UDC 004.89:004.82:004.75
DOI: 10.15587/2706-5448.2026.365453

Ihor Bibichkov,
Olena Shevchenko,
Oleksandr Shevchenko,
Oleksandr Stopin

DEVELOPING AN ONTOLOGY-MEDIATED SEMANTIC CONTROL METHOD FOR IMPROVING THE RELIABILITY OF LLM-GENERATED INFRASTRUCTURE-AS-CODE

The object of research is an LLM-assisted process of Infrastructure-as-Code (IaC) generation for cloud deployment. The problem is that large language models can convert user requests into syntactically valid IaC, but such generated IaC fails deployment. This occurs because such artefacts often omit required cloud relations or violate policy constraints, inaccurately interpret user intent, or lack rollback logic. As a result, the research proposes a Semantic Infrastructure-as-Code (SIaC) pipeline, where instead of a final IaC, the LLM first generates a candidate infrastructure graph. Further, this graph is completed and validated using ontology reasoning, SHACL/SPARQL validation, and other formal procedures before the generation of the final backend artifact. According to the 60 AWS ECS/Fargate-oriented scenarios that were tested in a LocalStack-based AWS emulation environment, Full SIaC achieved 74.4% sandbox deployment success, compared with 43.6% for direct LLM-to-IaC and 59.0% for LLM + RAG-to-IaC. Compared with the baselines, SIaC improved semantic intent coverage, dependency completeness, and policy compliance. It also reduced invalid dependencies and the need for manual corrections. This improvement is achieved because reliability is checked through an inspectable knowledge graph, not only through prompts and generated text. This graph supports the detection and correction of dependencies, placements, policies, and safety issues before deployment. In addition to higher deployment success, SIaC also provides a more controllable and manageable transformation path from natural-language intent to executable infrastructure.

This method is suitable if the user's requirements can be mapped to the maintained provider ontologies and if the additional time spent on analysis is considered an acceptable trade-off for improved reliability, auditability, and reduced maintenance.

Keywords: *Semantic Infrastructure-as-Code, infrastructure graph, semantic control, SHACL/SPARQL validation, dependency completeness.*

Received: 11.04.2026
Received in revised form: 13.06.2026
Accepted: 23.06.2026
Published: 25.06.2026

© The Author(s) 2026
This is an open access article
under the Creative Commons CC BY license
https://creativecommons.org/licenses/by/4.0/

How to cite
Bibichkov, I., Shevchenko, O., Shevchenko, O., Stopin, O. (2026). Developing an ontology-mediated semantic control method for improving the reliability of LLM-generated Infrastructure-as-Code. Technology Audit and Production Reserves, 3 (2 (89)), 121–133. https://doi.org/10.15587/2706-5448.2026.365453

1. Introduction

In recent years, cloud infrastructure is managed primarily through machine-readable configurations rather than manual, step-by-step processes. Terraform [1] and similar tools, such as OpenTofu [2], allow describing resources, dependencies, and execution plans in a declarative way, thereby improving deployment reproducibility and auditability. However, declarative configuration does not eliminate the complexity of cloud deployment. It became clear that understanding provider schemas, implicit dependencies, security constraints, and the operational trade-offs behind them is necessary.

The application of Large Language Models (LLMs) for Infrastructure-as-Code (IaC) generation has further intensified this problem. LLMs can reduce the expertise required for cloud automation. However, syntactically valid generated code is not guaranteed to be successfully deployable. Recent deployability researches reported that generated IaC can compile or pass validation but still fail to deploy. Moreover, generated IaC often differs from the user's intended outcome or does not pass security checks [3]. Multi-format IaC benchmarks indicated the same: while today's LLMs can produce syntactically valid artefacts, they still struggle with semantic alignment and complex infrastructure patterns [4].

This practical contradiction motivated this work. Cloud engineering teams require faster and more accessible deployment automation. However, direct natural-language-to-IaC workflows cannot reliably verify whether the infrastructure meets cloud-level requirements before deployment. Existing IaC tools address this issue only partially. Terraform/OpenTofu builds a plan from the difference between the desired configuration and the current state, builds dependency graphs, checks that they have no cycles, and creates resources once their dependencies are in place [5]. These operations occur after the desired configuration has already been written, and these tools do not account for the original user intent that led to that configuration.

Knowledge-driven approaches to cloud platforms also exist. The earlier research [6] proposed an ontological approach to the intellectualization of cloud computing platforms and introduced an intelligent web-service model. SIaC builds on this idea, but narrows the scope. Rather than attempting to model the entire cloud platform as intelligent, it introduces a semantic control layer on top of LLM-assisted IaC generation.

Other approaches rely on explicit models. TOSCA offers a model-driven language for describing application components, their relation-

ISSN-L 2664-9969; E-ISSN 2706-5448

ships, topology, and lifecycle processes [7]. Ontology-based model-driven engineering uses deployment knowledge and reasoning to auto-complete TOSCA models, detect code smells, detect errors, and match model elements [8]. The earlier work on ontological knowledge-base productivity showed that combining reasoning mechanisms makes ontology-driven systems more practical to process [9]. These researches demonstrate the usefulness of explicit semantic models. However, this work does not address an LLM-mediated path from natural language to IaC, where candidate deployment intent must be completed, validated, rejected when it is unsafe, and transformed into executable Terraform/OpenTofu artifacts.

Recent benchmarks and frameworks for natural-language IaC generation emphasize deployability, user intent alignment, security compliance, feedback, decomposition and validation rather than syntax alone [3, 4, 10, 11]. Although generating content using data from external sources can expand the context available to an LLM, information retrieval alone cannot ensure formal graph completion, closed-world validation, or policy-aware action planning. Zodiac demonstrated that failures in IaC can be caused by semantic mismatches between IaC scripts and cloud-level requirements [12]. Meanwhile, Open Policy Agent (OPA) and CloudFormation Guard support policy-as-code evaluation for structured artifacts [13, 14]. Despite their usefulness, these tools cannot infer missing infrastructure dependencies or translate natural-language deployment intent into a graph of the desired state.

Therefore, a semantic control layer is required to integrate natural-language deployment intent with executable IaC. Existing IaC validation tools mainly operate after a configuration has been generated. While they can check syntax, policy compliance and deployment plans, they cannot reconstruct the original natural-language intent or infer omitted infrastructure dependencies prior to backend code generation. This creates a discrepancy between user intent, generated IaC artefacts, and deployable cloud infrastructure. A semantic control layer can address this issue. It can ground the intent, complete missing cloud dependencies and verify mandatory resource properties before traditional IaC tools are invoked.

The object of research is LLM-assisted IaC generation process for cloud deployment. It connects high-level intent with operational semantics and the underlying back-end IaC artefacts.

The aim of research is to develop an ontology-mediated semantic control method for improving the reliability of IaC generation by LLM.

The practical aim is to transform natural language cloud deployment requests into desired-state graphs that can be validated and converted to a safe backend IaC plan prior to execution.

The research tests three hypotheses. H1 is that ontology-based IaC generation is more reliable and better aligns with the user's intent than a direct transition from LLM to IaC. H2 is that infrastructure ontology reduces the number of missing and incorrectly ordered dependencies. H3 asserts that validation using SHACL/SPARQL identifies cloud configuration errors that conventional syntax checking misses.

To achieve this aim, the research solves the following objectives:

– to formalize a graph-based semantic control model and transformation pipeline, and to implement it as AWS ECS/Fargate-oriented prototype;

– to design a reproducible benchmark and validation protocol for evaluating LLM-assisted IaC generation in AWS ECS/Fargate-oriented cloud deployment scenarios;

– to experimentally compare Full SIaC with the several baselines using operational, semantic, dependency, policy, rollback, manual-correction and overhead metrics;

– to evaluate the influence of individual SIaC components through ablation analysis.

2. Materials and Methods

2.1. Research scope and experimental material

The input material used in the experiments includes the SIaC ontology modules, the benchmark deployment scenarios, and the generated semantic graphs.

The research kept the evaluation deliberately narrow. It is focused on AWS ECS/Fargate-oriented deployments of containerized web applications run under AWS-compatible emulated conditions with LocalStack. The infrastructure model covers VPCs, public and private subnets, route tables, internet gateways/NAT gateways, security groups, IAM roles and policies, ECR, ECS clusters, task definitions, ECS services, application load balancers, target groups, listeners, and CloudWatch logs. Some resources are optional and only appear when required by a scenario: an RDS database, an S3 bucket. The backend target specified for the benchmark is HCL-style Terraform/OpenTofu configuration. Pulumi, AWS CDK, and the AWS Cloud Control API will be covered in the future.

2.2. Evaluation metrics

A specialized scenario benchmark was developed and employed to evaluate the prototype. This section explains the metrics used to compare IaC generation methods in this benchmark.

Let the benchmark contain N deployment scenarios. The benchmark scenario set is denoted as $\mathcal{T}=\{1,2,\ldots,N\}$.

