threats, the deployment should also be continuously re-evaluated and that could lead to re-deployments. The process should also integrate with existing tools in order to execute deployments, not by replacing current practices, but by enhancing them with validation, traceability, and adaptability. In this work, we propose a deployment process that promotes the aforementioned features.

[3]More than 40,000 CVEs were published on https://www.cve.org/ in 2025, an increase of 38% over 2023, see https://www.cvedetails.com/ for more details.

III. Proposed Deployment Process

The deployment process that we propose do not intend to replace existing development and deployment processes, but rather to complement and enhance them. It aims to provide a standard framework for all the stakeholders contributing directly or indirectly to a deployment, by explaining each actor's responsibilities and roles. It also addresses technological lock-in, a common drawback in many current deployment solutions. Our approach relies on specifying and enforcing constraints of various types to enable valid by construction deployments based on user-imposed constraints.

Figure 1 depicts the key stages of our proposed deployment process, which is structured into three primary phases:

- *Application Design:* In this initial phase, stakeholders define the application's architecture, its required infrastructure, its implementation, and a set of business-related constraints.
- *Deployment Design:* Based on the constraints collected in the previous phase, a list of feasible deployments is computed. If no deployment is possible, we return to the previous phase. Then, one deployment plan is then selected from this list, and a corresponding set of deployment scripts is produced.

- *Deployment and Operation:* Lastly, the generated deployment scripts are executed to deploy the application and operate it on the target platform.

As suggested by the cycle in blue in the figure, these three phases can be repeated whether when an internal or an external event change a deployment constraint. Supporting such a continuous deployment is important because it allows multiple evolution scenario. For example, it helps in building incrementally our software system or in taking into account evolution of its context (*e.g.* security events, change of contracts with a cloud provider, evolution of regulation...).

Each phase involves different actors with distinct roles and responsibilities and builds upon the outputs of the preceding one, forming a workflow that transitions from abstract definition to execution. Figure 2 provides a refined and detailed view of our process, highlighting the roles of various actors, the sequence of actions, and the artifacts resulting each task. The rest of this section details each of these phases and their tasks taking care to link explanation to the figure.

A. Application Design

The *Application Design* phase can either start from scratch in the case of a new project or from what already exists. In this case, either a deployment has already been completed and new requirements have emerged that need to be taken in account or there was no possible deployment and the design must be updated.

1) Business-Related Constraints: Business actors define constraints related to their area of expertise. Their primary responsibility is to express business-related constraints on the deployment. Task (A.1) of the process shows that business actors can add new constraints to the set of deployment constraints artifact. For instance, cloud cost optimization engineers can add cost-related constraints on the resources required to execute the application. Another example of business actors could be legal experts who could define legal constraints on the application, such as data residency requirements, compliance with regulations like GDPR, or restrictions on the use of certain third-party services.

A crucial feature of the process that we propose is the incremental definition of constraints, where constraints can be progressively specified, refined and extended through the deployment lifecycle. The incremental definition of constraints allows different stakeholders to contribute domain-specific requirements at different stages of the process.

By allowing stakeholders such as cost optimization engineers and legal experts to articulate constraints related to their areas of expertise, organizations can ensure that deployments align with business objectives and regulatory requirements. Integrating these constraints early in the deployment process promotes collaboration, ensuring that technical decisions are informed by business considerations. This approach leads to compliant deployments and aligning IT operations with organizational goals and regulatory standards.

Notice that we aim to provide domain-specific abstractions to support each business domain in the goal of facilitating the task of specifying constraints for the stakeholders. For example, legal experts may only be interested in constraints related to data and data locality. By consequence, each stakeholder would be able to use its own terminology even for shared elements. For example, a legal expert could speak of services while the implementer may speak of API and the DevOps of endpoints.

