

Constraint-Aware Infrastructure Script Generation with Multi-Agent Verification

Authors

Mia R. Sullivan^{1*}, Lachlan T. Evans², Claire J. Howard³

Affiliations

School of Computing and Information Systems, Faculty of Engineering and Information Technology, The University of Melbourne, Grattan Street, Parkville, VIC 3010, Australia

*Corresponding author: m.sullivan@unimelb.edu.au

Abstract

Infrastructure-as-code scripts are now widely used to manage cloud deployment, but automatically generated configuration files often contain incorrect resource dependencies, unsafe permission settings, invalid network rules, and excessive resource allocation. This study examines infrastructure script generation under deployment, security, and cost constraints. We propose DeployAgent, a multi-agent model that coordinates a requirement interpretation agent, a cloud resource planning agent, a configuration generation agent, a policy verification agent, and a cost-tuning agent. The requirement interpretation agent extracts service topology, workload scale, storage demand, access rules, and availability targets from natural-language deployment requests. The resource planning agent selects cloud components such as containers, load balancers, databases, storage buckets, and virtual networks. The configuration generation agent produces Terraform, Kubernetes YAML, and Helm configuration files. The policy verification agent checks dependency order, namespace isolation, secret exposure, least-privilege access, and network reachability. The cost-tuning agent adjusts instance size, replica count, autoscaling thresholds, and storage classes. The evaluation used deployment records from three cloud-native software projects over a six-month period. These records included 312 real deployment tickets, 1,870 historical configuration files, 46 service dependency graphs, and runtime logs from 11 Kubernetes clusters. In the replay experiment, DeployAgent successfully generated deployable configurations for 81.4% of tickets, while the single-agent model achieved 63.7%. Manual review found that unsafe permission assignments were reduced from 22.6 cases per 100 scripts to 7.9 cases, and invalid dependency definitions were reduced by 48.3%. In a staged deployment environment, the rollback rate decreased from 18.2% to 6.5%. Resource tuning also reduced average monthly cloud cost estimates by 14.7% without violating CPU, memory, or availability requirements. These findings show that multi-agent verification can improve the reliability, security, and economic efficiency of generated infrastructure code.

Keywords: Infrastructure as code; cloud deployment; configuration generation; policy verification; Kubernetes; Terraform; multi-agent systems

1. Introduction

Infrastructure-as-code (IaC) has become a common approach for provisioning and managing cloud infrastructure because it allows deployment specifications to be stored, reviewed, versioned, and reused in the same manner as software code. Tools such as Terraform, Kubernetes YAML, and Helm charts are widely used to define computing resources, network configurations, access permissions, storage, scaling policies, and application deployment parameters. Despite these advantages, IaC files are highly sensitive to configuration errors. Incorrect resource definitions may result in deployment failures, excessive privileges, unintended public access, exposed credentials, or inefficient resource allocation [1]. Previous studies have identified security weaknesses in Kubernetes manifests and other cloud configuration files, including insecure network policies, overly permissive access controls, and inappropriate secret management [2,3]. Defects have also been reported in Helm charts and IaC state-management processes, where inconsistencies between declared and deployed infrastructure may affect system reliability and reproducibility [4,5]. More broadly, machine-learning systems may rely on unintended shortcuts or spurious correlations rather than the underlying causal information required for reliable decisions, and identifying and removing multiple such shortcuts has been shown to improve model robustness [6]. This issue is relevant to automated IaC generation because configuration files contain repeated structural patterns that may allow a model to produce superficially plausible code without correctly representing deployment dependencies or operational constraints. IaC faults can therefore arise not only from syntax errors but also from tool behavior, incorrect dependency ordering, invalid access policies, and inconsistencies between configuration intent and runtime execution [7,8].

