ARPaCCino: An Agentic-RAG for Policy as Code Compliance

Francesco Romeo[1,2][0009-0006-3402-3675],
Luigi Arena[1][0009-0008-9844-0229],
Francesco Blefari[1,2][0009-0000-2625-631X],
Francesco Aurelio Pironti[1][0009-0003-3183-2977],
Matteo Lupinacci[1][0009-0000-2356-398X], and
Angelo Furfaro[1][0000-0003-2537-8918]

[1] University of Calabria, 87036, Rende (CS), Italy
`{francesco.romeo, luigi.arena, francesco.blefari, francesco.pironti, matteo.lupinacci, angelo.furfaro}@unical.it`
[2] IMT School for Advanced Studies Lucca, 55100, Lucca (LU), Italy
`{francesco.romeo, francesco.blefari}@imtlucca.it`

Abstract. Policy as Code (PaC) is a paradigm that encodes security and compliance policies into machine-readable formats, enabling automated enforcement in Infrastructure as Code (IaC) environments. However, its adoption is hindered by the complexity of policy languages and the risk of misconfigurations. In this work, we present ARPaCCino, an agentic system that combines Large Language Models (LLMs), Retrieval-Augmented-Generation (RAG), and tool-based validation to automate the generation and verification of PaC rules. Given natural language descriptions of the desired policies, ARPaCCino generates formal `Rego` rules, assesses IaC compliance, and iteratively refines the IaC configurations to ensure conformance. Thanks to its modular agentic architecture and integration with external tools and knowledge bases, ARPaCCino supports policy validation across a wide range of technologies, including niche or emerging IaC frameworks. Experimental evaluation involving a Terraform-based case study demonstrates ARPaCCino's effectiveness in generating syntactically and semantically correct policies, identifying non-compliant infrastructures, and applying corrective modifications, even when using smaller, open-weight LLMs. Our results highlight the potential of agentic RAG architectures to enhance the automation, reliability, and accessibility of PaC workflows.

Keywords: Policy as Code · Agentic AI · Retrieval Augmented Generation · Large Language Models.

1 Introduction

Over the years, software and infrastructure management have become increasingly challenging due to the growing complexity and scale of systems. To address these challenges, developers have embraced DevOps practices, which aim to reduce operational errors, accelerate provisioning, and support continuous updates throughout the development and operations lifecycle. Within this context, Infrastructure as Code (IaC) has emerged as a standard practice. By expressing infrastructure specifications in machine-readable code, IaC enables automated provisioning, configuration, and management. This approach significantly improves automation, scalability, reproducibility, and consistency across the entire service lifecycle.

Despite its benefits, IaC remains prone to misconfigurations and security vulnerabilities when applied without sufficient expertise. To mitigate these risks, comprehensive testing and validation are often necessary before deployment.

Policy as Code (PaC) extends the Infrastructure as Code (IaC) paradigm to the definition of security and compliance policies, expressing them as formal, machine-readable rules that can be automatically validated and enforced during the provisioning process. By integrating policy checks into Continuous Integration and Continuous Deployment (CI/CD) pipelines, PaC helps reduce human error and ensures infrastructure configurations meet security and compliance requirements early in the development lifecycle.

However, the adoption of PaC is often hindered by the steep learning curve of domain-specific policy languages and the difficulty of authoring correct, comprehensive rules—especially in dynamic and complex environments.

Recent advancements in Large Language Models (LLMs) and LLM-based techniques offer a promising solution to the limitations of current IaC and PaC practices, supporting their deeper integration into standard industry workflows. LLMs can translate high-level policy descriptions, written in natural language, into formal, machine-readable rules suitable for automated validation of IaC configurations.

Additionally, AI agent-based workflows can enhance the generation and refinement of IaC and PaC artifacts by iteratively interacting with domain-specific tools and structured knowledge bases. In particular, Retrieval-Augmented Generation (RAG) techniques extend LLM capabilities with contextual, domain-specific knowledge, enabling accurate handling of niche or emerging technologies without the need for extensive retraining.

In this context, we present ARPaCCino, an agentic system that combines a core reasoning LLM with RAG and specialized tools to automate the generation and validation of policies for IaC. Given a natural language description of the desired policies, ARPaCCino generates formal rules in `Rego` – the policy language used by Open Policy Agent (OPA) – then verifies and applies them to assess compliance of the provided IaC specification.