2) Application & Infrastructure Architecture: The work of defining the application architecture in terms of components, services, and their relationships is done by two types of actors: *software architects* and *cloud architects*. The software architect defines the application architecture abstractly without referring to the concrete implementation of the application. For example, in the case of an e-commerce application, the software architect may define two components: one providing a front-end service and one providing a database service. The relationship between both components is expressed as a *dependency* where the front-end component requires a database service in order to fulfill its functionality. This abstract definition allows architects to focus on the structural and functional roles of each component, leaving the responsibility of implementation details such as the choice of database engine or web framework to other stakeholders. An example of a constraint that could be defined in the abstract level is the *co-locality*, the architect may enforce that the database component should not be deployed to the same resource as the front-end component. The architect may also use constraints to express non-functional requirements such as security or reliability requirements.

The cloud architect on the other hand, defines the cloud resource structure required to execute the application in an abstract manner. The cloud architect can also express constraints on the infrastructure that will be used to operate the application. This allows to define the resources required independently of low-level implementation details such as the cloud provider. For example, for the e-commerce application, the cloud architect can define abstract resources to host the front-end and the database services. The abstract resources can be a containerized runtime environments, virtual machines or any kind of computing resources. The architect can also define physical constraints on the resources, such as the amount of RAM, CPU, or the presence of a GPU unit.

Notice that at this stage, two abstract resources must not be considered distinct unless explicitly specified by a constraint. For example, both the front-end and database components could be deployed on separate computing resources or grouped onto the same resource, depending on the defined constraints.

Having application and infrastructure architecture abstractions has benefits in the aspects of constraint definition and enforcement. It allows for a separation of concerns, which enables various stakeholders to work on their own area of responsibility or expertise. This separation simplifies the specification of constraints as it maintains them consistent with the appropriate level of abstraction. It also promotes reusability and flexibility because abstract architectures and their associated constraints can be reused across different environments