Large language models (LLMs) have recently been applied to code generation, software repair, configuration synthesis, and other software-engineering tasks. They can translate natural-language requirements into executable code and reduce the effort required to construct complex deployment specifications [9]. However, generated code may still contain logical errors, incomplete dependencies, invalid parameters, and security-sensitive operations that are difficult to detect from syntax alone [10,11]. These problems can be more serious in IaC than in conventional application code because a small configuration error may directly affect cloud resources, network exposure, access control, or operational cost [12,13]. Multi-agent approaches provide a possible way to improve reliability by decomposing a software task into specialized activities such as requirement interpretation, planning, code generation, validation, testing, and iterative revision [14,15]. Recent studies have extended these approaches to IaC generation, automated deployment testing, execution-feedback repair, and benchmark-based evaluation [16,17]. Their results suggest that iterative verification and role specialization can improve generated configurations, while also showing that successful text generation does not necessarily imply successful deployment [18-22]. Several limitations nevertheless remain in current automated IaC generation and validation methods. Many approaches concentrate on syntactic correctness or static rule checking, while

deployment behavior is examined only through limited test cases. Some systems are designed for a single cloud platform or a specific configuration language, which reduces their applicability to heterogeneous cloud environments [23-26]. Security evaluation is also frequently separated from deployment validation, even though permissions, network exposure, secret handling, and resource dependencies directly influence whether a configuration can be safely executed [24-28]. Cost-related settings receive even less attention. A configuration may be functionally valid and secure but still allocate excessive CPU[29-31], memory, storage, or redundant cloud resources. In addition, benchmark-based evaluation often measures whether a generated file matches a reference solution, rather than whether it can be deployed successfully and behave correctly under realistic runtime conditions. These limitations indicate the need for an integrated framework that evaluates IaC from operational, security, and resource-efficiency perspectives within the same generation and verification process[32-35].

This study develops DeployAgent, a multi-agent framework for the automated generation and verification of cloud infrastructure configurations. DeployAgent organizes the workflow around five specialized agents responsible for requirement analysis, resource planning, configuration generation, policy checking, and cost tuning. The framework generates Terraform, Kubernetes YAML, and Helm configurations and subsequently examines resource dependencies, access permissions, network rules, secret exposure, deployment constraints, and resource-allocation settings before the configurations are accepted. Its effectiveness is evaluated using real deployment tickets, historical configuration files, infrastructure dependency graphs, and Kubernetes runtime logs, allowing the generated outputs to be assessed beyond surface-level code similarity or syntax validity. By connecting natural-language requirements with multi-stage configuration generation, security inspection, deployment-oriented verification, and cost-aware optimization, this study aims to provide a more practical approach to trustworthy IaC automation. The proposed framework is intended to reduce configuration defects before deployment, improve the reliability of LLM-generated cloud infrastructure, and provide a reproducible basis for evaluating multi-agent systems in realistic DevOps and cloud-engineering environments.

2. Materials and Methods

2.1. Sample and Study Setting

This study used records from three cloud-native software projects collected over six months. The dataset included 312 real deployment tickets, 1,870 historical configuration files, 46 service dependency graphs, and runtime logs from 11 Kubernetes clusters. A ticket was selected only when it included a clear deployment request, related configuration files, and a recorded deployment result. The tasks involved container services, load balancers, databases, object storage, virtual networks, access-control rules, secrets, and autoscaling settings. Terraform files, Kubernetes YAML files, and Helm charts were kept in their original formats. Each ticket was

treated as one deployment task. Its linked files, dependency graph, and runtime logs were used as supporting context.[36-38]

2.2. Experimental Design and Control Conditions

DeployAgent was compared with a single-agent configuration generator. DeployAgent used five agents for requirement reading, resource planning, file generation, policy checking, and cost tuning. The control model received the same deployment request and generated configuration files in one step. Both systems used the same language model, prompt length, input data, output format, and test environment[39-42]. The original configuration linked to a ticket was not provided during generation. This prevented the model from copying the reference solution. Each generated output was first tested in a replay setting and then deployed in a staged Kubernetes environment. This comparison showed whether separate agent roles and constraint checks improved the generated scripts.

2.3. Measurement Methods and Quality Control

Terraform files were checked through syntax validation and execution planning. Kubernetes YAML files were checked through schema validation and server-side dry-run tests. Helm charts were rendered and linted before deployment. A script was counted as deployable only when it passed file checks, created the required resources, and reached a healthy service state in the staged environment. Security checks covered excessive permissions, open network access, exposed secrets, missing namespace isolation, and violations of least-privilege rules. Dependency errors included missing resource links, incorrect deployment order, and invalid network connections. Monthly cost was estimated with the cloud-provider price catalog used by each project. Tool versions, cluster settings, security rules, and test images were fixed during the experiment. Two DevOps reviewers checked the security and dependency results independently. Differences were discussed until a final decision was reached.

