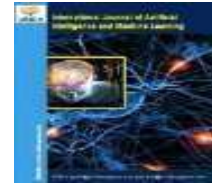




International Journal of Artificial Intelligence and Machine Learning

Publisher's Home Page: <https://www.svedbergopen.com/>



Research Paper

Open Access

Policy-Driven Automation for Standardized Enterprise Database Provisioning Across Heterogeneous Cloud Platforms

Srihari Tubati

Independent Researcher, USA

Abstract

Ticket-based database provisioning rarely fails loudly. It fails by drift: a network rule loosened during an incident and never retightened, a monitoring hook skipped under deadline pressure, a tag left inconsistent - none of it visible as a problem until an audit, or a breach, forces someone to reconstruct what was actually configured against what policy required in the first place. This article presents a policy-driven provisioning framework built around one separation: what an enterprise requires of a database environment, versus how each cloud platform satisfies that requirement. The second half is expressed through platform-specific provisioning profiles, each implementing a single shared enterprise policy model. The architecture draws on direct experience standardizing provisioning automation across several major cloud database platforms, alongside a related AWS-native deployment automation effort supporting more than 150 developers across thousands of deployment operations. Stated early and explicitly: some of what follows was implemented and observed, and some extends that implemented core into a generalized architecture not yet fully deployed. Approval gating, automated evidence generation, continuous drift detection as its own control loop, and exception governance are presented as architectural design - not uniformly as measured, deployed capability. Applied evidence, order-of-magnitude and practitioner-reported rather than drawn from a controlled study, indicates provisioning turnaround moved from days to minutes, with a substantial reduction in repetitive manual engineering effort alongside it. The discussion argues that policy-as-code's real governance value in provisioning is structural: it converts compliance from an after-the-fact audit exercise into a built-in property of how an environment is allowed to exist at all, while acknowledging plainly that automation raises the blast radius of a defective policy rather than removing the need for that policy to be correct.

Keywords: Policy-driven automation; provisioning profile; policy-as-code; configuration drift; cloud database provisioning; enterprise governance standard; approval gate; multi-cloud infrastructure automation

1. Introduction

A ticket sits in a queue. Whoever picks it up decides, on that day, using whatever they happen to remember about that platform's requirements, what the resulting database environment actually looks like. That is the real starting condition for most enterprise database provisioning - and it is a worse problem than slow turnaround. Turnaround time is visible; it gets escalated, tracked, complained about. Configuration inconsistency is invisible until it costs something: a security audit that cannot confirm what was actually deployed, an incident where the isolation boundary an engineer assumed was in place turns out never to have been enforced.

This article documents a framework built to close that gap - a policy-driven provisioning architecture that defines enterprise requirements for database environments once, at the policy layer, and lets each supported cloud platform satisfy those requirements through its own provisioning profile. The underlying idea is not new to infrastructure engineering broadly; separating durable requirement from swappable mechanism is a familiar pattern in large-scale systems. What is specific to this article is its application to standardized, governed database provisioning across genuinely heterogeneous cloud platforms, inside a regulated financial-services production environment.

1.1 Scope, and an honest boundary between what was built and what is proposed

Scholarly honesty requires drawing a boundary that most practitioner writing blurs. The core of what follows - translating a standard database-service request into platform-specific provisioning steps governed by a shared policy model, and enforcing naming, tagging, and capacity policy at release time rather than leaving it to convention - reflects work that was actually implemented and operated. Several mechanisms discussed later extend that implemented core into a more complete, generalized architecture: an explicit approval-gate

model for higher-risk requests, automated evidence-record generation for compliance review, continuous drift detection run as its own recurring control loop separate from provisioning-time checks, and a formal exception and waiver governance model. These are presented as architectural design, and in places partial implementation, rather than as uniformly deployed and measured capability. Wherever a claim below is a generalization rather than a direct operational result, this article says so.

The scope stays deliberately narrow: standing up a new, policy-compliant database environment, and keeping its configuration from drifting afterward. Data migration, schema deployment, application-credential lifecycle management, query tuning, patch and engine-version upgrades, replication configuration, disaster-recovery testing, and release management sit explicitly outside that scope. Conflating provisioning with these adjacent concerns is a common source of overclaiming in infrastructure-automation writing, and this article tries hard to avoid it.