The set of evaluated baselines is denoted as $\mathcal{M}=\{m_1,m_2,\ldots,m_q\}$.

For each scenario i, the evaluation requires a gold set of semantic requirements R_i^{gold} and a gold set of required dependencies $R_{i,m}^{impl}$. The generated implementation is then compared with set D_i^{gold} – gold dependency set for scenario i, and $D_{i,m}^{impl}$ – dependencies generated or inferred by method m for the same scenario i. The following indicator is used for binary outcomes

$$1[x]=\begin{cases}1, & \text{if condition } x \text{ is true,}\\ 0, & \text{otherwise.}\end{cases}$$

Taken together, the metrics capture operational correctness, semantic alignment, dependency quality, policy compliance, rollback behavior, the manual correction effort, and the planning overhead (Table 1).

The total planning time for a method can be decomposed as follows

$$T_m=T_m^{LLM}+T_m^{RAG}+T_m^{reasoning}+T_m^{SHACL}+T_m^{policy}+T_m^{diff}+T_m^{backend},$$

where T_m^{LLM} – time spent generating the candidate graph or IaC artifact; T_m^{RAG} – time spent retrieving provider documentation, examples, or best practices; $T_m^{reasoning}$ – time spent on ontology closure and dependency inference; T_m^{SHACL} – time spent on structural graph validation; T_m^{policy} – time spent on security, governance, and provider-specific checks; T_m^{diff} – time spent on identity resolution, canonicalization, and typed delta computation.

Planning overheads are not evaluated in isolation. Rather, they are considered in relation to deployment success, policy compliance, and the effort required for manual corrections. In applied settings, a slower method may be acceptable if it significantly reduces failed deployments and unsafe configurations.

Error analysis determines why methods fail. Each scenario that does not succeed is given one or more error labels, and these labels can overlap because a single deployment failure can often be caused by several different things at once

$$\text{ErrRate}_{c,m}=\frac{1}{N}\sum_{i=1}^{N}1\left[c\in E_{i,m}\right].$$

In this formula, c denotes an error category, and $E_{i,m}$ denotes the set of error categories observed for scenario s under method m. The error categories are listed in [16].

Table 1

Core evaluation metrics

Metric	Purpose	Definition
Deployment Success Rate (DSR) *Direction:* Higher is better	Measures whether the generated artifact successfully provisions the intended infrastructure in the Dockerized LocalStack-based AWS emulation environment	$\mathrm{DSR}_m=\frac{1}{N}\sum_{i=1}^{N}s_{i,m}$, where $s_{i,m}$ – a binary success indicator for scenario i under method m
First-Attempt Success (FAS) *Direction:* Higher is better	Measures whether deployment succeeds without iterative repair	$\mathrm{FAS}_m=\frac{1}{N}\sum_{i=1}^{N}f_{i,m}$, where $f_{i,m}$ is 1 only when scenario i succeeds on the first attempt under method m
Semantic Intent Coverage (SIC) *Direction:* Higher is better	Measures how much of the user intent is represented in the validated graph and backend artifact. This metric evaluates semantic alignment rather than syntactic validity. The design of SIC follows earlier work on semantic annotation similarity for process profiles by comparing structured descriptions at the semantic, rather than textual or syntactic, level [15]	$\mathrm{SIC}_m=\frac{1}{N}\sum_{i=1}^{N}\frac{\left\lvert R_i^{gold}\cap R_{i,m}^{impl}\right\rvert}{\left\lvert R_i^{gold}\right\rvert}$
Dependency Completeness (DC) *Direction:* Higher is better	Measures whether required infrastructure dependencies are represented or inferred	$\mathrm{DC}_m=\frac{1}{N}\sum_{i=1}^{N}\frac{\left\lvert D_i^{gold}\cap D_{i,m}^{impl}\right\rvert}{\left\lvert D_i^{gold}\right\rvert}$
Invalid Dependency Rate (IDR) *Direction:* Lower is better	Measures how many generated or inferred dependencies are incorrect	$\mathrm{IDR}_m=\frac{1}{N}\sum_{i=1}^{N}\frac{\left\lvert D_{i,m}^{impl}\setminus D_i^{gold}\right\rvert}{\max\left(1,\left\lvert D_{i,m}^{impl}\right\rvert\right)}$, the denominator uses a maximum with 1 to avoid division by zero in scenarios where a method produces no dependencies
Policy Compliance Rate (PCR) *Direction:* Higher is better	Measures compliance with security, governance, and provider-specific policies	$\mathrm{PCR}_m=1-\frac{\sum_{i=1}^{N}V_{i,m}^{policy}}{\sum_{i=1}^{N}V_i^{checked}}$, where the numerator is the number of violated policy checks, and the denominator is the number of policy checks evaluated across all scenarios
Rollback Correctness (RC) *Direction:* Higher is better	Measures whether failed or interrupted plans restore a consistent state	$\mathrm{RC}_m=\frac{N_m^{RC}}{\max\left(1,N_m^{fail}\right)}$
Manual Correction Effort (MCE) *Direction:* Lower is better	Measures the amount of human correction required before successful deployment	$\mathrm{MCE}_m=\frac{1}{N}\sum_{i=1}^{N}e_{i,m}$
Planning Overhead (OH) *Direction:* Lower is better	Measures the additional time introduced by reasoning, validation, graph differencing, and backend generation compared with a baseline method	$\mathrm{OH}_{m,base}=T_m-T_{base}$

For each method, the error analysis reports both the raw counts and the per-scenario rates. This enables the identification of failure types reduced by ontology reasoning, SHACL validation, policy checks, graph differencing, and compensation planning.

Some scenarios are intentionally unsafe or self-contradictory. For such scenarios, the expected behaviour is to reject the request safely or escalate it for human review, and this is captured by the Safe Rejection Rate (SRR)

$$\mathrm{SRR}_m=\frac{N_m^{correct_reject}}{\max\left(1,N^{unsafe}\right)},$$

where $N_m^{correct_reject}$ – number of unsafe or contradictory scenarios correctly rejected or escalated by method m; N^{unsafe} – total number of unsafe or contradictory scenarios in the benchmark.

A scenario is counted as correctly rejected when the method detects a safety, policy, or feasibility violation and stops before generating or applying an unsafe infrastructure change.

2.3. Statistical analysis and reproducibility

Because the same benchmark scenarios are evaluated under all methods, the experimental design is paired.

For each metric, the pairwise improvement of Full SIaC over a baseline is reported as

$$\Delta M_{F,base}=M_F-M_{base}.$$

For binary paired metrics such as DSR and FAS, the research uses McNemar's test and reports paired bootstrap 95% confidence intervals over the scenarios. For continuous metrics such as SIC, DC, IDR, MCE, and OH, the research uses Wilcoxon signed-rank tests with a Holm-Bonferroni correction. Each confidence interval was computed using 10,000 bootstrap resamples over the scenario-level paired differences.

To ensure reproducibility, every benchmark run records the essential per-scenario fields that are listed in [16].

3. Results and Discussion

3.1. Formal graph-based semantic control model and its implementation as AWS ECS/Fargate-oriented prototype

The general semantic control model includes a graph representation of the user's deployment intents and a formal pipeline for transforming it into validated backend code infrastructure artefacts (Fig. 1).

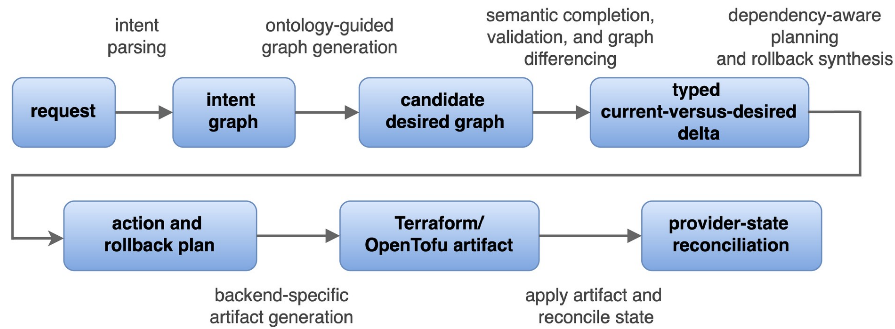

Fig. 1. SIaC semantic transformation pipeline

The LLM output for the user request is the initial intent and a candidate graph. SIaC treats this output as a preliminary graph that must be completed and validated before backend IaC generation. The semantic layer applies ontology reasoning to identify missing resources and dependencies and uses SHACL/SPARQL, provider schema, and policy checks to eliminate invalid or unsafe configurations. Finally, it compares the validated graph with the current infrastructure state. The resulting typed delta is converted into lifecycle-aware actions fitted with rollback guards and compensating actions, which are then lowered into the backend IaC. Once everything has been processed, the observed provider state is reconciled with the expected semantic graph to identify any remaining discrepancies.