2.4. Data Processing and Model Formulation

Deployment requests and configuration files were converted into a common schema. The schema recorded resource type, dependency relation, permission rule, network rule, workload demand, and availability target. Missing information was kept as unknown and was not replaced with assumed values. Resource cost was scaled before candidate selection. DeployAgent selected the configuration that obtained the highest combined score for deployment, policy compliance, requirement matching, and cost:

$$\max_{c \in C} [\alpha D(c) + \beta P(c) + \gamma R(c) - \delta \tilde{C}(c)] \quad \text{subject to} \quad g_j(c) \leq 0, j=1, \dots, m$$

The weights α, β, γ , and δ represented the importance of deployment status, policy compliance, requirement matching, and cost. The constraint functions covered access permissions, network reachability, CPU, memory, storage, and service availability. Deployment success rate was calculated as:

$$DSR = \frac{1}{N} \sum_{i=1}^N I(d_i=1)$$

A value of one indicated that the script passed validation, completed staged deployment, and reached the required service state.

2.5. Evaluation Metrics and Statistical Analysis

The main measures were deployment success rate, unsafe permission assignments per 100 scripts, invalid dependency rate, staged rollback rate, and estimated monthly cloud cost. A deployment was accepted only when all required services were available and no blocking policy error was found. Cost reduction was calculated from the same tickets under both systems. Results were reported as percentages or mean values for all tested tickets. Confidence intervals were estimated by bootstrap resampling at the ticket level. Paired deployment results were compared with McNemar's test. Paired cost values were compared with the Wilcoxon signed-rank test. A two-sided significance level of 0.05 was used.

3. Results and Discussion

3.1. Deployment Success of Generated Scripts

DeployAgent produced deployable configurations for 81.4% of the 312 tickets. The single-agent generator reached 63.7%. The difference was 17.7 percentage points. The single-agent model often generated files that met basic format rules but missed service links, resource settings, or deployment order. DeployAgent first identified the required resources and their relations, then created the configuration files [24,25]. Fig. 1 shows a linter-feedback loop used in earlier IaC generation work. Such a loop can correct errors after file generation, but it does not fully address planning errors at the start. The higher deployment rate indicates that early resource planning reduced these errors.

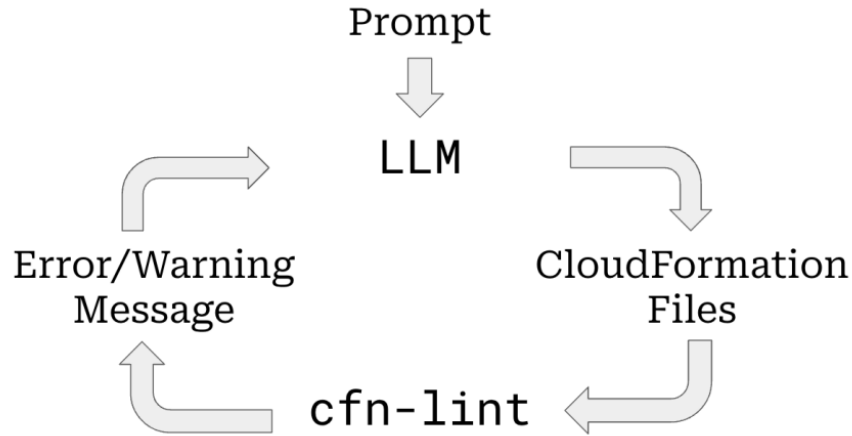


Fig. 1. Linter feedback for correcting generated infrastructure-as-code files.

3.2. Permission and Dependency Errors

Manual review recorded 7.9 unsafe permission assignments per 100 scripts for DeployAgent, compared with 22.6 cases for the single-agent model. This was a 65.0% reduction. Invalid dependency definitions also decreased by 48.3%. The policy-checking agent examined access roles, secret use, namespace settings, and network rules before the final files were returned. The resource-planning agent checked links among services, databases, storage, and networks. Earlier IaC generation methods often focused on syntax checks or failed deployments. The findings show that permission and dependency checks should be part of the generation process [26,27].