Research question: how can a common enterprise policy model, implemented through platform-specific provisioning profiles, standardize database-environment delivery across heterogeneous cloud platforms - and what operational effects can actually be documented from the automation that exists today?

2. Literature Review

2.1 Configuration as data, and its costs at scale

Treating infrastructure configuration as declarative data rather than tribal knowledge executed by hand is not a new idea in large-scale systems engineering. Facebook's holistic configuration-management work established, at considerable scale, that configuration errors are a leading cause of production incidents precisely because configuration sits outside the testing and review discipline normally applied to code (Tang et al., 2015). That framing - configuration as a first-class systems-engineering artifact, not an afterthought - underlies nearly every later argument in this article, even though the platforms and tooling landscape have changed a great deal since 2015. Xu and Zhou's survey of configuration-error approaches makes a related point from a different angle: most configuration failures are not exotic. They are the predictable consequence of unchecked defaults, undocumented dependencies, and manual-entry error, which happens to be exactly the failure mode ticket-driven provisioning invites (Xu & Zhou, 2015). Both works predate the current infrastructure-as-code and policy-as-code tooling generation. They are retained here specifically because no post-2020 peer-reviewed source establishes the foundational configuration-as-liability argument with comparable empirical grounding at comparable scale.

2.2 Infrastructure-as-code adoption, and where it still leaks

The empirical literature on infrastructure-as-code quality is considerably more mature than the drift-and-compliance literature it ought to feed into. Regvart et al.'s controlled comparison of Terraform, Pulumi, and AWS CloudFormation shows that provisioning-tool choice carries measurable performance consequences rather than being a purely stylistic decision: CloudFormation's provisioning time ran to roughly 2.4 times that of Terraform under their test conditions, 622.97 seconds against 253.33 (Regvart et al., 2026). That matters for a policy-driven architecture for a specific reason - provisioning-profile implementation choices are not policy-neutral. The platform-specific mechanism layer carries its own performance and maintainability costs, independent of whatever policy it happens to be enforcing.

Code-review practice for infrastructure-as-code reveals a second tension worth sitting with. Bessghaier et al.'s large-scale study of OpenStack infrastructure-as-code reviews finds that infrastructure changes churn more than comparable application code, and that review effort concentrates heavily among a small number of specialist reviewers (Bessghaier et al., 2025). That concentration pattern connects directly to enterprise provisioning governance: a policy model that still requires deep platform-specific review expertise for every single environment has not actually solved the bottleneck it claims to solve - it has just moved it somewhere else. Misconfiguration taxonomies compound the concern. Nasiri et al.'s cataloging of Ansible configuration errors shows that a meaningful share of infrastructure defects are structurally repeatable patterns, not one-off mistakes (Nasiri et al., 2024), which argues for encoding those patterns once, in policy, rather than letting each engineer rediscover them per platform. War et al. push this further still, cataloging infrastructure-as-code vulnerabilities across the full OWASP Top-10 category set and establishing that infrastructure-layer weaknesses span access control, secrets handling, and misconfiguration alike - not a narrow, easily-bounded risk surface (War et al., 2025).

2.3 Policy-as-code: adoption reality versus the tool's promise

Policy-as-code tooling - engines like Open Policy Agent, applied to infrastructure and deployment pipelines rather than authored from scratch by the practitioners adopting them - has moved from niche practice into a recognizable category of its own. Foalem et al.'s empirical study of open-source policy-as-code adoption puts a concrete shape on that category: OPA accounts for a substantial majority of the policy files they examined, and

security and compliance use cases dominate why organizations adopt the pattern at all (Foalem et al., 2026). That finding validates the premise behind this article's architecture, since security and compliance are exactly the domain where a shared policy model pays off fastest. But it also exposes a gap. Adoption-pattern studies document that organizations write policy as code - not that the resulting policies stay correctly enforced once environments exist and keep changing.

Nogueira and Resende's assessment of reusing policy-as-code across CI/CD and Kubernetes admission control speaks directly to the durable-policy-versus-swappable-mechanism argument this article is making. Testing the same policy logic against 29 manifests across 37 scenarios, they report detection accuracy and false-positive and false-negative rates that are, frankly, better than most practitioners would probably expect from a single shared policy layer applied across two structurally different enforcement points (Nogueira & Resende, 2026). That result is encouraging for the provisioning-profile model proposed here, though their evaluation is scenario-based rather than a longitudinal production deployment - a limitation this article's own applied evidence shares, for what it's worth.