The ontology module links plain-language deployment intent to cloud resources, validation constraints, lifecycle actions, and backend-specific IaC artefacts. It provides the pipeline with an intermediate representation that can be inspected, completed by reasoning, checked by closed-world validation, and converted by the system into an action plan and then into backend-specific artefacts. Currently, the ontology comprises 90 classes, 54 object properties, 49 data type properties, 13 SHACL node shapes, 8 SPARQL CONSTRUCT rules, 7 named security policy constraints and 15 Terraform backend mapping templates. The SHACL layer also adds 35 executable validation checks.

The ontology supports four core SIaC functions: grounding the intent, inferring dependencies, checking validation and policy, and planning at the level of individual actions. It is organized in a modular structure. The upper-level modules define concepts that do not depend on any provider, while the domain modules specialize them for deployment, validation, lifecycle, and backend-mapping work (Table 2). At this stage, domain modules contain only concepts and relations for the selected AWS ECS/Fargate scope.

Examples of basic ontology concepts are given in Table 3.

In addition to the core classes, the ontology defines a small set of relations (Table 4) that facilitate dependency inference, validation, state-transition planning and backend mapping. The central relation is siac:dependsOn, which is a general dependency between infrastructure resources. More specific relations, such as placedInSubnet, usesRole, routesTo and registeredInTargetGroup, are defined as sub-properties of dependsOn. This allows the reasoner to read semantic dependencies and action-ordering constraints from them.

The ontology also includes domain-specific inference patterns that are used to complete the incomplete graphs that an LLM produces. Table 5 lists a representative set of these rules, written as OWL/RDFS axioms, SPARQL CONSTRUCT rules, or rule-based reasoning components.

In addition, the ontology includes domain-specific SHACL validation constraints. Reasoning can infer which elements should be present, while closed-world validation can check whether a graph is operationally deployable. Table 6 lists a representative set of validation and policy rules.

Table 2

Ontology module decomposition

Ontology module	Responsibility
siac-core	Defines provider-independent upper-level concepts such as Resource, Dependency, Capability, and Requirement, as well as core relations connecting user intent, resources, and dependencies
siac-cloud-resource	Specializes abstract resources into generic cloud-managed resource categories, including compute, networking, storage, IAM, and observability resources
siac-networking	Describes VPCs, subnets, route tables, gateways, security groups, load balancers, listeners, target groups, and ingress/egress relations
siac-iam	Describes identity and access management concepts, including roles, policies, permissions, trust relationships, execution roles, and task roles
siac-container-ecs	Describes AWS ECS/Fargate container deployment concepts, including ECS clusters, task definitions, ECS services, container images, ECR repositories, and service-to-target-group relations
siac-storage	Describes stateful and object storage resources, including RDS instances, DB subnet groups, S3 buckets, backup policies, versioning policies, and storage safeguards
siac-state-lifecycle	Represents current, desired, previous, failed, rolled-back, and historical states, as well as lifecycle transitions between infrastructure states
siac-action-planning	Represents lifecycle actions, action dependencies, execution plans, rollback actions, compensating actions, retry actions, and human-review actions
siac-policy-validation	Defines validation and governance concepts, including SHACL constraints, SPARQL checks, security rules, provider constraints, policy-as-code rules, and human-review conditions
siac-backend-mapping	Defines mappings from semantic resources and actions to backend-specific artifacts such as Terraform/OpenTofu HCL, Pulumi code, AWS CDK constructs, or direct cloud API calls

Table 3

Basic ontology vocabulary

Concept	Modules	Role in SIaC
Resource	siac-core	Abstract parent for Cloud Resource, Virtual Private Cloud, Subnet, Security Group, IAM Role, ECR Repository, ECS Task Definition, ECS Service, ALB, Target Group, RDS Instance, S3 Bucket
Dependency	siac-core	Typed relation such as dependsOn, attachedTo, placedInSubnet, allowsIngressFrom, usesRole, exposesPort, storesImageIn, routesTo
Capability	siac-core	Exposed capability such as HTTP ingress, private persistence, image registry, log sink, database connectivity
Requirement	siac-core	Required capability or condition that must be satisfied before a resource can be created or used
State	siac-core	Current, desired, previous, failed, rolled-back, and historical states represented as named graphs
Action	siac-action-planning	Create, update, delete, no-op, validate, rollback, retry, or human-review action
Constraint	siac-policy-validation	SHACL shape, OWL class restriction, SPARQL rule, policy-as-code rule, or provider schema constraint
Mapping	siac-backend-mapping	Provider/backend-specific transformation from semantic resource types to Terraform/OpenTofu/Pulumi/AWS API constructs

Table 4

Basic ontology relations

Relation	Domain	Range	Description
dependsOn	Resource	Resource	Represents a general dependency between infrastructure resources
requiresCapability	Resource or Requirement	Capability	Indicates that a resource or requirement needs a capability
providesCapability	Resource	Capability	Indicates that a resource provides a capability
satisfiesRequirement	Resource or Capability	Requirement	Links implementation elements to user or system requirements
hasState	Resource	State	Associates a resource with its current, desired, failed, or previous state
hasAction	Resource	Action	Associates a resource with a lifecycle action
hasConstraint	Resource or Action	Constraint	Attaches validation, policy, or provider constraints
hasGuard	Action	Constraint	Specifies safety conditions required before executing an action
hasCompensatingAction	Action	Action	Defines the compensation or rollback action for a planned action
mappedTo	Resource or Action	Mapping	Connects semantic graph elements to backend-specific constructs

Table 5

Representative implemented semantic completion patterns

Pattern	Inferred semantic effect	Implementation mechanism
ECS service completion	If an ECS service is requested, infer the need for an ECS cluster, task definition, subnet placement, security group, and IAM execution role	SPARQL CONSTRUCT + SHACL validation
Public web endpoint	If the intent requires public HTTP/HTTPS ingress, infer an Application Load Balancer, listener, target group, health check, and appropriate security group rules	SPARQL CONSTRUCT + action/policy rule engine
Private database	If the intent requires a private database, infer RDS instance, DB subnet group, private subnet placement, restricted ingress from the application security group, and backup policy	SPARQL CONSTRUCT + policy validation
Container image provenance	If a task definition references an application container image, infer an ECR repository or validated external registry mapping	SPARQL CONSTRUCT + provider-schema validation
Rollback safeguard	If an action deletes or replaces a stateful resource, infer a snapshot/backup guard and, if needed, human approval	Rule-based reasoning + SHACL validation

Table 6

Representative implemented validation rules

Validation rule	Purpose
ECS service shape	Every ECS service must reference an ECS cluster, task definition, subnet/network configuration, security group, and desired container port mapping
RDS private placement	An RDS instance must not be placed in a public subnet unless explicitly approved by a policy exception
Security group ingress	Database security groups must not allow ingress from 0.0.0.0/0. They should normally allow ingress only from the application security group
IAM least privilege	Task execution roles should contain only permissions required for image pulling, log publishing, secrets access, and declared resource access
Destructive operation guard	Delete or replace actions for stateful resources require snapshot, backup, versioning, or human approval

Formal model of the semantic deployment pipeline. The following formal model specifies the main stages of the SIaC semantic-control pipeline shown in Fig. 1.

Pipeline inputs and semantic graph representation. Let U denote the space of natural-language deployment intents, P denote the provider schema (what resources and parameters are allowed), G_c denote the current semantic infrastructure graph, K denote the domain ontology. Let R denote the set of reasoning rules, S denote the SHACL shape set, C denote the set of semantic and policy constraints, B denote the set of supported IaC backend adapters, $b \in B$ the selected backend. These inputs define the deployment request and the graph model used throughout the pipeline. A deployment request is represented as

$$x = \left(u, P, G_c, K, R, S, C, b\right),\ u \in U,\ b \in B. \qquad (1)$$

A semantic infrastructure graph is modeled as

$$G = \left(V_G, E_G, \tau, \mu\right), \qquad (2)$$

where V_G – the set of nodes representing resources, requirements, capabilities, states, actions, constraints, or mappings; E_G – the set of typed semantic relations between nodes; τ – a typing function assigning ontology classes to nodes; μ – a property/configuration function assigning attributes to nodes. The functions are defined as

$$\tau : V_G \to \mathcal{C}_K,\ \mu : V_G \to 2^{\mathcal{A} \times \mathcal{V}}, \qquad (3)$$

where C_K denotes ontology classes; A denotes configuration attributes, and V denotes attribute values.

Intent grounding, resource grounding, and semantic completion. The first step separates the user's intent from the provider-specific resources that might implement it. The candidate graph generated by the LLM is treated as a preliminary representation that must be completed by ontology reasoning before validation. Therefore, the request is first grounded into a provider-independent intent graph

$$I = \text{GroundIntent}\left(u\right). \qquad (4)$$

This graph illustrates the deployment requirements independently of any particular cloud provider. For example, consider a request for a secure public web application with a private database. The resulting graph contains a set of capabilities and constraints: public HTTP access, a containerised runtime, an isolated database, restricted access to that database, logging, recoverability, and controlled updates. It is important to make this distinction because the same intent can be achieved using different sets of resources, depending on the provider, the backend, and the policy in place.