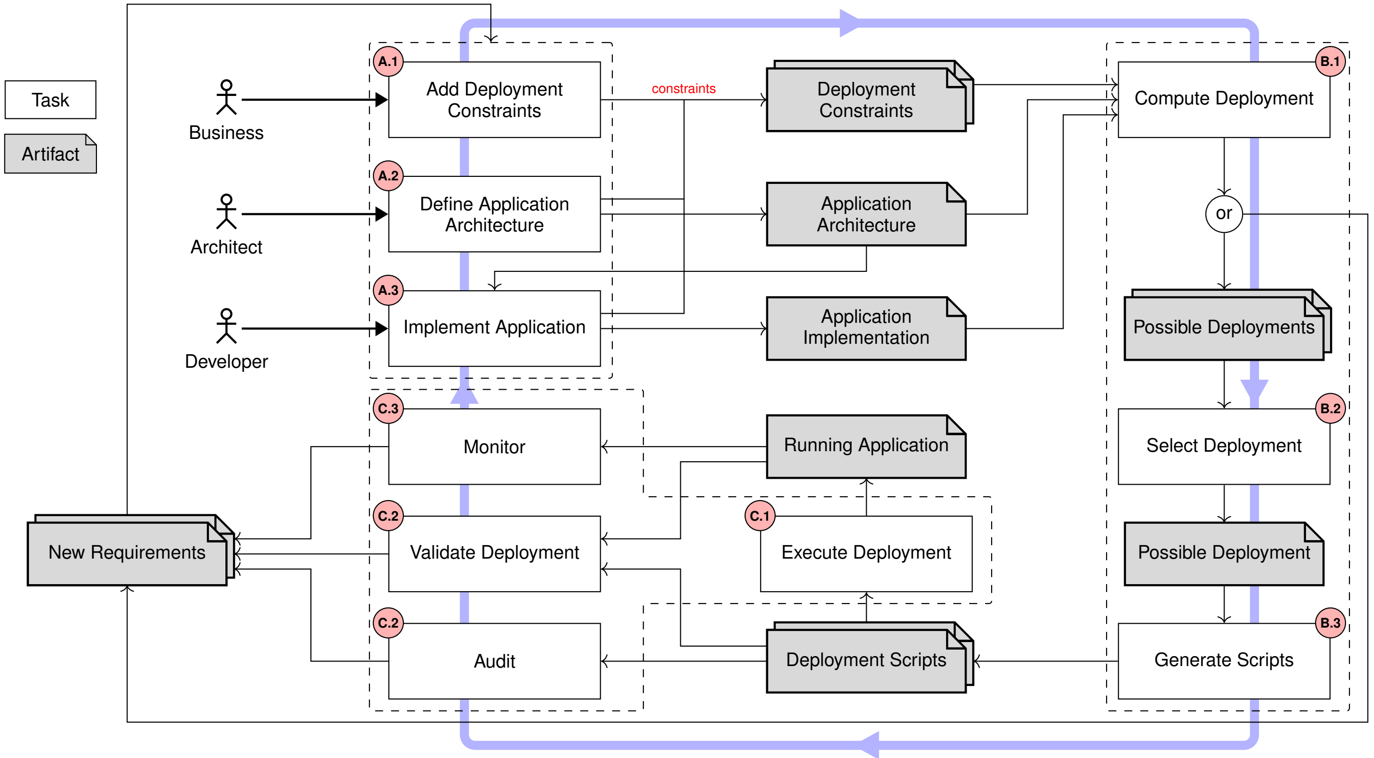


Fig. 2. Detailed Deployment Process

and deployment targets. Constraints can also be validated early in the process, before any concrete implementation is decided, which helps to catch issues earlier and reduce costly rework. These abstractions also allows for automation and optimization by enabling constraint solvers to compute valid deployment plans that satisfy user-specified requirements. In addition, they allow constraints to be stated in a cloud-agnostic manner, which reduces the technological lock-in issue. Abstract models make it easy to enforce policies uniformly across layers so that high-level constraints such as data locality or compliance regulations are maintained throughout deployment [18].

The result of this step is the *abstract architecture* of both the application and its required infrastructure resources, as well as a *set of constraints* defined in the architecture level (task A.2 of the deployment process).

3) Application Implementation: Task A.3 involves the implementation of the application. *Developers* are responsible for providing a set of concrete implementations that conform to the abstract architecture defined earlier. The implementation must fulfill the structural, functional and non-functional expectations specified by the software architect, ensuring consistency between the abstract model and its specification. In addition to the implementations, developers can define concrete-level constraints, such as requiring a specific version of a dependency or a particular runtime environment. By separating the abstract architecture from its implementations, the process allows for reusability, and early validation of constraints. It also enables multiple implementations to coexist, supporting scenarios like A/B testing or environment-specific deployments, all while maintaining alignment with business and technical requirements.

For instance, referring to the e-commerce application example, the database component could be implemented through different relational database implementations such as MySQL or PostgreSQL. These choices may vary depending on factors like performance, licensing, or team expertise. The result of this step is a list of *implementations* (for both services and components), and a *set of concrete-level constraints* that must be satisfied.

B. Deployment Design

1) Feasible Deployments Computation: Based on the constraints defined across multiple levels by software architects, cloud architects, business stakeholders, and developers, and the set of available implementations, valid deployment plans can be computed from the specified requirements. These constraints may include high-level business goals such as budget limits or data locality requirements, technical limitations like specific software versions or resource configurations, and compliance policies that must be enforced throughout the system. Each actor contributes domain-specific knowledge to the constraint set, ensuring that the final deployment respects a range of operational needs. With the abstract architecture serving as a foundation, and concrete implementations mapped onto it, we are able to explore the space of possible deployments, filter out those that violate any constraints, and generate a set of feasible, compliant deployment options. If there exists possible deployments, this approach ensures that the

deployments not only function as intended, but also align with broader organizational policies and goals. Task (B.1) takes as an input the application abstract architecture, implementations and the set of constraints defined in steps (A.1), (A.2), and (A.3), and results in a list of possible deployments artifacts.