If the validation reveals non-compliance, ARPaCCino can autonomously propose and apply iterative corrections to the IaC configuration until the specified requirements are satisfied. The system's RAG module can be supplied with custom domain-specific knowledge bases, enabling its use across a wide range of technologies, including less common or emerging frameworks, provided that relevant documentation and examples are available.

The main contributions of this work can be summarized as follows:

- We propose a novel approach to Policy as Code generation and Infrastructure as Code validation, leveraging agentic systems that combine LLMs, RAG, and external tool integrations.
- We implement this approach in the ARPaCCino system, which generates formal policy rules from natural language, assesses infrastructure compliance, and iteratively refines configurations until policy conformance is achieved.
- We demonstrate the effectiveness of ARPaCCino through a realistic Terraform-based use case, showing its ability to autonomously retrieve domain knowledge, synthesize and verify policy rules, and validate or revise IaC specifications accordingly.

The remainder of this paper is organized as follows. Section 2 provides background knowledge on Infrastructure as Code, Policy as Code, and AI agents. Section 3 details the ARPaCCino system architecture, including its core LLM engine and the available tools. Section 4 presents a real use case involving Terraform, demonstrating the end-to-end workflow from a natural language policy description to a verified IaC definition. Section 5 examines the results of the experimental evaluation conducted with different scenarios and LLMs. Lastly, Section 6 discusses conclusions and outlines directions for future work.

Through ARPaCCino, our aim is to advance the state of automated Policy as Code by leveraging agentic AI and RAG techniques to reduce developer burden, improve compliance, and support evolving infrastructure ecosystems.

2 Background

The provisioning of services in modern computing environments is a complex task that requires custom architectures tailored to specific use cases. These systems often consist of multiple interconnected components, such as microservices, databases, and networked elements, that increase the overall complexity and hinder maintainability. To manage this growing complexity, developers have adopted DevOps methodologies, aiming to reduce error rates, accelerate service provisioning, and enable continuous software delivery.

2.1 Infrastructure as Code

To support efficient Continuous Integration and Continuous Deployment [1] in such environments, IaC emerged as a foundational DevOps practice [2]. IaC enables programmatic provisioning, configuration, and management by using machine-readable code [3]. This enhancing automation, reproducibility, and consistency across both development and operational phases.

To support IaC practice in DevOps, several languages, platforms and tools have been developed that allow the creation, customization, and orchestration of system components, including microservices, virtual machines, and networking layers. Popular IaC tools include Terraform [4] (declarative, cloud-agnostic), Ansible [5] (configuration-focused), and Pulumi [6] (uses general-purpose languages). These tools enable the infrastructure to be versioned, tested, and deployed as application code.

Despite these advantages, IaC tools are still susceptible to misconfigurations and logic errors, which may lead to performance issues or security vulnerabilities. Over time, several solutions have been developed to test and validate the IaC infrastructure to ensure the correctness of the system before its deployment [7,8].

More recently, LLMs have been applied to IaC workflows to reduce manual effort and enhance reliability. LLMs can translate high-level natural language descriptions into valid infrastructure code, thus accelerating development and mitigating syntactic and semantic mistakes [9,10]. Some approaches also integrate automated validation and correction loops, enabling the detection and resolution of configuration errors prior to deployment [10].

2.2 Policy as Code

PaC extends the IaC paradigm by codifying security, compliance, and operational policies into machine-readable formats. This enables automated policy enforcement throughout the software development lifecycle, ensuring continuous compliance and reducing human error [11]. PaC integrates directly into CI/CD pipelines, facilitating automated validation of infrastructure configurations and application deployments against predefined rules, thereby shifting security left in the DevSecOps pipeline [12].

The de facto standard implementation is *Open Policy Agent (OPA)* [13], an open-source general-purpose policy engine using the `Rego` declarative language to express fine-grained authorization, admission control (e.g., in Kubernetes), and data-filtering rules via API calls or as an integrated library. Complementing this is *HashiCorp Sentinel* [14], tailor-made for the HashiCorp enterprise stack (Terraform Enterprise, Vault, Consul, Nomad), featuring its own policy language, support for logical constructs and imports, and multiple enforcement levels (advisory, soft-mandatory, hard-mandatory). Sentinel enables proactive pre-deployment governance, enforcing policies as a prerequisite to resource provisioning.

Despite its advantages, PaC adoption can face challenges such as the steep learning curve for policy languages and managing policy drift in dynamic environments. However, the advent of AI and LLMs offers significant opportunities to overcome these limitations, enabling AI-assisted policy generation, automated validation, and intelligent support for policy comprehension.