Second, the intent graph is grounded into a candidate desired infrastructure graph

$$G_d = \text{GroundResources}\left(I, K, P\right). \qquad (5)$$

Within the chosen scope, the objective is to create a containerized web application on AWS ECS/Fargate. Therefore, the candidate graph may include the core ECS/Fargate stack with optional data-layer resources. Although the LLM has mapped intent-level requirements onto infrastructure-level resources, the graph may still contain structural and semantic defects.

Third, ontology reasoning is used to complete the candidate graph and infer missing dependencies

$$G_d^{\star} = \text{Closure}\left(G_d, K, P, R\right), \qquad (6)$$

where $G_d^{\star}$ is the enriched desired graph before validation.

An ECS service, for example, implies an ECS cluster, a task definition, a network configuration, a security group, and an IAM execution role. An RDS instance implies a DB subnet group, placement in a private network, restrictions on its security group, and backup-related safeguards. Thus, instead of treating the LLM's output as final IaC, the system completes it using explicit semantic rules.

Closed-world validation. Once the reasoning process is complete, the graph is checked for missing or invalid elements. In cloud deployment, required resources and properties cannot be left implicit, so any omissions are treated as errors. This step determines whether the graph can proceed to the planning stage or whether it requires repair or human review, using structural, policy and provider-specific checks

$$\begin{aligned} Val = {} & \text{Validate}_{\text{SHACL}}\left(G_d^{\star}, S\right) \cup \\ & \cup \text{Validate}_{\text{policy}}\left(G_d^{\star}, C\right) \cup \text{Validate}_{\text{provider}}\left(G_d^{\star}, P\right), \end{aligned} \qquad (7)$$

where Val is the set of detected violations. If no violations are detected, the validated desired graph is defined as

$$Val = \varnothing \Rightarrow G_d^{+} := G_d^{\star}. \qquad (8)$$

If violations are detected, the method stops before planning and returns a repair or review request

$$Val \neq \varnothing \Rightarrow \text{RepairOrReview}\left(Val\right). \qquad (9)$$

Current-versus-desired alignment and typed state-transition delta. Before the state transition is computed, the validated desired graph is canonicalized and aligned, by identity, with the current semantic infrastructure state. This prevents the planner from treating existing resources as new ones. It also provides the basis for computing a typed transition delta

$$\text{Align}\left(G_c, G_d^{+}\right) \to \left(\hat{G}_c, \hat{G}_d^{+}\right). \qquad (10)$$

This process can be described as follows:

The resource set represented by a graph is defined as

$$\text{Res}\left(G\right) = \left\{v \in V_G \mid \tau\left(v\right) \sqsubseteq \text{Resource}\right\}. \qquad (11)$$

Thus, the current and desired resource sets are

$$\mathcal{R}_c = \text{Res}\left(\hat{G}_c\right),\ \mathcal{R}_d = \text{Res}\left(\hat{G}_d^{+}\right). \qquad (12)$$

Then the typed state-transition delta is partitioned into

$$\Delta = \left\langle \Delta_{create}, \Delta_{update}, \Delta_{replace}, \Delta_{delete}, \Delta_{noop} \right\rangle. \qquad (13)$$

The corresponding resource subsets are:

$$\Delta_{create} = \mathcal{R}_d \setminus \mathcal{R}_c,\ \Delta_{delete} = \mathcal{R}_c \setminus \mathcal{R}_d,\ \mathcal{R}_{common} = \mathcal{R}_c \cap \mathcal{R}_d;$$

$$\Delta_{update} = \left\{r \in \mathcal{R}_{common} \mid \text{Changed}\left(r\right) \wedge \text{UpdateInPlace}\left(r\right)\right\};$$

$$\Delta_{replace} = \left\{r \in \mathcal{R}_{common} \mid \text{Changed}\left(r\right) \wedge \neg\text{UpdateInPlace}\left(r\right)\right\};$$

$$\Delta_{noop} = \left\{r \in \mathcal{R}_{common} \mid \neg\text{Changed}\left(r\right)\right\}.$$

The typed delta treats different lifecycle actions as distinct entities rather than as one generic change set. This preserves their execution semantics, which the planner requires to ensure deployment is both safe and correct.

Action dependency graph and execution planning. The resource-level transition is then transformed into a model at the level of actions. A resource graph represents dependency relations among infrastructure resources. An action graph defines the order in which operations such as create, delete, and verifyState must be executed to preserve safe execution.

The resource-level transition is transformed into an action dependency graph

$$A=\left(V_A,E_A\right),\tag{14}$$

where V_A contains lifecycle actions and E_A contains ordering constraints.

Let O denote the operation alphabet, i. e., the finite set of primitive operation types supported by the SIaC planner and the selected backend

$$\mathcal{O}=\left\{\text{create},\text{update},\text{delete},\text{snapshot},\ldots,\text{verifyState}\right\}.$$

Then action nodes are represented as

$$V_A\subseteq\mathcal{O}\times\left(\mathcal{R}_c\cup\mathcal{R}_d\right),\tag{15}$$

and action-ordering constraints are represented as

$$\left(a_i,a_j\right)\in E_A\Rightarrow a_i\prec a_j,\tag{16}$$

where $a_i\prec a_j$ means that action a_i must be executed before action a_j.

Let define A_{create}, A_{update}, A_{delete} as the subsets of action nodes, which primitive operation O is respectively *create*, *update*, and *delete*. A_{aux} include state-level or control actions such as snapshotting, refreshing the provider state, and verifying the resulting infrastructure state, etc.

Then the forward and destruction subplans are defined as:

$$\pi_{forward}=\text{TopoSort}\left(A_{create}\cup A_{update}\cup A_{aux}\right),$$
$$\pi_{destroy}=\text{ReverseTopoSort}\left(A_{delete}\right).\tag{17}$$

The full dependency-aware execution plan is obtained by merging forward, replacement, and destruction subplans under ordering constraints

$$\pi=\text{Merge}_{\prec}\left(\pi_{forward},\pi_{destroy}\right).\tag{18}$$

Rollback and compensation planning. Each execution plan includes a compensation model that defines how actions can be safely reversed or mitigated. Risky changes are planned with recovery safeguards in advance, which means that rollback is handled as part of the deployment process rather than being delegated to post hoc operational handling.

The rollback/compensation plan associated with π is defined as follows

$$\rho=\left\{\left(a_i,\text{rev}\left(a_i\right),\text{guard}\left(a_i\right),\text{comp}\left(a_i\right)\right)|\,a_i\in\pi\right\}.\tag{19}$$

Here rev(a_i) denotes the reversibility class, guard(a_i) denotes required safety conditions, and comp(a_i) denotes the compensating action. The reversibility class is given by

$$\text{rev}\left(a_i\right)\in\left\{\text{fullyReversible},\text{partiallyReversible},\text{irreversible}\right\}.\tag{20}$$

An action is executable only if all its safety guards are satisfied:

$$\text{Executable}\left(a_i\right)\Leftrightarrow\forall g_j\in\text{guard}\left(a_i\right),\,g_j=\text{true}.\tag{21}$$

Backend generation and provider-state reconciliation. Finally, the validated execution and compensation plans are loVred into a backend-specific artifact. After execution, the system receives the observed provider state. It is then compared with the expected semantic graph to detect residual drift or incomplete execution.

The validated execution and compensation plans are translated into a backend-specific artifact

$$\text{BackendAdapter}_b\left(\pi,\rho\right)\rightarrow\text{artifact}_b.\tag{22}$$

After execution, the observed provider state is imported and aligned with the expected desired graph:

$$G_c'=\text{ImportState}\left(\mathcal{P},B_s\right),\tag{23}$$

$$\text{Align}\left(G_c',G_d^+\right)\rightarrow\left(\hat{G}_c',\hat{G}_d^+\right),\tag{24}$$

where $\mathcal{P}$ denotes the cloud provider runtime/API, and B_s the state backend used to import and persist infrastructure state.

The residual post-execution delta is then computed as

$$\Delta_{post}=\text{TypedDiff}\left(\hat{G}_c',\hat{G}_d^+\right).\tag{25}$$

If the post-execution delta is empty, the semantic state is updated; otherwise, residual drift or incomplete execution is reported:

$$\Delta_{post}=\varnothing\Rightarrow G_c\leftarrow G_c',\ \Delta_{post}\neq\varnothing\Rightarrow\text{ReportDrift}\left(\Delta_{post}\right).\tag{26}$$

The pipeline therefore establishes feedback between planning and the observed cloud state.

Optional plan ranking. When more than one valid plan exists, SIaC can rank them with a single scalar objective. The objective function is used only for optional plan ranking, while the evaluation can report each metric separately.