Using constraint solvers to perform this computation introduces automation to the traditional, manual and error-prone process, reducing the likelihood of human mistakes. It also provides traceability, each selected deployment plan can be justified by a set of the original constraints. In the case where there is no solution to the constraint solving problem (*i.e.* there is no possible deployment), constraint solvers can compute the conflict core. This can help identify which constraints must be relaxed in order to obtain a valid deployment. Constraint-driven deployments also allows for optimization: among the valid deployments, solvers can prioritize those that minimize cost, maximize performance, or balance trade-offs according to user-defined objectives. This makes the deployment process not just feasible, but also aligned with operational priorities. The formalization of constraints and architecture allows adaptability, as constraints evolve (*e.g.*, new regulations, updated performance targets), the solver can re-evaluate the space of possible deployments without reworking the entire system manually.

Auditability is a core feature promoted in our deployment process. Constraints are treated as explicit, first-class entities in the deployment specification, as a result, it enables traceability throughout the deployment lifecycle. Each constraint can be mapped to a specific user, versioned and associated with the specific stage at which it is evaluated. Auditability allows the stakeholders involved in the deployment process to precisely identify which constraints were satisfied and which failed. It is worthy to note that this traceability is inherently tied to the constraint-solving algorithm used in our process. If the solver is capable of exposing justification traces, conflict sets, or unsatisfiable cores, it becomes possible to map each decision back to the original constraints. This not only supports detailed audits but also enables explainable constraint violations. Another important benefit of having auditable deployments it the support of adaptation and evolution of deployment policies. In some contexts, teams may decide to relax certain constraints to enable faster iteration, make testing easier, or address exceptional use cases.

2) Deployment Planning: From the set of the generated feasible deployments from task (B.1), a deployment plan can be chosen based on various criteria such as cost, performance, security, or geographic distribution. The selection of a deployment (task (B.2) on the detailed figure) can be an automated process or done by manual intervention, guided by the organization preferences and requirements. For an application that needs to operate on a tight budget, the deployment chooser may prioritize most cost-effective deployment, even if it sacrifices some performance. In some scenarios, the decision can also be driven by operational considerations such as optimizing latency for end users, or ensuring compatibility with existing infrastructure. The deployment selection can even be random. Choosing a deployment from a set of feasible options ensures that the final deployment aligns with business, technical, and legal priorities. The result of this step is a possible deployment artifact.

For the e-commerce application example, suppose the constraint solver proposes three possible deployment options: one using a US-based cloud provider with low latency for North American users but higher operational costs; another hosted in the European Union (EU) with strong GDPR compliance and moderate costs. Depending on the business preferences, the deployment planner could select the EU-based option to meet legal requirements around customer data protection. This ensures that the selected deployment not only meets technical requirements but also aligns with business, legal, and operational goals.

3) Deployment Scripts Generation: Task (B.3) consists of the generation of deployment scripts from the selected deployment plan, it provides a powerful bridge between high-level design and executable infrastructure. Once a feasible and constraint-compliant deployment is selected, the system can generate deployment scripts (IaC or other) in order to provision and configure the required resources. As the targeted infrastructure can be heterogeneous (*e.g.* with different technologies, at different providers), a generated script may also be a heterogeneous assembly. For instance, a part of a script could be generated in Ansible while other parts could be Python scripts using cloud vendor API. The key benefit of this approach is the ability to switch or extend the IaC backend, enabling support for multiple tools. This flexibility reduces the need for deep expertise in a specific tool and lowers the barrier for adoption across diverse teams and environments.

Supporting multiple IaC backends enhances tool interoperability and makes the deployment process future-proof, making the migration between technologies easier. By abstracting the complexity of script authoring, the process becomes accessible to non-technical experts, such as project managers or legal teams, who can influence deployments through declarative constraints rather than low-level code. This improves collaboration across roles, accelerates deployment cycles, and promotes a more inclusive and agile DevOps culture [19].

C. Deployment & Operation

1) Deployment Execution: The generated deployment scripts task (B.3) could be executed in task (C.1) using existing CI/CD pipeline solutions in order to provision and configure the resources required to operate the application. Once the scripts are generated from the selected, constraint-compliant deployment plan, they can be integrated into standard DevOps workflows. This approach ensures that the proposed process does not reinvent the wheel, but rather builds upon well-established tools and practices already in use. Instead of replacing the current deployment processes, this model complements them by bringing constraint management, and decision support into the early stages of deployment planning. As a result, organizations benefit from improved deployment

quality and traceability without disrupting their existing tool chains and workflows.