2.3 AI Agent

An *AI agent* is an autonomous software entity capable of reasoning about goals and executing actions to achieve specified objectives [15]. It is typically characterized by its ability to perceive the environment in which it operates, respond to environmental changes, and interact with external systems or other agents.

An AI Agent is also provided with a *memory*, useful to learn from past experiences and maintain context while addressing a task. Fig. 1 illustrates a schema of an AI agent structure.

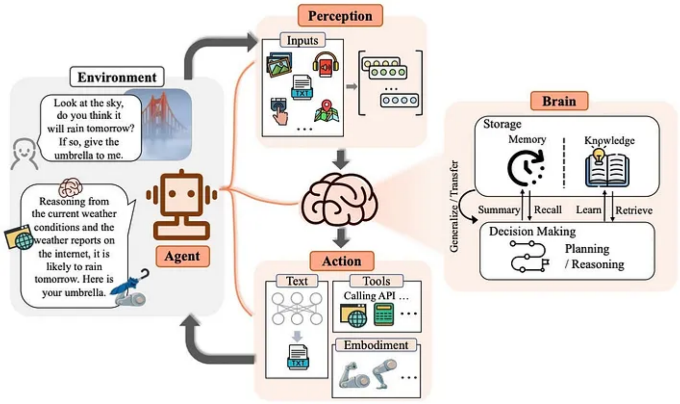

Environment
Look at the sky, do you think it will rain tomorrow? If so, give the umbrella to me.
Reasoning from the current weather conditions and the weather reports on the internet, it is likely to rain tomorrow. Here is your umbrella.
Agent
Perception
Inputs
TXT
...
Brain
Storage
Memory
Knowledge
Generalize / Transfer
Summary
Recall
Learn
Retrieve
Decision Making
Planning / Reasoning
Action
Text
TXT
Tools
Calling API ...
...
Embodiment
...

Fig. 1. AI agent structure [16]

As discussed in [16], the emergence of LLMs marks a significant advancement in the development of intelligent agents. This evolution has led to the rise of *LLM agents*, which use LLMs as the core reasoning engine to perform task decomposition, planning, and decision-making. While maintaining the reactive and interactive characteristics of traditional agents, LLM agents are augmented with the ability to invoke external tools (e.g., calculators, code interpreters, or knowledge bases) to solve domain-specific subtasks. The LLM continuously evaluates whether the task has been completed or if further tool-based refinement is required, enabling flexible and iterative problem-solving.

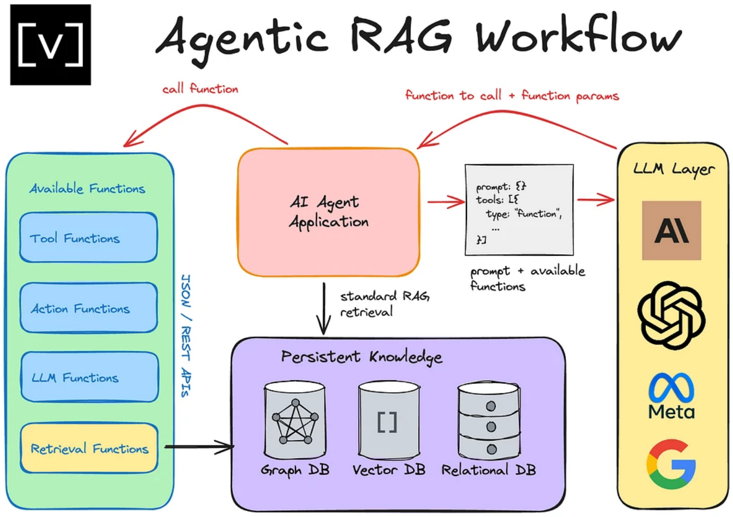
Agentic RAG architectures While off-the-shelf LLMs and agent frameworks built on them offer broad utility, they often struggle with domain-specific or expert-level tasks due to the absence of embedded, up-to-date knowledge. Although this limitation can be addressed through retraining or fine-tuning, such approaches are typically resource-intensive and time-consuming.

A more scalable and cost-effective solution is offered by the RAG paradigm [17]. In RAG systems, the LLM is coupled with two key components: *(i)* a *repository* of domain-specific knowledge (for example, a curated collection of documents) and *(ii)* a *retriever* that locates relevant content from this repository to enrich the model's input context.

This architecture allows LLMs to answer specialized or evolving queries by incorporating external knowledge at inference time.