Finally, alternative plans may be ranked by a scalar objective function

$$J\left(\pi\right)=\lambda_1D\left(\pi\right)+\lambda_2I\left(\pi\right)+$$
$$+\lambda_3\text{Cmpl}\left(\pi\right)-\lambda_4\text{Risk}\left(\pi\right)-\lambda_5T\left(\pi\right)-\lambda_6M\left(\pi\right).\tag{27}$$

Higher values of $J(\pi)$ indicate better deployability $D(\pi)$, intent alignment $I(\pi)$, and compliance Cmpl(π), as well as lower risk Risk(π), planning time $T(\pi)$, and manual correction effort $M(\pi)$. The coefficients λ_i allow the experimenter to express different evaluation priorities. The objective function is used only as an optional ranking mechanism.

Fig. 2 shows a simplified example of the user's natural-language intent, the corresponding candidate graph, its semantic completion, and the validation results. The corresponding scenario is available in repository [16] under the identifier S01.

In Fig. 2, edge labels show relation names without namespace prefixes.

The distinctive feature of the obtained method compared to other LLM-based approaches is that the LLM output is not used as final IaC. Instead, it is transformed into a preliminary semantic representation, and this draft is further completed by ontology reasoning and then checked by closed-world validation. This separates intent interpretation from backend artifact generation and makes the transformation path auditable.

Comparison with related work. Approaches that address a similar problem can be categorised into traditional planning using Terraform/OpenTofu [1, 2], LLM-based or LLM + RAG based IaC generation, and multi-agent IaC generation approaches [10, 11, 17]. Agentic RAG research often uses a knowledge graph as a source of constraints for coherent LLM generation [18]. This supports the general idea of guiding LLM output using structured knowledge. The comparison also includes explicit model-driven and knowledge-based IaC approaches [7, 8]. Unlike traditional planning [1, 2], the proposed model incorporates semantic validation prior to the execution of IaC on the server side. Unlike approaches [10, 11, 17], it does not rely solely on generated text or retrieved examples. Unlike [7, 12], SIaC shifts part of the verification to the desired state graph even before server-side code is generated.

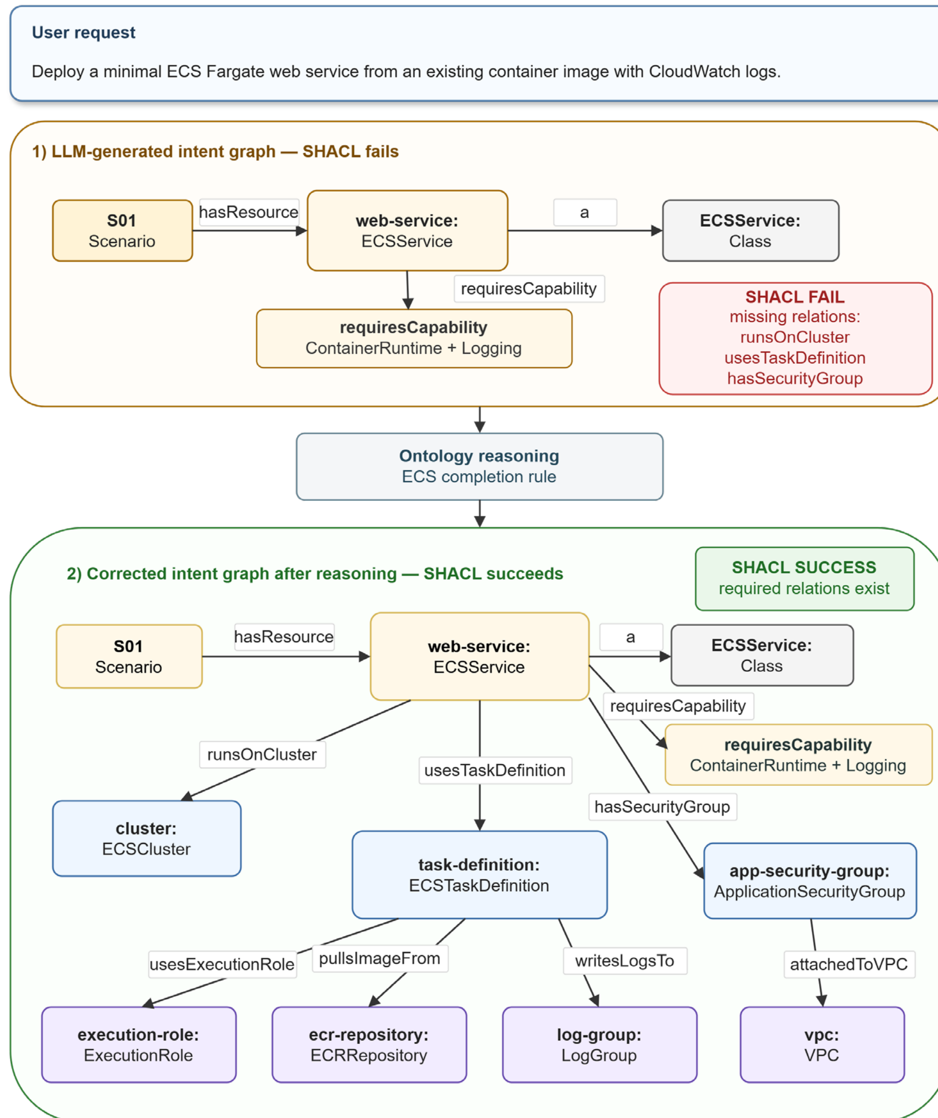

Fig. 2. Ontology-based completion and SHACL validation of an LLM-generated ECS/Fargate intent graph

The limitation of this result is that the formal model was evaluated only within the selected AWS ECS/Fargate-oriented scope and Terraform/OpenTofu backend.

Future work should extend the graph model to additional providers, deployment patterns, and backend adapters.

3.2. Benchmark scenarios and validation protocol

The prototype benchmark includes 60 deployment scenarios. The inputs include: the natural-language requests, the gold semantic requirement sets, the expected resource graphs, and the policy constraints. These materials are available in the supplementary repository [16]. Every scenario belongs to one of the categories in Table 7, and together they cover simple, medium, and difficult deployments.

The validation protocol is described in Table 8.

Under the sandbox protocol, a scenario is counted as successfully deployed if the generated artefact successfully completes every step of the pipeline, which is defined in Section 3.1. If a scenario is only taken to the plan stage, the result is recorded as a plan success rather than a deployment success.

Table 7

Benchmark scenario categories

Scenario category	Count	Purpose in the evaluation
Basic ECS/Fargate web service	8	Tests ordinary deployability and required ECS/Fargate resources
Public endpoint routing	8	Tests ALB, listener, target group, health check, and ingress-rule inference
Private database integration	10	Tests RDS placement, DB subnet groups, restricted ingress, and backup safeguards
Storage and IAM integration	8	Tests S3 access, IAM roles/policies, and least-privilege constraints
Restricted ingress and unsafe requests	8	Tests policy validation, safe rejection, and escalation to human review
Update, replacement, deletion, and drift variants	10	Tests current-vs-desired graph differencing, action ordering, rollback, and reconciliation
Drift and reconciliation	8	Tests drift detection, provider-default normalization, and safe reconciliation of observed vs desired infrastructure state

Table 8

Validation layers in the experiment

Layer	Implementation
Syntax validation	Terraform/OpenTofu validator or backend-specific parser
Plan validation	Terraform/OpenTofu plan generation without application, if possible
Semantic validation	SHACL, OWL/RDFS consistency, SPARQL ASK/CONSTRUCT checks
Policy validation	OPA, CloudFormation Guard, Sentinel-equivalent checks, or custom policy evaluator
Deployment validation	Sandbox is applied in the LocalStack-based AWS emulation environment; endpoint, service-state, and connectivity checks if supported by the emulated services
State validation	Comparison of provider-observed state, backend state, and ontological state graph

The distinctive feature of the benchmark is that it evaluates both syntactic and semantic validity. *Compared with general IaC* generation benchmarks, which focus mainly on code generation quality, syntactic correctness, or deployability, this benchmark assesses whether the generated infrastructure preserves the user's intent and satisfies cloud-level dependencies and constraints.

The proposed benchmark is limited by its size, provider scope and controlled emulation environment. Its metrics depend on the correctness and completeness of the gold requirement sets, gold dependency sets, policy labels and manual-edit counting rules.

Future work should expand the benchmark with more providers and deployment architectures, adversarial and ambiguous requests, etc.

3.3. Experimental comparison with baseline methods

This research compares SIaC with baselines listed in Table 9.