2) Deployment Validation: Once the generated deployment scripts corresponding to the selected deployment plan are executed, the necessary infrastructure is provisioned and configured in order to operate the application. To verify that the actual deployed environment matches the intended design and specifications (task C.2 on the detailed process figure), automated testing frameworks can be used. These tools provide automatic means to assess whether the running infrastructure conforms to what was generated in the deployment scripts. For example, tools like Testinfra [20] allow users to write automated tests that assert expected system states, such as package installations, open ports, running services, and file configurations, ensuring that the deployment process results in the desired infrastructure setup.

By testing and validating deployments, users can make sure that their applications are running in a correctly configured environments. It reduces the risk of misconfigurations that could lead to security vulnerabilities, service downtime, or performance issues. Adding automated validation into the deployment lifecycle allows for quicker feedback loops, enabling teams to detect and fix issues early.

3) Deployment Operation: After deployment and validation, monitoring, which is addressed in task C.3 of the process, is a critical part of maintaining the performance, and security of the deployed application and infrastructure. Monitoring tools provide continuous visibility into the runtime behavior of the system, capturing metrics such as resource utilization (CPU, memory, disk), network traffic, response times, etc. These metrics are used for detecting anomalies, diagnosing issues, and ensuring that the system operates within the defined constraints.

Integrating monitoring into the deployment process helps close the loop between planning and execution. It enables stakeholders to observe whether the deployment satisfies the initial performance, security, and compliance requirements. If not, new implementations, constraints or architectures could be changed which will trigger the re-execution of the process. Monitoring could result in changes into the design phase for future deployments, supporting continuous improvement and adaptation. This feedback loop is particularly useful when business actors impose constraints that can be monitored and enforced post-deployment. Monitoring transforms deployments from one-time events into continuously managed systems that respond to real-world conditions.

IV. Related Work

In the landscape of cloud-native and DevOps-driven software delivery, deployment processes have become increasingly complex, involving multiple actors, diverse technologies, and requirements. To address these multifaceted challenges, researchers have proposed various approaches focusing on different aspects of deployment, such as model-driven abstractions, security integration, and constraint-based optimization. This section reviews three representative works that align closely with our vision and inform the development of our proposed constraint-based deployment framework.

The DOML [21] approach introduces a model-driven paradigm that provides a unified description of the application that the user wants to deploy. It provides means to model the components of the application, the abstract infrastructure required to deploy and operate the application along with concrete provider-specific configurations. The authors promote the ease of use, reducing the complexity of technical expertise required to deploy the application, and overcoming the vendor lock-in problem by generating Infrastructure-as-Code scripts from the models defined by the user. DOML aims to allow actors with different levels of technical expertise to be able to contribute to the deployment process through a low-code interface by supporting a wide range of IaC backends like Terraform and Ansible. While DOML improves portability and allows for easier deployments, it focuses primarily on modeling and generation and does not explicitly prioritize constraint verification and role separation.

Zephyrus2 [22] optimizes the deployment problem using constraint programming (*CP*) and satisfiability modulo theories (*SMT*) [23] to compute deployment plans that satisfy user-defined constraints. The authors introduce a declarative specification language that allows users to define software components, cloud infrastructure resources, and a collection of deployment constraints, *e.g.*, resource configurations, co-location constraints, and fault tolerance policies. Zephyrus2 is able to solve complex deployment scenarios involving hundreds of components and virtual machines in seconds. Despite these strengths, Zephyrus2 is primarily appropriate for initial deployment planning and optimization. It does not have mechanisms for security constraints integration, actor-specific constraint management, and distinct accountability during deployment. While Zephyrus2 offers a solid foundation for declarative deployment, it does not fully address the operational and security challenges faced in dynamic, multi-actor, and compliance-driven cloud environments.