A typical RAG pipeline operates as follows: an external knowledge corpus is preprocessed into manageable chunks, transformed into vector embeddings, and indexed for fast retrieval. When a user submits a query, the retriever encodes it into a vector, searches the index for the most semantically relevant chunks, and returns them. These retrieved excerpts are then combined with the user query to form an augmented prompt, which is passed to the LLM. The result is a context-aware, knowledge-informed response that extends beyond the model's original training data.

When this retrieval loop is embedded within an agent framework, enabling iterative reasoning, tool use, and multi-step workflows, the resulting architecture is referred to as *Agentic RAG*. Fig. 2 provides an overview of a typical architecture of an *Agentic RAG* system, illustrating the main components and their interactions during the query processing workflow. Modern frameworks such as *LangChain* [18], *LlamaIndex* [19], and *Langroid* [20] support the rapid development of Agentic RAG systems by abstracting core components like document ingestion, embedding management, retrieval orchestration, and LLM-based decision-making.



```
prompt: {}
tools: [{
  type: "function",
    ...
}]
```

Fig. 2. Agentic RAG architecture. © Vectorize.io

AI Agents for IaC and PaC Recent research has shown the growing applicability of AI techniques, in particular LLMs, in the domains of IaC and PaC.

In the context of IaC, LLMs have been successfully applied to automatically generate infrastructure definitions [21,22,23]. However, these models are susceptible to well-known limitations, including *hallucinations*, which may result in code that is syntactically incorrect or semantically invalid. As a result, naive applications of LLMs may introduce critical misconfigurations or deployment issues due to erroneous code in terms of both syntax and semantics.

To face these limitations, more advanced approaches have adopted LLM agents that combine reasoning with external tool integration and RAG. Some examples include the agentic architectures proposed in [24,25,26], which demonstrate the value of iterative, tool-assisted development cycles. These systems partially address the shortcomings of standard LLMs by incorporating validation, self-correction, and reasoning loops.

Similar techniques can be applied to the domain of PaC, where formal policy rules, typically expressed in languages such as `Rego`, can be generated from natural language descriptions. While some preliminary exploration in this direction exists [27], to the best of our knowledge, the system presented in this work is the first to autonomously: (i) translate natural language policy descriptions into formal PaC rules, (ii) assess the compliance of a given IaC configuration with the generated policies, and (iii) iteratively modify the infrastructure definition to ensure full compliance with the specified requirements.

3 ARPaCCino Architecture

ARPaCCino is an Agentic RAG system designed to translate natural language policy descriptions into formal `Rego` rules and validate IaC architectures against those policies. Leveraging the flexibility of the Agentic RAG approach, ARPaCCino is able to autonomously refine the IaC configuration until it satisfies all specified policy constraints. At its core, ARPaCCino consists of a reasoning engine based on an LLM, which orchestrates execution by interpreting requests, generating action plans, and invoking a suite of specialized tools. A high-level overview of the system architecture is shown in Figure 3.

RAG Tool. The RAG Tool provides access to domain-specific knowledge, including official documentation, the `Rego` language definition, and examples for both the Open Policy Agent (OPA) and the supported IaC frameworks. The LLM invokes this tool whenever domain-specific knowledge is required. The retrieved content is used to enhance the prompt, effectively extending the LLM's capabilities. This modularity enables ARPaCCino to support uncommon or emerging technologies, provided that a structured knowledge base is supplied.

Infrastructure Tools. To ensure compatibility with multiple IaC frameworks, ARPaCCino leverages a set of specialized *infrastructure tools*, tailored to each framework. Those tools perform the required pre-processing on the given infrastructure definition before proceeding with the policy validation.

Rule Checker Tool. The LLM engine, after fetching the appropriate knowledge from the RAG tool, generates the `Rego` rules corresponding to the input