3.3. Staged Deployment and Rollback Results

The rollback rate in the staged environment decreased from 18.2% for the single-agent model to 6.5% for DeployAgent. This was a decrease of 11.7 percentage points. A configuration file may pass syntax checks but still fail during resource creation. Common causes include missing dependencies, incorrect access rules, unavailable settings, and unsuitable resource sizes [28,29]. Fig. 2 shows an iterative process that links file generation with deployment checks. DeployAgent added policy and capacity checks before staged deployment. The lower rollback rate shows that these checks reduced deployment failures.

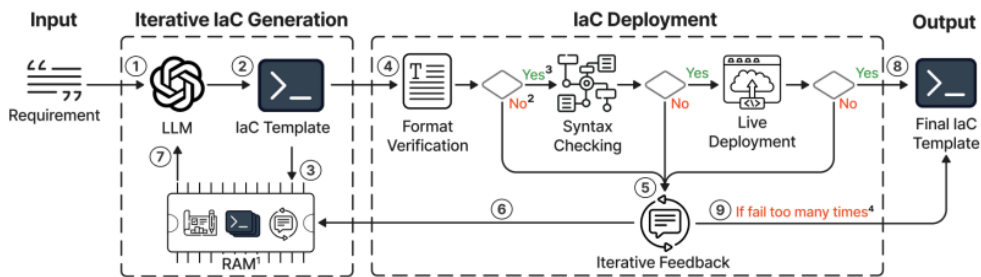


Fig. 2. Iterative generation and deployment validation of infrastructure scripts.

3.4. Cost-Tuning Results and Practical Meaning

DeployAgent reduced estimated monthly cloud cost by 14.7% without breaking the stated CPU, memory, or availability requirements. The cost agent changed instance size, replica count, autoscaling thresholds, and storage classes after the service design had been checked. The lower cost did not result from removing required resources or reducing service availability. Previous IaC studies have mainly assessed file validity and deployment success. Cost is often reviewed after deployment. The current results show that cost can be checked during script generation. The dataset covered three projects and staged clusters. Further tests should include more cloud providers, larger workloads, and long-term billing records.

4. Conclusion

This study evaluated DeployAgent for infrastructure script generation under deployment, security, and cost requirements. The system used separate agents to read requests, plan resources, generate files, check policies, and adjust cost settings. It achieved an 81.4% deployment success rate, while the single-agent model achieved 63.7%. Unsafe permission assignments fell from 22.6 to 7.9 cases per 100 scripts. Invalid dependency definitions decreased by 48.3%, and the rollback rate fell from 18.2% to 6.5%. Estimated monthly cloud cost decreased by 14.7% without reducing required CPU, memory, or service availability. The findings show that generated infrastructure files should be checked for service links, permissions, network rules, and resource cost before deployment. DeployAgent may support DevOps teams and cloud deployment platforms that need safer and lower-cost configuration files. The study included only three projects and staged Kubernetes clusters. Future work should test more cloud providers, larger systems, real production workloads, and long-term cost records.

References

1. Zhang, Z. (2026). A Study on the Impact of Cost Presentation on Decision-making Bias and Cancellation Behavior in Online Subscriptions. Available at SSRN 7349839.
2. Mini Han Wang , " AI-Powered Innovations in Ophthalmic Diagnosis and Treatment ", Bentham Science Publishers (2025). <https://doi.org/10.2174/97988988122871250101>.
3. Qi, C., & Qiao, X. (2026). Building and Operating a Large Scale Multi-Agent System: A Case Study from Industry. Available at SSRN 6795198.
4. Hong, Z., Liang, S., Xu, T., & Chen, H. (2026). A Study on Failover Verification and Recovery Objective Prediction for Cross-Region Cloud Services.
5. Quan, Q., Gao, Y., & Wang, Q. (2025). The impact of teacher emotional support on students' engagement in AI-mediated English learning environments: The mediating role of resilience and self-efficacy. *Acta Psychologica*, 260, 105766.
6. Zhang, Z., Wang, J., Li, Z., Wang, Y., & Zheng, J. (2025). Annocoder: A mti-agent-based code generation and optimization model. *Symmetry*, 17(7), 1087.