VeriDevOps [24] proposes a methodology for integrating security verification and monitoring into the DevOps process. It uses techniques such as automated security testing, formal security requirements through natural language processing (NLP), and runtime verification for detecting weaknesses and enforcing rules in real time. The methodology promotes a shift-left security[4], combining it in the initial phases of the software lifecycle and enabling verification during every phase, from requirements and design through to deployment and operation. VeriDevOps focuses on detection and response to vulnerabilities and assurance of security requirements verification throughout the DevOps pipeline. It does not address the design of deployment or the consideration of declarative, multi-dimensional constraints on deployment planning choices and actor responsibilities. Thus, while VeriDevOps focus is on security, it works mainly on vulnerability detection and mon-

[4] *Shift-left security* is the practice of integrating security measures early in the software development lifecycle to identify and address vulnerabilities proactively.

itoring rather than on structuring the deployment logic itself or enforcing high-level declarative constraints for deployment planning.

Our constraint-based deployment method builds on and refines concepts presented in DOML, VeriDevOps, and Zephyrus2. Like DOML, our model aims to separate deployment description from execution by providing an additional layer of abstraction, which allows for supporting greater portability and weaker tool coupling. Unlike VeriDevOps, which focuses on runtime monitoring and security verification, our process embeds compliance mechanisms into the deployment logic itself such that all paths of execution are adhering to the original constraints. This makes our solution particularly well-suited for cloud environments under regulatory oversight or those requiring strict control over infrastructure governance. From Zephyrus2, we take the idea of constraint-based planning through declarative specification to automatically guide decision-making over advanced deployment options.

All of these systems aim to improve the deployment process, however, they treat constraints as either part of system modeling or as optimization constraints specific to initial deployment planning. In contrast to that, our system proposes to treat constraints as first-class, declarative inputs. This allows the users to define not only technical requirements, but also security measures and regulatory requirements. By making constraints an integral part of the deployment process, we enable the automatic identification of secure and compliant deployment plans, and direct actor accountability.

V. Conclusion

Modern deployment processes are of a complex nature, fragmented, and difficult to manage due to the use of heterogeneous tools and the involvement of stakeholders with diverse roles and expertise. The orchestration of multiple frameworks in cloud-native deployments leads to a lack of standardization, increasing the risk of inconsistencies, misconfigurations, and security vulnerabilities, while also causing technological lock-in that makes migration or adoption harder. The diversity of actors and their disconnected workflows makes the collaboration challenging, which contributes to communication gaps and deployment errors. Constraint checking is typically ad-hoc, making it difficult to ensure continuous compliance with requirements in a structured and traceable manner.

In this work, we proposed a vision that addresses the challenge of unifying software and infrastructure deployment through a constraint-based approach. We advocated for a deployment process that is driven by constraints, where these constraints are defined early, integrated throughout the workflow, and treated as first-class inputs. By making constraints explicit rather than secondary checks or ad-hoc validations, the deployment process can become tool-independent, portable, auditable, and reproducible.

A constraint-driven approach enables stakeholders from diverse domains such as software architects, infrastructure engineers, or legal experts to express their requirements in a formalized way. This early integration ensures that all requirements are taken into consideration when computing possible deployments, reducing the risk of misalignment or violations later in the lifecycle. Our approach would leverage constraint solvers to validate the user-defined constraints, and to generate valid deployment plans that satisfy these requirements. It integrates with Infrastructure-as-Code tools to produce executable deployment scripts.

Challenges: To adopt this approach, we need to address several challenges. One of the obstacles lies in the modeling of constraints. In real-world deployments, constraints come from multiple stakeholders which expresses their requirements in very different forms and at different level of abstraction. Bringing all these constraints into a unified framework requires the definition of a set of constraint modeling languages and abstraction layers that can capture different forms of constraints in a structured, machine-readable manner. Furthermore, each stakeholder may use domain specific terms to refer common concepts. This require an alignment phase that must be taken into account in the unified framework. Notice that managing correspondances between constraints expressed in different ways may need to be organization specific.