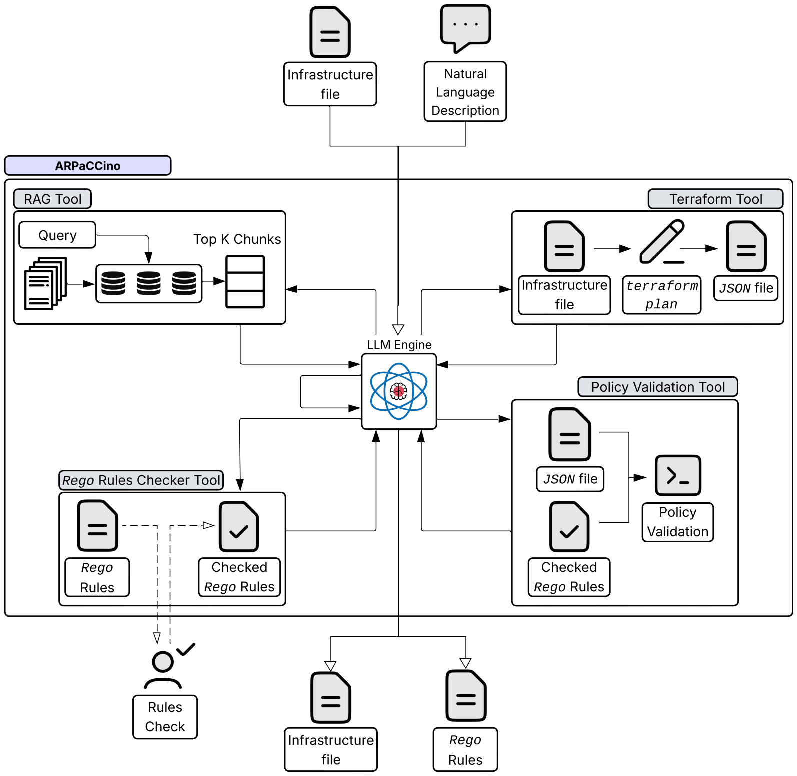


Fig. 3. ARPaCCino Architecture

policy description. The generated rules should then be verified prior to the automatic validation and architecture improvement. While OPA's native `opa check` command ensures syntactic correctness, it does not evaluate the semantic validity or logic of the rules. The Rule Checker Tool addresses this limitation by incorporating feedback from an external domain expert (or oracle), who reviews and either accepts or rejects the generated rules. This step ensures the soundness of the policy prior to enforcement.

Policy Validation Tool. The Policy Validation Tool takes as input the preprocessed infrastructure and the semantically verified `Rego` rules. It performs a deterministic evaluation to determine whether the infrastructure complies with the generated policies. Based on this result, the system decides whether the current IaC specification is ready for deployment or requires further adjustments.

4 Case study

To evaluate the effectiveness of ARPaCCino, we present a real case study based on the widely adopted IaC framework Terraform [4]. In this scenario, the knowledge base available to the RAG tool includes documentation for OPA and for the Terraform provider for ProxMox [28]. The infrastructure is defined using standard Terraform configuration files (`.tf`), and the system invokes the `terraform plan` command to preprocess the infrastructure. This command generates a JSON-formatted execution plan, which serves as the input to the policy validation phase.

4.1 Expected Workflow

Algorithm 1 outlines the expected workflow of ARPaCCino. For illustrative purposes, we assume a simplified scenario in which each sub-task completes in a single tool invocation. This abstraction allows us to highlight the logical sequence of steps without delving into low-level agentic behaviors.

Algorithm 1 Expected ARPaCCino workflow

```
Input: Infrastructure file, Natural language description of the policies
Output: Verified Infrastructure file, Generated Rego rules.
 1: Retrieve OPA Knowledge using RAG Tool
 2: Generate Rego Rules from the natural language description
 3: Verify the Rego Rules using the Checker Tool
 4: if Rules are wrong then
 5:     go to 2
 6: end if
 7: Preprocess the Infrastructure file using the Terraform Tool
 8: Validate the Infrastructure JSON file against the Rego Rules using the Policy Validation Tool
 9: if Infrastructure is not policy-compliant then
10:     Retrieve Terraform Knowledge using RAG Tool
11:     Correct the Infrastructure file
12:     go to 7
13: end if
14: return Verified Infrastructure file, Generated Rego rules.
```

In real cases, the agentic nature architecture of ARPaCCino often requires multiple iterations with the same tool in order to iteratively refine the output and achieve satisfactory results. However, this complexity is abstracted away from the end user. ARPaCCino manages all intermediate decisions and tool invocations internally. This design ensures a seamless experience, allowing users to focus on high-level objectives while the system autonomously orchestrates the underlying reasoning and execution processes.

Terraform File excerpt

```
...
resource "proxmox_virtual_environment_vm" "cloned_vm" {
  ...
  cpu {
    cores = 4
  }
  memory {
    dedicated = 8192
  }
  disk {
    interface    = "scsi0"
    datastore_id = "Storage"
    size         = 150
  }
  network_device {
    bridge = "intVM"
    model  = "virtio"
  }
  tags        = ["PaC"]
  description = "Cloned for PaC test"
}
```

Policy prompt

Allow only virtual machines with 4 cores in Terraform.