7. Wu, J., Zhang, J., Wu, D., & Peng, Y. (2026). Visual Transformation Mechanisms for Integrating Digital Cultural Heritage Resources into Junior High School Art Curricula.
8. Huang, J., Xu, T., Yin, J., & Yang, J. (2026). Data Quality Monitoring and Intelligent Scheduling Optimization in Financial Data Workflow Automation. Available at SSRN 7179338.
9. Xu, Y., Cui, J., Zhu, Z., Shang, H., Luan, S., Xu, M., ... & Zhu, S. (2026). Wam-flow: Parallel coarse-to-fine motion planning via discrete flow matching for autonomous driving. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (pp. 24918-24928).
10. Zheng, J., & Makar, M. (2022). Causally motivated multi-shortcut identification and removal. *Advances in Neural Information Processing Systems*, 35, 12800-12812.
11. Xu, T., Zhu, W., & Zhang, J. (2026). A Study on the Application of Alternative Data in Credit Assessment for the Unbanked Population.
12. Xiong, W., Guo, Y., & Zeng, Y. (2026). A Study on the Energy Efficiency of MEP Systems and Coordinated Dispatch Mechanisms for Multi-energy Systems in High-density Urban Buildings. Available at SSRN 7120518.
13. Zheng, X., Chen, X., Gong, S., Griffin, X., & Slabaugh, G. (2025, September). Xfmamba: Cross-fusion mamba for multi-view medical image classification. In *International Conference on Medical Image Computing and Computer-Assisted Intervention* (pp. 672-682). Cham: Springer Nature Switzerland.
14. Zhao, J., Fan, J., & Li, L. (2026). A Study on an Explainable Causal-Enhanced LLM Agent for Predicting the Forming Quality of Automotive Component Materials.
15. You, S. (2026). Verifiable Audit Mechanisms in AI Compliance Automation: Scalability. Available at SSRN 6547458.
16. Jiao, Y., Shi, T., Zhao, B., & Wang, A. (2026). The Impact of VR Sketching Simulations on the Assessment of Scale and Site Suitability in Public Art Spaces.
17. Su, J., He, C., Zhu, F., Xu, X., Guan, D., & Si, C. (2024, September). Hpff: Hierarchical locally supervised learning with patch feature fusion. In *European Conference on Computer Vision* (pp. 293-309). Cham: Springer Nature Switzerland.
18. Liang, S., Du, Y., Chen, W., & Liu, Z. (2026). Order Allocation and Emergency Transportation Decisions Amid Multi-Source Procurement Disruptions.
19. Bao, Y., & Wang, H. (2026). Orion: A High-Throughput, Fault-Tolerant Event Routing Architecture for Hyperscale Data Streams. *Fault-Tolerant Event Routing Architecture for Hyperscale Data Streams* (January 01, 2026).
20. Yang, Q., & Du, Y. (2026). Predicting Rollback Risks and Controlling Gradual Rollout Traffic for Online Ranking Feature Deployments.
21. Lyu, Y., Liu, K., Cao, Z., Luo, Y., Yin, L., Yang, T., & Chen, Z. (2026). FLAIR: An LLM-integrated multi-layer framework for fine-grained activity type inference in human trajectory data. *Transportation Research Part C: Emerging Technologies*, 190, 105787.