Table 9

Baseline methods used in the experiment

ID	Baseline	Description
B1	Direct LLM-to-IaC	The user prompt is sent directly to an LLM, which generates IaC without a semantic intermediate layer
B2	LLM + RAG-to-IaC	The LLM receives provider documentation, examples, and best-practice snippets before generating IaC
B3	Ontology-no-reasoning	The LLM produces an ontology graph, but does not apply ontology closure, dependency inference, or semantic repair
B4	Full SIaC	The complete pipeline. It includes LLM candidate graph, reasoning closure, SHACL/policy validation, graph diff, plan generation, backend emission
B5	Human-written Terraform/OpenTofu	Reference implementation written by an expert. It is used as an upper-bound correctness and maintainability reference

The main comparisons are conducted between Full SIaC and other baselines. These comparisons show whether the semantic layer provides any measurable benefits beyond improved prompting and retrieval.

The baseline Ontology-no-reasoning has been selected to determine whether an ontological representation alone is sufficient, or whether reasoning, inference of dependencies, and semantic refinement are, in fact, necessary.

Implementation details. The configuration was designed to ensure reproducibility of the comparison among baseline methods and scenarios. Each LLM-based method uses the same GPT-5.5 endpoint, set to a temperature of 0.1, a top-p of 1.0, and an output cap of 16,000 tokens, with up to five retries permitted. Each scenario was run three times independently per method. All the input materials for the experiments are available from [16].

For the base LLM + RAG model, the static corpus was compiled from downloaded documentation on the Terraform provider for AWS, edited provider-specific summaries, and code snippets from Terraform/OpenTofu test cases. These documents cover the ECS/Fargate test scenarios, ALB, RDS, S3/IAM, updates, drifts, and unsafe requests. Searching was performed using BM25 on keywords within Markdown section snippets, with the top five results used as context. To prevent information leakage, reference results, manually corrected plans, and evaluation answers were excluded from the corpus.

The generated IaC was emitted as HCL-style Terraform/OpenTofu configuration and run with the OpenTofu command line interface (CLI), the AWS provider, and pinned provider versions. The experiments ran inside Dockerized LocalStack-based AWS emulation, each in its own workspace with a local state backend and a clean state reset before every scenario. Therefore, the deployment, plan/apply, rollback and reconciliation results should be considered as evidence from a controlled sandbox, rather than from production AWS.

The comparison results are presented in the following tables.

Table 10 demonstrates the main quantitative results across the baselines and the human-written reference, using the metrics from Section 2.2. The arrows show whether higher or lower values are better. Full SIaC is the main automated method here, measured against the LLM-based baselines.

The human-written baseline was designed to serve as a reliable, deployable Terraform/OpenTofu reference for the valid scenarios, not as a semantic safety controller. Therefore, its lower rollback correctness and safe rejection rates should not be interpreted as evidence of lower Terraform quality; they reflect the absence of explicit rejection or compensation mechanisms.

Table 10

Overall performance by baseline

Method	DSR↑%	FAS↑%	SIC↑%	DC↑%	IDR↓%	PCR↑%	RC↑%	SRR↑%	MCE↓%	OH↓
Direct LLM-to-IaC	43.6	15.4	61.9	51.4	16.9	56.1	15.5	4.2	6.3	**2 s**
LLM + RAG-to-IaC	59.0	26.3	69.0	61.3	11.5	64.6	19.1	29.2	4.9	4 s
Ontology-no-reasoning	57.0	25.0	71.7	59.6	9.5	68.4	29.8	41.7	4.0	5 s
Full SIaC	*74.4*	*48.1*	*87.6*	*87.6*	*3.4*	*87.0*	**76.2**	**70.8**	**1.6**	11 s
Human-written Terraform/OpenTofu	**78.2**	**61.5**	**90.9**	**87.7**	**3.3**	**88.5**	56.0	54.2	**1.6**	3 s

ISSN-L 2664-9969; E-ISSN 2706-5448

In all result tables, bold marks the best value in each column or row. *Italic underlined* values mark the best automated result when the human-written reference provides the overall best result.

Under the LocalStack-based sandbox protocol, Full SIaC achieved the best performance among automated methods across the main quality and safety metrics in this benchmark. Compared to Direct LLM-to-IaC and LLM + RAG-to-IaC, it achieved higher values for deployment success, semantic intent coverage, dependency completeness, policy compliance, rollback correctness, and safe rejection. It also reduced invalid dependencies and manual correction effort at the same time. The human-written Terraform/OpenTofu baseline is still the strongest reference on several correctness-oriented metrics. This result is expected because it was handwritten by an expert. Nevertheless, Full SIaC approaches it in deployment and semantic correctness and outperforms it on safety-oriented metrics, such as rollback correctness and safe rejection. This improvement is associated with increased planning time spent on reasoning, validation, graph differencing, and action planning.

Semantic intent coverage and dependency correctness. Table 11 separates semantic correctness from deployment success and shows how missing or incorrect dependencies affect the results.

Among the automated methods, Full SIaC shows the highest semantic intent coverage and dependency completeness, as well as the lowest invalid dependency rate. Compared to Direct LLM-to-IaC and LLM + RAG-to-IaC, Full SIaC results in fewer missing requirements and dependencies and has a lower rate of invalid inferences. This suggests that deployment becomes more reliable, not only because the backend artefact is more complete and semantically consistent, but also because the desired infrastructure graph is completed and verified before any backend code is generated. The increase in dependency completeness aligns well with the purpose of ontology reasoning: to restore the infrastructure relations that direct LLM generation often omits.

Error category analysis. Failures could be assigned more than one label, so a single scenario-baseline run may receive multiple labels simultaneously. Each baseline was evaluated on 60 scenarios with three independent runs each, yielding 180 scenario-baseline runs, and the percentages in Table 12 are taken over those 180. Because the labels overlap, the category percentages do not sum to 100%. A missing-dependency label indicates at least one expected resource or relation was omitted; it does not necessarily correspond to a plan/apply failure.

Full SIaC reduces error rates across all reported categories compared with Direct LLM-to-IaC and LLM + RAG-to-IaC. The largest reductions are in rollback/compensation failures, unsafe requests that were not rejected, plan/apply failures, and invalid inferred dependencies. These categories correspond to the pipeline stages intended to provide policy validation, action planning, compensation modeling, and semantic graph validation. However, missing dependencies remain the most common error category. This also holds for Full SIaC. This indicates gaps in the ontology and rules, as well as provider-specific edge cases and scenarios that the current rule set does not yet fully capture. Thus, the error analysis supports the reliability claim and identifies dependency completion and rule coverage as priorities for further work.

Statistical comparison. Table 13 provides the most relevant paired statistical comparisons between Full SIaC and the baselines. Improvements in deployment success, semantic intent coverage, dependency completeness, policy compliance, and manual correction effort are all statistically significant. The comparison with LLM + RAG shows that retrieval alone does not explain the improvement. The comparison with the Ontology-no-reasoning baseline shows that ontology reasoning improves dependency completeness. The only metric where Full SIaC performs worse is planning overhead. Full SIaC requires additional time for reasoning, validation, graph differencing, and action planning.

Computational overhead. Fig. 3 shows the computational overhead broken down by stage, thereby identifying the stages that contribute most to SIaC planning time before backend generation. Each segment is labeled with the planning time for that stage in seconds, and the total is shown at the end of each bar.

Table 11

Semantic intent coverage and dependency correctness

Method	Mean SIC ↑	Mean DC ↑	Mean IDR ↓	Missing requirements	Missing dependencies	Invalid inferred dependencies
Direct LLM-to-IaC	61.9 ± 15.4%	51.4 ± 13.0%	16.9 ± 11.9%	1101	1636	330
LLM + RAG-to-IaC	69.0 ± 14.3%	61.3 ± 13.1%	11.5 ± 9.8%	849	1298	251
Ontology-no-reasoning	71.7 ± 12.8%	59.6 ± 13.6%	9.5 ± 10.2%	808	1371	199
Full SIaC	**87.6 ± 9.6%**	**87.6 ± 9.0%**	**3.4 ± 4.4%**	356	459	105

Table 12

Error categories by method

Error category	Direct LLM	LLM + RAG	Ontology-no-reasoning	Full SIaC	Primary detection stage
Missing dependency	180 (100.0%)	178 (98.9%)	177 (98.3%)	**144 (80.0%)**	Reasoning/SHACL
Invalid inferred dependency	154 (85.6%)	138 (76.7%)	118 (65.6%)	**80 (44.4%)**	SHACL/action graph
Policy violation	173 (96.1%)	172 (95.6%)	167 (92.8%)	**133 (73.9%)**	Policy validation
Plan/apply failure	88 (48.9%)	64 (35.6%)	67 (37.2%)	**40 (22.2%)**	Backend plan/apply
Failed rollback/compensation	71 (39.4%)	68 (37.8%)	59 (32.8%)	**20 (11.1%)**	Compensation model
Unsafe request not rejected	23 (12.8%)	17 (9.4%)	14 (7.8%)	**7 (3.9%)**	Human review/policy