Generated `Rego` rule

```
package terraform

deny[msg] if {
    resource := input.planned_values.root_module.resources[_].values
    cpu := resource.cpu[_].cores
    cpu != 4
    msg := "VM must have exactly 4 cores"
}
```

Fig. 4. ARPaCCino running example

4.2 Running Example

A minimal yet illustrative example of the capabilities of ARPaCCino is shown in Figure 4. The environment consists of a single machine equipped with 4 CPU cores and 8 GB of RAM. The user instructs ARPaCCino to allow only machines with exactly 4 cores. ARPaCCino processes this request by generating the corresponding `Rego` policy, then evaluating the compliance of the infrastructure against it. Since the current configuration satisfies the constraint, the system confirms policy compliance without requiring any modifications to the Terraform definition.

5 Experimental Results

The experimental evaluation conducted aimed to assess the effectiveness of ARPaCCino in the generation and validation of Policy as Code, with a focus on its applicability in a Terraform-based IaC scenario. The evaluation considered the following key aspects:

- **Syntactic correctness** of the generated `Rego` policies;
- **Semantic alignment** of the policies with the natural language user instructions;
- **Detection capability** for identifying policy violations within Terraform execution plans;
- **Repair effectiveness** in automatically correcting non-compliant infrastructure definitions.

All experiments were conducted on a ProxMox-based Asus ESC4000A-E12 server with an AMD EPYC 9004 processor, 2x 48GB L40s NVIDIA GPUs, and 196 GB of RAM. The publicly available LLMs were run using Ollama inside a Ubuntu 24.04 virtual machine hosted on the server, provided with 16 cores, 128 GB of RAM, and both available GPUs. Closed-source models were accessed via API-based interactions using their publicly available endpoints.

5.1 Evaluation methodology

We evaluated ARPaCCino on a defined and fixed Terraform infrastructure, with five distinct policy prompts of increasing difficulty. Among these, three prompts required the modification of the provided infrastructure, due to their incompatibility with the original IaC definition.

To assess the effectiveness of each system component, we conducted an ablation study with the following configurations:

- **LLM-only:** the base LLM is used without access to retrieval or external tools;
- **LLM + RAG:** the model is enhanced with retrieval capabilities but lacks access to tool execution;
- ARPaCCino **(full):** the complete agentic system with both RAG and tool invocation capabilities enabled.

Furthermore, we tested different LLMs for each of the discussed scenarios, to understand to what extent the capabilities of the "raw" LLM affect the overall generation performance of the system. We chose to evaluate ARPaCCino with three different models: the open-weight model `Qwen3` in its 30 billion parameters version and the closed `GPT-4o` and `Claude Sonnet 4` models. These models represent the current state-of-the-art and support tool calling, ensuring a fair comparison in all the described scenarios.

5.2 LLM vs RAG vs Agentic RAG

In table 1 are depicted the results of the ablation study. The table shows a comparison across three different approaches, LLM, RAG, and Agentic RAG, applied to three models (`Qwen3:30b`, `GPT-4o`, and `Claude Sonnet 4`). The *Model* column specifies the large language model employed for the specific batch of tests. The *Configuration* column indicates whether the base LLM, RAG, or agentic RAG configuration was used. The *Syntax* column shows how many out of five `Rego` policy generations were syntactically correct, and the *Semantic* column reports how many of the syntactically correct policies were also semantically correct. The last column, *Notes*, provides additional observations regarding each setup, such as errors encountered or behavior noted during the executions. As expected, the agentic approach used in ARPaCCino greatly enhances the system's capability of generating syntactically and semantically correct `Rego` policies.

Model	**Configuration**	**Syntax**	**Semantic**	**Notes**
`Qwen3:30b`	LLM	0/5	—	`Rego` rules generated with syntactical errors
`Qwen3:30b`	RAG	0/5	—	`Rego` rules generated with syntactical errors
`Qwen3:30b`	Agentic RAG	4/5	4/5	Loop during the policy correction for 1 prompt
`GPT-4o`	LLM	0/5	—	`Rego` rules generated with syntactical errors
`GPT-4o`	RAG	0/5	—	`Rego` rules generated with syntactical errors
`GPT-4o`	Agentic RAG	5/5	5/5	1/3 Terraform file modified
`Claude Sonnet 4`	LLM	5/5	5/5	`Rego` rules generated without external knowledge
`Claude Sonnet 4`	RAG	5/5	5/5	`Rego` rules generated
`Claude Sonnet 4`	Agentic RAG	5/5	5/5	`Rego` rules generated and checked

Table 1. Performance summary and comparison

Base LLMs most likely lack knowledge about OPA and `Rego` to effectively generate correct policies. Furthermore, even when such knowledge is provided through the RAG module, most of the models are not capable of generating satisfying policies on the first try, failing the syntax check. Hence, the availability of deterministic verification tools is crucial to allow the system to iteratively correct itself, eventually reaching a satisfying output. Our approach achieves success most of the time, with failures related to an excessive number of retries during the policy correction (in the tests, the max amount of workflow iterations was fixed to 3) or the system's inability to modify the original Terraform file.