Another challenge is the generation of IaC deployment scripts from the set of computed deployment plans. This task is non-trivial due to the wide variety of existing IaC tools, each with its own syntax, capabilities, and abstraction levels. Furthermore, multi-cloud deployments usually require combining scripts using various tools and API. Generating deployment scripts that are consistent, correct, and optimized across multiple tools and platforms requires the definition of a common intermediate abstraction layer that decouples the logical deployment from the concrete implementation details of individual IaC backends.

Ensuring traceability, auditability, and accountability is a core challenge in the proposed constraint-driven deployment approach. Each constraint must be explicitly linked to its author, and associated with the deployment stage at which it is applied. Achieving this requires the underlying constraint solver to support explainability features such as justification traces and conflict sets. Allowing each decision or failure to be traced back to its origin. This level of transparency offers an informed decision-making when policies evolve or constraints need to be relaxed for exceptional scenarios.

Future Work: Our aim is to cope with the challenge of defining a framework for expressing constraints using various domain specific languages while supporting their unification. Then, these constraints would be solved following an approach similar to Zephyrus2. Here, we intend to work both on the nominal flow where the deployment problem is solvable but also on the error flow when no solution exists. In case of existing deployment, we will work on extending DOML to generate the needed deployment scripts. Lastly, as our focus is on security, we plan to study in details the monitoring of the running system and of its security context and the new requirements they could add. Such requirement evolution must then be fed to the application design to start a new deployment iteration.

This work should lead to concrete tools that will allow us to experimentally implement our proposed deployment process.

Evaluation of the future work: The evaluation of our approach is an important step of the future work. Our goal is to validate each step of the proposed constraint-driven deployment process using real-world deployment scenarios. This evaluation will focus on assessing the applicability and expressiveness of the framework in practical scenarios, ensuring that the user-defined constraints can be modeled, processed, and verified within the system. To carry out this validation, we plan to implement a complete deployment workflow for a complex, real-world application such as GitLab [25]. This choice is motivated by the application's architectural complexity, its multi-service composition, and the range of deployment requirements it typically demands.

Finally, such a new deployment process also requires to be confronted to current organizational practices. We therefore intend to assess its possible adoption through an empirical study based on a survey aimed at the various actors that we propose to involve in the deployment management.

References

[1] A. Rahman, R. Mahdavi-Hezaveh, and L. Williams, "A systematic mapping study of infrastructure as code research," *Information and Software Technology*, vol. 108, pp. 65–77.

[2] M. Shahin, M. A. Babar, and L. Zhu, "Continuous integration, delivery and deployment: a systematic review on approaches, tools, challenges and practices," *IEEE access*, vol. 5, pp. 3909–3943, 2017.

[3] M. Shahin, M. A. Babar, M. Zahedi, and L. Zhu, "Beyond continuous delivery: An empirical investigation of continuous deployment challenges," in *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, Nov. 2017, pp. 111–120.

[4] O. C. Adeusi, Y. O. Adebayo, P. A. Ayodele, T. T. Onikoyi, K. B. Adebayo, and I. O. Adenekan, "IT standardization in cloud computing: Security challenges, benefits, and future directions," *World Journal of Advanced Research and Reviews*, vol. 22, no. 3, pp. 2050–2057, 2024.

[5] D. Sokolowski, "Deployment coordination for cross-functional DevOps teams," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2021. ACM, 2021, pp. 1630–1634.

[6] S. Deng, H. Zhao, B. Huang, C. Zhang, F. Chen, Y. Deng, J. Yin, S. Dustdar, and A. Y. Zomaya, "Cloud-native computing: A survey from the perspective of services," *Proceedings of the IEEE*, vol. 112, no. 1, pp. 12–46, Jan. 2024.

[7] D. Gannon, R. Barga, and N. Sundaresan, "Cloud-native applications," *IEEE Cloud Computing*, vol. 4, no. 5, pp. 16–21, Sep. 2017.