Table 13

Statistical comparison

Metric	Comparison	Difference	95% *CI*	*p*-value
DSR	Full SIaC vs Direct LLM	+30.8 pp	[20.5, 41.0] pp	<0.001
DSR	Full SIaC vs LLM + RAG	+15.4 pp	[5.8, 25.0] pp	0.004
FAS	Full SIaC vs Direct LLM	+32.7 pp	[23.1, 41.7] pp	<0.001
SIC	Full SIaC vs Direct LLM	+25.8 pp	[23.1, 28.1] pp	<0.001
DC	Full SIaC vs ontology without reasoning	+28.0 pp	[25.9, 30.2] pp	<0.001
PCR	Full SIaC vs LLM + RAG	+22.4 pp	[19.7, 25.0] pp	<0.001
MCE	Full SIaC vs Direct LLM	–4.74	[–4.91, –4.58]	<0.001
OH	Full SIaC vs Direct LLM	8.91 s	[8.69, 9.15] s	<0.001

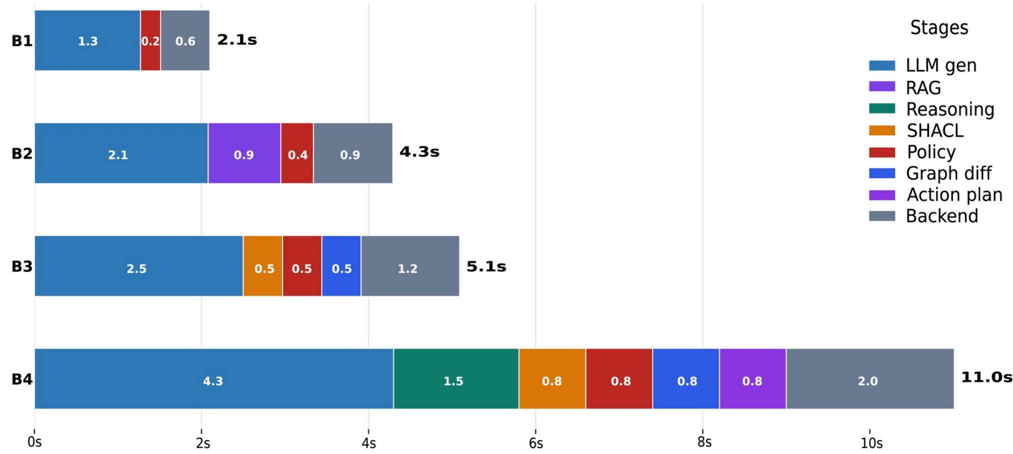

Fig. 3. Computational overhead decomposition by pipeline stage

Full SIaC has the highest planning overhead precisely because it executes the full semantic-control pipeline, which includes reasoning, validation, graph differencing, action planning, and backend generation.

The distinctive feature of the comparison is that the results confirm that the proposed method improves the representation of the infrastructure prior to backend code generation, rather than simply creating Terraform artefacts with an improved syntactic structure.

Compared to the direct conversion of an LLM to IaC, the Full SIaC approach demonstrates the benefits of adding a layer of semantic control. Compared to the LLM+RAG-to-IaC approach, it shows that additional documentation context is insufficient without logical inference, validation and planning. *Compared to Ontology-no-reasoning*, it shows that an ontological representation alone is insufficient. *Compared to human-written Terraform/OpenTofu*, Full SIaC approaches expert-level correctness on several metrics and outperforms it on safety-oriented metrics such as rollback correctness and safe deviation.

The main limitation of this comparison is that the evaluation was conducted in a controlled AWS-emulated environment and within a limited scenario set.

Future work could evaluate the comparison in a real cloud environment, optimize planning overhead, and include maintainability, cost, and operational-risk metrics.

3.4. Ablation analysis

The ablation research in Table 14 shows the effect on a key metric after removing the component expected to contribute to it. The **bold** values mark the worst degradation among the ablation variants for each metric. For a metric with an upward arrow, this is the lowest value, and for one with a downward arrow, it is the highest. The Full SIaC row is the reference setup, not an ablated variant.

Table 14

Ablation research of SIaC components

Variant	DSR↑%	SIC↑%	DC↑%	IDR↓%	PCR↑%	RC↑%	MCE↓
Full SIaC	74.4	87.6	87.6	3.4	87.0	76.2	1.6
No ontology reasoning	66.7	80.5	**61.0**	**9.3**	82.3	56.0	3.7
No SHACL validation	62.2	**79.4**	79.3	6.7	73.1	60.7	3.5
No policy checks	60.9	82.8	79.6	5.9	**55.4**	61.9	3.6
No graph diff	**45.5**	82.1	77.1	6.3	80.2	50.0	**4.6**
No action planning	51.3	84.0	76.1	5.7	81.6	46.4	4.5
No rollback model	62.2	85.0	79.1	5.6	80.1	**17.9**	4.5
No human review/escalation	65.4	86.0	80.7	5.4	77.9	52.4	3.6
No reconciliation	61.5	85.4	78.3	5.3	81.9	66.7	3.6

The greatest reduction in performance is observed when the graph comparison module is removed: the DSR metric drops from 74.4% to 45.5%. This suggests that comparing the current and desired states of the infrastructure and generating a typed set of changes play a key role in the success of the deployment. Removing the action planning module also reduces success rates, as dependencies must be transformed into the correct execution order.

Other removals primarily weaken the quality guarantees for which they are responsible. Without semantic reasoning, validation, policy checking or compensation model, the system exhibits reduced correctness, compliance or operational safety.

An important implication of this result is that it explains why the initial hypotheses were only partially confirmed. Semantic control reduces, but does not entirely eliminate, missing dependencies, policy violations, and failures in the operation of providers and the runtime environment.

Compared with the other evaluation results, the results of ablation tests show that none of the individual mechanisms of SIaC is sufficient on its own. SIaC enhances reliability through a combination of mechanisms such as semantic graph completion, structural and policy checks, state comparison, action ordering, and rollback.

A limitation of this ablation analysis is that only isolated instances of component removal were examined, whereas real-world interactions between components can be far more complex. These interactions should be examined in future research.

3.5. Discussion

The results confirm the main premise: a semantic control layer, based on an ontology and situated between the deployment of natural language intent and the server-side generation of IaC, can enhance the reliability of cloud system deployments using LLMs. Full SIaC outperformed Direct LLM-to-IaC and LLM + RAG-to-IaC not only in terms of deployment success but also across other metrics. This indicates that the method improves the desired representation of the infrastructure prior to the generation of executable IaC artefacts, rather than simply producing syntactically higher-quality Terraform/OpenTofu code. This supports H1 stated in the Introduction.

However, the results should be interpreted with important qualifications. The baseline Ontology-no-reasoning demonstrates that an ontological representation alone is insufficient.

The increase in reliability is achieved through the collaborative work of all pipeline stages. Therefore, SIaC is better described as an integrated semantic management pipeline rather than as a simple ontological wrapper around an LLM. This supports H2 stated in the Introduction, but with the qualification mentioned above.

A comparison with the human-written Terraform reference code also clarifies the scale of the results. The full version of SIaC was the strongest automated method in the test set, but it did not outperform the expert-written baseline on all correctness-oriented metrics. At the same time, SIaC performed better on safety-oriented metrics, as these mechanisms are explicitly represented in the semantic control pipeline. Thus, SIaC reduces the reliability gap between infrastructure code generated by an LLM and code written by experts, but does not eliminate it entirely.

These findings are consistent with recent research on IaC generation, which shows that syntactic correctness is insufficient for reliable infrastructure generation [3, 4]. Compared to post-generation semantic verification approaches, such as Zodiac [12], SIaC shifts part of the verification procedure to an earlier stage where missing dependencies, unsafe placements, policy violations, and incomplete lifecycle logic can be detected before backend IaC generation. This supports H3 stated in the Introduction.

The ablation research indicates why a full pipeline is necessary.

The main trade-off concerns the planning overhead (Fig. 2). This overhead is acceptable when increased reliability, security, auditability, and reduced manual correction costs are more important than minimal response times.

Practical significance. The results obtained can be applied in DevOps workflows and cloud environment automation as semantic pre-execution quality control. This approach is particularly relevant for internal developer platforms, CI/CD pipelines, DevOps assistants, and similar tools, where requests for cloud deployment may be incomplete or formulated in natural language.

Threats to validity and limitations of research. When interpreting the experimental results, a number of limitations relating to reliability should be taken into account.

First, all deployment experiments were conducted in an emulated AWS environment based on LocalStack using Docker, therefore, the reported deployment success data reflect behaviour in a controlled test environment. Although LocalStack supports reproducible testing, it cannot fully replicate all aspects of real-world AWS behavior. These include asynchronous service provisioning, eventual consistency, and quota enforcement, which occur only in real AWS accounts.

Second, the current ontology and rule base cover a deployment scope focused solely on AWS ECS/Fargate.

Third, the semantic management pipeline requires additional planning time, making it less suitable for latency-sensitive workflows in which infrastructure decisions must be made within strict latency constraints. Examples include real-time autoscaling, emergency failover, incident-response automation, and interactive deployment assistants.