Continuous, runtime-level compliance is where the literature thins out noticeably. Falazi et al. describe compliance management for infrastructure-as-code-based deployments specifically during runtime rather than only at deployment time, treating drift detection as a distinct architectural concern from initial provisioning correctness (Falazi et al., 2023) - a framing this article adopts directly for its own drift-detection discussion. Angermeir et al. push toward automated continuous security compliance as a target state rather than a solved problem, which is itself telling: the field is still working out what continuous ought to mean operationally, not just conceptually (Angermeir et al., 2024).

2.4 Cloud trustworthiness, provider variation, and why one profile per platform is necessary

Not every cloud provider implements a given control the same way, and that variation turns out to be measurable rather than anecdotal. Akhtar et al.'s trustworthiness scoring across cloud service providers found scores ranging from roughly 0.78 to 0.84 for the strongest-performing provider down to 0.44 to 0.55 for the weakest, depending on compliance posture - establishing empirically that the cloud is not a uniform compliance surface (Akhtar et al., 2024). Verdet et al.'s analysis of security-policy adoption in Terraform configurations across AWS, Azure, and GCP sharpens this further: access-policy controls saw the strongest adoption across providers in their sample, while encryption-at-rest was, somewhat surprisingly, the most neglected control category regardless of provider (Verdet et al., 2025). That specific asymmetry - access control gets attention, encryption-at-rest quietly does not - is exactly the kind of platform-crossing inconsistency a shared enterprise policy model, enforced identically no matter which team configured which platform, is meant to close.

Portability middleware research offers a related but distinct angle. Bharany et al.'s work on PaaS-consuming-application portability across heterogeneous clouds addresses application portability rather than provisioning-policy portability, but its underlying premise - that heterogeneous cloud platforms need an abstraction layer to avoid re-deriving the same integration logic per platform - parallels this article's provisioning-profile argument closely enough to be worth citing as adjacent evidence (Bharany et al., 2022). The NIST cloud computing reference architecture remains the standard vocabulary for describing multi-provider cloud service relationships, and it is retained here as a foundational, pre-2020 reference because no later peer-reviewed work has really superseded it as the baseline architectural taxonomy for cloud service roles (Liu et al., 2011).

2.5 Automated remediation, and the limits of just automate the fix

A recurring temptation in drift-detection design is to auto-remediate everything the system detects. The empirical record argues against that instinct. Xu, Gao, and Wei's large-scale study of Kubernetes operator bugs - cited here strictly as literature on automated-remediation risk, not as any claim about operator expertise - found that most examined bugs traced to incorrect state observation, and that existing detection tooling caught only about 55 percent of them (Xu et al., 2024). Aghili et al.'s characterization of AIOps projects reinforces the same caution from the operations-automation side: automated operational tooling is maturing, but it remains unevenly reliable across the practices it claims to automate (Aghili et al., 2023). Bulut and Hwang's work mapping natural-language security findings to standardized vulnerability scores gestures toward one path forward - machine-assisted triage that still routes judgment calls to a human, rather than closing the loop automatically (Bulut & Hwang, 2021). Romeo et al.'s more recent agentic, retrieval-augmented approach to policy-as-code compliance checking extends that triage idea toward large-language-model-assisted policy verification, though it reads as an emerging technique rather than an established production practice (Romeo et al., 2026).

2.6 DevSecOps practice synthesis and the distributed-transaction problem

Prates and Pereira's broad mapping of DevSecOps practices across more than a hundred primary studies situates policy-as-code and automated compliance checking within a much larger practice landscape - and the mapping's sheer breadth is itself instructive. Automated provisioning governance is one of dozens of controls that an organization is expected to coordinate, rather than a standalone solution (Prates & Pereira, 2025). Finally, the failure-handling design proposed in this article - compensating action rather than transactional rollback - has a direct analogue in distributed-transaction research. Daraghmi et al.'s work on enhanced Saga patterns for microservices demonstrates, under load testing, that compensation-based transaction handling can materially outperform a baseline approach in completed-transaction volume: roughly 9,128 completed orders versus 6,252 for the baseline. That is the closest empirical validation available in the approved literature for treating infrastructure provisioning failure as a compensation problem rather than an atomic-rollback problem (Daraghmi et al., 2022).