Surprisingly, the `Claude Sonnet 4` model already possesses the required knowledge and is capable of generating correct `Rego` rules for our tests even in the base LLM scenario, showing how powerful this cutting-edge model is. However, it is worth noting that only the agentic approach ensures that the generated rules are syntactically and semantically correct, while the simple use of an LLM does not provide any warranty in that sense. This is a consequence of the agentic approach's capacity to reason about any errors and implement appropriate corrections autonomously.

Furthermore, another crucial result obtained involves the use of smaller and cheaper models. Despite a very powerful (and costly) model, such as `Claude Sonnet 4` might be able to generate satisfying rules, ARPaCCino's approach enables the use of much smaller models (30 billion parameters in our tests) to achieve comparable results. This allows the use of one of the appropriate publicly available models in the scenario of PaC generation and IaC compliance verification, effectively eliminating dependencies on external model providers.

5.3 Model Comparison for the Agentic RAG

In the full Agentic RAG configuration of ARPaCCino, we evaluated how the choice of underlying LLM affects overall performance. As expected, the model's capabilities significantly influence task execution, primarily in terms of the number of RAG and tool invocations required to reach a successful outcome. As reported in Table 2, more powerful models tend to require fewer (average) calls to the tools to produce satisfying policies and to correct the infrastructure file. However, this reduction comes at a cost, as more powerful models are also more expensive to use, while the involved tools have negligible costs. Thus, the choice of LLM in ARPaCCino should balance between the capabilities and the cost of the executions.

Model	**RAG Call (avg)**	**Tool Call (avg)**
`Qwen3:30b`	4.4	11.4
`GPT-4o`	3.8	9.0
`Claude Sonnet 4`	3.2	7.8

Table 2. Average RAG and tool calls in the Agentic RAG

6 Conclusions and Future Work

In this work, we introduced ARPaCCino, a novel agentic system for the generation of Policy as Code and the validation of Infrastructure as Code configurations. By adopting the Agentic RAG paradigm, ARPaCCino provides an effective solution for the effortless creation and validation of security policies in IaC environments. Our results demonstrate that ARPaCCino, thanks to the use of deterministic validation tools, significantly improves performance, enabling accurate and reliable policy implementation, particularly when using smaller language models. Among the models evaluated, `Claude Sonnet 4` emerged as the most effective for this task. However, the adoption of the agentic RAG paradigm also enables the effective use of smaller models (e.g. `Qwen3:30b`), which, when supported by a well-curated knowledge base and appropriate tools, can achieve performance comparable to that of larger and more expensive models. In future work, we plan to extend ARPaCCino with automated semantic verification of the generated `Rego` rules. This remains a challenging task due to the complexity of verifying semantic correctness automatically. Additionally, we aim to explore the integration with self-RAG [29] technologies, which could further enhance the autonomy and adaptability of the system during the generation and validation phases.

Acknowledgments. This work was partially supported by the SERICS project (PE00000014) under the MUR National Recovery and Resilience Plan funded by the European Union - NextGenerationEU. The work of Francesco A. Pironti was supported by Agenzia per la cybersicurezza nazionale under the 2024-2025 funding program for promotion of XL cycle PhD research in cybersecurity (CUP H23C24000640005).