Finally, maintaining the ontology requires continuous knowledge engineering, particularly considering developments in cloud providers' APIs and best practices.

Prospects for further research. Future work should expand the scope of the ontology, add more AWS and other provider services, provide support for additional deployment targets and backend adapters, and evaluate the method in real-world cloud environments. Further research should also improve rollback and reconciliation for stateful resources, reduce planning overhead through incremental or adaptive inference, and expand the test suite with ambiguous queries, production-like traces, and additional operational metrics.

4. Conclusions

1. A graph-based semantic control method has been formalised, and an AWS ECS/Fargate-oriented prototype has been implemented based on it. The service ontology contains 90 classes, 103 object and data properties, 28 validation\inference rules and policy constraints, 15 Terraform mapping templates, and 35 validation checks. The prototype demonstrated that LLM output can be controlled via a semantic graph prior to IaC generation. From an applied perspective, this provides a more transparent, verifiable, and auditable path from intent to deployment.

2. A benchmark comprising 60 AWS ECS/Fargate-oriented scenarios and a validation protocol were developed. This allows to evaluate not only the syntax validity of the generated IaC, but also intent coverage, dependencies, policy, rollback and safe rejection. From an applied perspective, the benchmark can serve as a basis for comparing future LLM/IaC systems.

3. Based on the developed prototype, a comparison was conducted between the proposed method and a set of selected baselines. According to the comparison results, Full SIaC achieved a DSR of 74.4% compared to 43.6% for Direct LLM, 59.0% for LLM + RAG and 57.0% for Ontology-no-reasoning. These results indicate that SIaC achieved the strongest performance among automated methods in the benchmark and can reduce deployment errors, policy violations and manual corrections, although it does not completely outperform human-written Terraform.

4. The ablation analysis established the contribution of the components. Without graph diff, DSR drops from 74.4% to 45.5%. Without action planning – to 51.3%. Without ontology reasoning, DC drops from 87.6% to 61.0%. Without the rollback model, RC drops from 76.2% to 17.9%. Without policy checks, PCR drops from 87.0% to 55.4%. This shows that the effect is not due to a single ontology but to the entire semantic-control pipeline. From an applied perspective, this finding helps identify which components are critical for the reliable implementation of SIaC.

Acknowledgments

This publication is based upon work from COST Action CA23147 GOBLIN – Global Network on Large-Scale, Cross-domain and Multilingual Open Knowledge Graphs, supported by [19].

Conflict of interest

The authors declare that they have no conflict of interest in relation to this research, whether financial, personal, authorship, or otherwise, that could affect the research and its results presented in this paper.

Financing

The research was performed without financial support.

Data availability

Manuscript-related data is located at [16].

Use of artificial intelligence

The authors confirm that Grammarly version 1.2 was used for grammar editing of the manuscript. OpenAI ChatGPT version 5.5 was used while searching relevant sources for the literature review in Section 1 and to make the data visualisation in Fig. 2. The same model was also used as an experimental system in the LLM-based baselines described in Section 3.3. All results obtained from AI were verified by

the authors. The use of AI did not affect the scientific research results, which were formulated by the authors independently.

Authors' contributions

Ihor Bibichkov: Conceptualization, Software, Investigation, Resources, Data curation, Writing – original draft; ***Olena Shevchenko***: Conceptualization, Methodology, Software, Validation, Formal analysis, Investigation, Resources, Data curation, Writing – review and editing; ***Oleksandr Shevchenko***: Methodology, Formal analysis, Investigation, Supervision; ***Oleksandr Stopin***: Software, Validation, Formal analysis, Investigation.

References

1. Terraform language documentation. *HashiCorp Developer*. Available at: https://developer.hashicorp.com/terraform/language
2. Getting started. *OpenTofu*. Available at: https://opentofu.org/docs/intro/
3. Zhang, T., Pan, S., Zhang, Z., Xing, Z., Sun, X. (2025). Deployability-Centric Infrastructure-as-Code Generation: Fail, Learn, Refine, and Succeed through LLM-Empowered DevOps Simulation. *arXiv:2506.05623.* https://doi.org/10.48550/arXiv.2506.05623
4. Davidson, S., Sun, L., Bhasker, B., Callot, L., Deoras, A. (2025). Multi-IaC-Eval: Benchmarking Cloud Infrastructure as Code Across Multiple Formats. *arXiv:2509.05303*. https://doi.org/10.48550/arXiv.2509.05303
5. Dependency graph. Terraform Internals. HashiCorp *Developer*. Available at: https://developer.hashicorp.com/terraform/internals/graph
6. Shevchenko, A. Yu., Shevchenko, E. L. (2012). How to bring artificial intelligence into the clouds. *Eastern-European Journal of Enterprise Technologies, 3 (12 (51)),* 66–70. Available at: https://journals.uran.ua/eejet/article/view/2472
7. Topology and Orchestration Specification for Cloud Applications (TOSCA) Version 2.0. (2025). *OASIS Standard*. Available at: https://docs.oasis-open.org/tosca/TOSCA/v2.0/TOSCA-v2.0.html
8. Vasileiou, Z., Kumara, I., Meditskos, G., Tokmakov, K., Radolović, D., Cruz, J. G. et al. (2025). A knowledge-based approach for guided development of Infrastructure as Code. *Software and Systems Modeling, 25 (2),* 515–548. https://doi.org/10.1007/s10270-025-01294-1
9. Bibichkov, I., Sokol, V., Shevchenko, O. (2017). Ontological knowledge bases productivity optimization through the use of reasoner combination. *Eastern-European Journal of Enterprise Technologies, 5 (2 (89)),* 49–54. https://doi.org/10.15587/1729-4061.2017.112347
10. Kon, P., Liu, J., Qiu, Y., Fan, W., He, T., Lin, L. et al. (2024). IaC-Eval: A Code Generation Benchmark for Cloud Infrastructure-as-Code Programs. *Advances in Neural Information Processing Systems 37,* 134488–134506. https://doi.org/10.52202/079017-4273
11. Khan, R. N. H., Wasif, D., Cho, J.-H., Butt, A. (2025). Multi-agent code-orchestrated generation for reliable Infrastructure-as-Code. *arXiv:2510.03902v1.* https://doi.org/10.48550/arXiv.2510.03902
12. Qiu, Y., Kon, P. T. J., Beckett, R., Chen, A. (2024). Unearthing Semantic Checks for Cloud Infrastructure-as-Code Programs. *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, 574–589. https://doi.org/10.1145/3694715.3695974
13. Open Policy Agent documentation. *Open Policy Agent*. Available at: https://www.openpolicyagent.org/docs
14. What is AWS CloudFormation Guard? *Amazon Web Services*. Available at: https://docs.aws.amazon.com/cfn-guard/latest/ug/what-is-guard.html
15. Shevchenko, O. L. (2013). Semantic annotations similarity measure to compare processes profiles. *Eastern-European Journal of Enterprise Technologies, 3 (2 (63)),* 48–52. https://doi.org/10.15587/1729-4061.2013.14445
16. SIaC Supplementary Artifacts. *GitHub*. Available at: https://github.com/shevchenko-oleksandr/SIaC
17. Nekrasov, R., Fossati, S., Kumara, I., Tamburri, D. A., van den Heuvel, W.-J. (2026). IaC Generation with LLMs: An Error Taxonomy and A Study on Configuration Knowledge Injection. *ACM Transactions on Software Engineering and Methodology*. https://doi.org/10.1145/3817608
18. Fliahin, V., Turuta, O., Turuta, O. (2025). Approaching LLM alignment using agents with RAG. *Proceedings of the 5th International Workshop of IT-professionals on Artificial Intelligence (ProfIT AI 2025). CEUR Workshop Proceedings, 4164,* 207–215. Available at: https://ceur-ws.org/Vol-4164/short6.pdf
19. *COST (European Cooperation in Science and Technology)*. Available at: https://www.cost.eu

Ihor Bibichkov, *Senior Lecturer, Department of Artificial Intelligence, Kharkiv National University of Radio Electronics, Kharkiv, Ukraine, ORCID: https://orcid.org/0000-0003-1424-6960*

Olena Shevchenko, *Candidate of Technical Sciences, Associate Professor, Department of Software Engineering, Kharkiv National University of Radio Electronics, Kharkiv, Ukraine, ORCID: https://orcid.org/0000-0003-3177-5530*

✉***Oleksandr Shevchenko,*** *Candidate of Technical Sciences, Professor, Department of Artificial Intelligence, Kharkiv National University of Radio Electronics, Kharkiv, Ukraine, e-mail: oleksandr.shevchenko@nure.ua, ORCID: https://orcid.org/0000-0002-0068-4698*

Oleksandr Stopin, *Senior Lecturer, Department of Artificial Intelligence, Kharkiv National University of Radio Electronics, Kharkiv, Ukraine, ORCID: https://orcid.org/0009-0002-3074-3772*

✉*Corresponding author*