22. Zhao, Z., & Welsch, R. E. (2026). Point-in-Time Financial RAG with Frozen LLMs and Market-Feedback Adaptive Retrieval. arXiv preprint arXiv:2605.31201.
23. Yuan, Y., Huang, J., Yin, J., & Xu, T. (2026). Confidence Calibration and Semantic Error Analysis for Ambiguous Coreference Resolution in User-generated Social Media Text. Available at SSRN 7393238.
24. Huang, Q., Xu, Z., Zhang, X., & Zhang, J. (2025). UniShield: An Adaptive Multi-Agent Framework for Unified Forgery Image Detection and Localization. arXiv preprint arXiv:2510.03161.
25. Liang, J., Xiao, Y., Yang, H., Yu, Z., Liu, J. N., & Yang, K. (2026, April). C-HyPOD: Causal Hyperbolic Representation Learning with Prototype Orthogonal Disentanglement for Graph Out-of-Distribution Recommendation. In Proceedings of the ACM Web Conference 2026 (pp. 6541-6550).
26. Xi, X., Chen, P., Qiu, Y., Luo, R., Tong, P., & Liang, J. Video-BCI: Bayesian Cognitive Integration of Self-Prior Hypotheses for Video Understanding. In Forty-third International Conference on Machine Learning.
27. Liang, S., Hsu, K. H., & Liu, Z. (2026). Rolling Scheduling and Capacity Balancing for Server Assembly Under Engineering Change Disturbances.
28. Xu, Z., Zhang, X., Li, R., Tang, Z., Huang, Q., & Zhang, J. (2024). Fakeshield: Explainable image forgery detection and localization via multi-modal large language models. arXiv preprint arXiv:2410.02761.
29. Chaoyong Ma, Nan Si, Xuwei Song, Chen Liang and Haoming Luo, A Bearing Fault Diagnosis Method Based on Concise Empirical Wavelet Transform, International Journal of Comprehensive Engineering, 2023, 12(1), 1-6
30. Bing Wang, Haihong Tang, Xiaojia Zu and Wuwei Feng, Application of Parameter Adaptive VMD in Bearing Fault Diagnosis, International Journal of Comprehensive Engineering, 2023, 12(1), 34-42
31. Bao, Y., Qiu, Y., & Wang, H. (2026). FLARE: A Real-Time Framework for Detecting Malicious Account Activities at Internet Scale. Available at SSRN 7185659.
32. Xu, Z., Zhang, X., Zhou, X., & Zhang, J. (2025). AvatarShield: Visual Reinforcement Learning for Human-Centric Video Forgery Detection. arXiv preprint arXiv:2505.15173.
33. Yuan, Y., Xu, T., Yin, J., & Huang, J. (2026). Modeling Loading Anomalies and Identifying Root Causes in Media Consumption Workflows on Large Social Media Platforms. Available at SSRN 7393318.
34. Tian Teng and Xiao Zhang, Design Principles and Feasibility Assessment of an Intelligent Spare-Parts Shelving System for Shipboard Storage Environments, Journal of Comprehensive Engineering, 2026, 15(1), 24-30

35. Jiang, C. (2026). Information doesn't work? Testing experiential drivers of VR application in heritage tourism by affective-cognitive pathways. *Journal of Hospitality and Tourism Technology*, 1-28.
36. Jianheng Liang, Yang Yang, Ran Zhang, Guican Wu, Yuanyuan Yang, TaiFu Li, Honglin Duan and Yuyan Li, Research on Perfume User Experience Evaluation Based on Human Machine Hybrid Generation Adversarial Network, *International Journal of Comprehensive Engineering*, 2024, 13(1), 40-54
37. Xi Qiao, Di Zhao, Nan Si, Jifeng Sui and Yonggang Xu, Weighting Spectrum Editing Based on Amplitude Mapping and Its Application in Composite Fault Diagnosis of Rolling Bearings, *International Journal of Comprehensive Engineering*, 2024, 13(1), 13-21
38. Wang, T., & Xia, Z. (2025). Stability of In-Context Learning: A Spectral Coverage Perspective. *arXiv preprint arXiv:2509.20677*.
39. Xixin Chen, Ming Tu, Lei Su and Ke Li, Rolling Bearing Fault Diagnosis Based on Digital Twin Method, *International Journal of Comprehensive Engineering*, 2024, 13(1), 22-39
40. Liu, T., He, J., Zhou, R., Song, J., Liu, H., & Liang, Y. (2025). Revolutionizing clinical decision making through deep learning and topic modeling for pathway optimization. *Scientific Reports*, 15(1), Article 28787. <https://doi.org/10.1038/s41598-025-12679-z>
41. Wu, X., Xing, X., Yao, J., Qian, Q., & Song, J. (2025). Scnet: spectral convolutional networks for multivariate time series classification. *Applied Intelligence*, 55(6), Article 456. <https://doi.org/10.1007/s10489-025-06352-1>
42. Ma, C., Song, J., Xu, Y., Fan, H., Wu, X., & Sun, T. (2024). Vehicle-Based Machine Vision Approaches in Intelligent Connected System. *IEEE Transactions on Intelligent Transportation Systems*, 25(3), 2827-2836. <https://doi.org/10.1109/TITS.2023.3276325>