[8] European Parliament and Council of the European Union, "General Data Protection Regulation," European Parliament and Council of the European Union, 2016. [Online]. Available: https://eur-lex.europa.eu/legal-content/EN/TXT/PDF/?uri=CELEX:32016R0679

[9] N. A. Zaguir, G. H. de Magalhães, and M. de Mesquita Spinola, "Challenges and enablers for gdpr compliance: Systematic literature review and future research directions," *IEEE Access*, vol. 12, pp. 81 608–81 630, 2024.

[10] A. Act *et al.*, "Health insurance portability and accountability act of 1996," *Public law*, vol. 104, p. 191, 1996.

[11] V. U. Ugwueze, "Cloud native application development: Best practices and challenges," *International Journal of Research Publication and Reviews*, vol. 5, pp. 2399–2412, 2024.

[12] M. Guerriero, M. Garriga, D. A. Tamburri, and F. Palomba, "Adoption, support, and challenges of infrastructure-as-code: Insights from industry," in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, Sep. 2019, pp. 580–589.

[13] "Terraform," 2025, accessed: 2025-05-08. [Online]. Available: https://www.terraform.io/

[14] "Ansible," 2025, accessed: 2025-05-08. [Online]. Available: https://ansible.com/

[15] J. Sandobalin, E. Insfran, and S. Abrahão, "Argon: A model-driven infrastructure provisioning tool," in *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, Sep. 2019, pp. 738–742.

[16] J. Opara-Martins, R. Sahandi, and F. Tian, "Critical analysis of vendor lock-in and its impact on cloud computing migration: a business perspective," vol. 5, no. 1, p. 4.

[17] J. Hong, T. Dreibholz, J. A. Schenkel, and J. A. Hu, "An overview of multi-cloud computing," in *Web, Artificial Intelligence and Network Applications*, L. Barolli, M. Takizawa, F. Xhafa, and T. Enokido, Eds. Springer, 2019, pp. 1055–1068.

[18] L. Wang, J. P. Near, N. Somani, P. Gao, A. Low, D. Dao, and D. Song, "Data capsule: A new paradigm for automatic compliance with data privacy regulations," in *Heterogeneous Data Management, Polystores, and Analytics for Healthcare*, V. Gadepally, T. Mattson, M. Stonebraker, F. Wang, G. Luo, Y. Laing, and A. Dubovitskaya, Eds. Springer, 2019, pp. 3–23.

[19] G. Novakova Nedeltcheva, A. De La Fuente Ruiz, L. Orue-Echevarria Arrieta, N. Bat, and L. Blasi, "Towards supporting the generation of infrastructure as code through modelling approaches - systematic literature review," in *2022 IEEE 19th International Conference on Software Architecture Companion (ICSA-C)*, March 2022, pp. 210–217.

[20] "Testinfra," 2025, accessed: 2025-05-08. [Online]. Available: https://testinfra.readthedocs.io/en/latest/

[21] M. Chiari, B. Xiang, G. N. Nedeltcheva, E. Di Nitto, L. Blasi, D. Benedetto, and L. Niculut, "DOML: a new modelling approach to infrastructure-as-code," in *International Conference on Advanced Information Systems Engineering*. Springer, 2023, pp. 297–313.

[22] E. Ábrahám, F. Corzilius, E. B. Johnsen, G. Kremer, and J. Mauro, "Zephyrus2: On the fly deployment optimization using smt and cp technologies," in *International Symposium on Dependable Software Engineering: Theories, Tools, and Applications*, M. Fränzle, D. Kapur, and N. Zhan, Eds. Springer, 2016, pp. 229–245.

[23] C. Barrett and C. Tinelli, *Satisfiability Modulo Theories*. Springer, 2018, pp. 305–343.

[24] E. P. Enoiu, D. Truscan, A. Sadovykh, and W. Mallouli, "VeriDevOps software methodology: Security verification and validation for devops practices," in *Proceedings of the 18th International Conference on Availability, Reliability and Security*, ser. ARES '23. ACM, 2023, pp. 1–9.

[25] "Gitlab," 2025, accessed: 2025-05-08. [Online]. Available: https://docs.gitlab.com/development/architecture/