References

1. Akond Ashfaque Ur Rahman, Eric Helms, Laurie Williams, and Chris Parnin. Synthesizing continuous deployment practices used in software development. In *2015 Agile Conference*, pages 1–10. IEEE, 2015.
2. Jez Humble and David Farley. *Continuous delivery: reliable software releases through build, test, and deployment automation*. Pearson Education, 2010.
3. Michele Guerriero, Martin Garriga, Damian A Tamburri, and Fabio Palomba. Adoption, support, and challenges of infrastructure-as-code: Insights from industry. In *2019 IEEE International conference on software maintenance and evolution (ICSME)*, pages 580–589. IEEE, 2019.
4. hashicorp/terraform.
5. Red Hat. Ansible automation platform, 2024.
6. Pulumi Corporation. Pulumi: Modern infrastructure as code.
7. Akond Rahman, Effat Farhana, Chris Parnin, and Laurie Williams. Gang of eight: A defect taxonomy for infrastructure as code scripts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 752–764, 2020.
8. Mohammed Mehedi Hasan, Farzana Ahamed Bhuiyan, and Akond Rahman. Testing practices for infrastructure as code. In *Proceedings of the 1st ACM SIGSOFT International Workshop on Languages and Tools for Next-Generation Testing*, pages 7–12, 2020.
9. Md Mahadi Hassan, John Salvador, Akond Rahman, and Santu Karmaker. Large language models for it automation tasks: Are we there yet? *arXiv preprint arXiv:2505.20505*, 2025.
10. Satyadhar Joshi. A review of generative ai and devops pipelines: Ci/cd, agentic automation, mlops integration, and large language models. *CD, Agentic Automation, MLOps Integration, and Large Language Models (June 2025)*, 2025.

11. Policy as code. `https://developer.hashicorp.com/sentinel/docs/concepts/policy-as-code`.
12. Roshan N Rajapakse, Mansooreh Zahedi, M Ali Babar, and Haifeng Shen. Challenges and solutions when adopting devsecops: A systematic review. *Information and software technology*, 141:106700, 2022.
13. GitHub - open-policy-agent/opa: Open Policy Agent (OPA), June 2025.
14. Sentinel | HashiCorp Developer — developer.hashicorp.com. `https://developer.hashicorp.com/sentinel`. [Accessed 26-06-2025].
15. Michael J. Wooldridge. *An introduction to multiagent systems*. Wiley, Chichester, 2. ed. edition, 2012.
16. Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, Rui Zheng, Xiaoran Fan, Xiao Wang, Limao Xiong, Yuhao Zhou, Weiran Wang, Changhao Jiang, Yicheng Zou, Xiangyang Liu, Zhangyue Yin, Shihan Dou, Rongxiang Weng, Wenjuan Qin, Yongyan Zheng, Xipeng Qiu, Xuanjing Huang, Qi Zhang, and Tao Gui. The rise and potential of large language model based agents: a survey. *Science China Information Sciences*, 68, 2025.
17. Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in neural information processing systems*, 2020.
18. Harrison Chase. Langchain, October 2022.
19. Jerry Liu. Llamaindex, November 2022.
20. Prasad Chalasani and Somesh Jha. Langdroid.
21. Mayur Amarnath Palavalli and Mark Santolucito. Using a Feedback Loop for LLM-based Infrastructure as Code Generation, November 2024.
22. En Low, Carmen Cheh, and Binbin Chen. Repairing Infrastructure-as-Code using Large Language Models. In *2024 IEEE Secure Development Conference (SecDev)*, pages 20–27, October 2024.
23. Kalahasti Ganesh Srivatsa, Sabyasachi Mukhopadhyay, Ganesh Katrapati, and Manish Shrivastava. A Survey of using Large Language Models for Generating Infrastructure as Code, March 2024.
24. Junhee Lee, SungJoo Kang, and In-Young Ko. An LLM-driven Framework for Dynamic Infrastructure as Code Generation. In *Proceedings of the 25th International Middleware Conference: Demos, Posters and Doctoral Symposium*, pages 9–10, Hong Kong Hong Kong, December 2024. ACM.
25. Tianyi Zhang, Shidong Pan, Zejun Zhang, Zhenchang Xing, and Xiaoyu Sun. Deployability-Centric Infrastructure-as-Code Generation: An LLM-based Iterative Framework, June 2025.
26. Matteo Lupinacci, Francesco Blefari, Francesco Romeo, Francesco Aurelio Pironti, and Angelo Furfaro. ARCeR: An agentic RAG for the Automated Definition of Cyber Ranges. In *Availability, Reliability and Security*, pages 23–40, Cham, 2025. Springer Nature Switzerland.
27. Fabio Martinelli, Francesco Mercaldo, Luca Petrillo, and Antonella Santone. Security Policy Generation and Verification through Large Language Models: A Proposal. In *Proceedings of the Fourteenth ACM Conference on Data and Application Security and Privacy*, pages 143–145, Porto Portugal, June 2024. ACM.
28. GitHub - bpg/terraform-provider-proxmox: Terraform / OpenTofu Provider for Proxmox VE, June 2025.
29. Akari Asai, Zeqiu Wu, Yizhong Wang, Avirup Sil, and Hannaneh Hajishirzi. Self-rag: Learning to retrieve, generate, and critique through self-reflection, 2023.