2.7 The gap this article addresses

Read together, this literature establishes three things fairly clearly, and leaves one thing largely unaddressed. Clearly established: configuration-as-liability is a well-founded premise (Tang et al., 2015; Xu & Zhou, 2015); policy-as-code tooling has real, measurable adoption concentrated in security and compliance use cases (Foalem et al., 2026); and cloud providers genuinely differ in how they implement equivalent controls, which is precisely why a single implementation cannot simply be reused unmodified across platforms (Akhtar et al., 2024; Verdet et al., 2025). Left largely unaddressed: sustained, production-scale evidence of what happens to policy-as-code provisioning governance after the environment is created - ongoing drift, exception handling, and evidentiary compliance at enterprise scale, across genuinely heterogeneous platforms, over time. This article's applied contribution addresses that gap, with the caveat that its own evidence for the post-provisioning half of that gap is architectural rather than fully measured.

3. Materials and Methods

3.1 Separating enterprise policy from platform implementation

The architecture's foundational design decision is to define security, resilience, monitoring, naming, and capacity requirements exactly once, at the enterprise level, rather than separately for each cloud platform. Each supported platform then implements those requirements through its own provisioning profile - a declared, testable mapping from the enterprise policy model onto that platform's actual mechanisms. A profile specifies how the platform achieves network isolation, which encryption and key-management primitives satisfy the enterprise's data-protection requirement, how backup configuration and retention map onto policy, which monitoring and logging hooks exist and how they connect to central observability, and what capacity and resilience settings - replica counts, availability-zone spread - satisfy the enterprise's capacity and resilience baseline. Naming and tagging conventions live inside the profile too, since the literal convention differs slightly by platform even when the underlying intent is identical everywhere. Table 1 illustrates this control mapping across two representative platform categories.

This is not, on its own, a novel abstraction pattern - separating durable requirements from swappable mechanisms is common practice in large infrastructure systems. What is the operational contribution here is applying it specifically to enterprise database provisioning policy, tested and enforced before a profile is ever approved for use. A profile is not approved until it demonstrates, for every mandatory enterprise control, one of three states: implemented and independently verifiable, explicitly declared unsupported by that platform, or routed through an approved exception process. That three-way test is what keeps a profile from looking complete while silently omitting a required control.

Enterprise Policy Objective	Platform A Mechanism	Platform B Mechanism	Verification State
Encryption at rest	Native key-management integration	Platform-managed encryption service	Implemented, verified
Network isolation	Private subnet + security group policy	Virtual network + firewall rule set	Implemented, verified
Backup and retention	Automated snapshot schedule	Platform-native backup policy	Implemented, verified

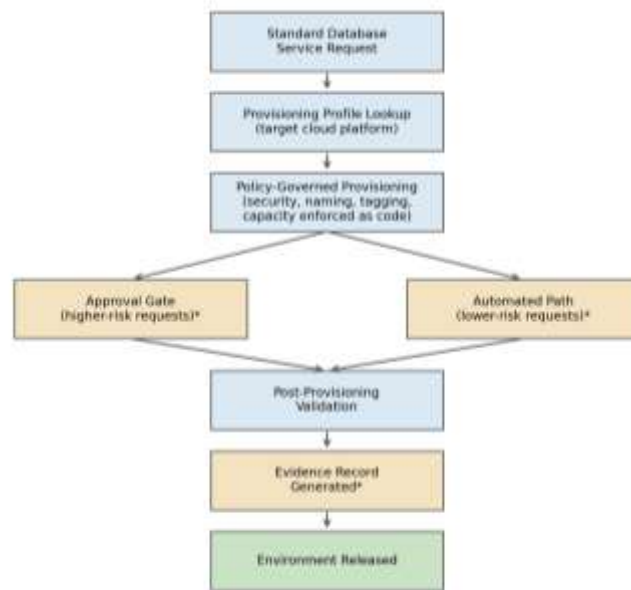
Monitoring and logging hooks	Native metrics + log export	Platform monitoring integration	Implemented, verified
Continuous drift detection	Scheduled re-scan (proposed)	Scheduled re-scan (proposed)	Generalized architecture
Exception / waiver governance	Time-bound waiver record (proposed)	Time-bound waiver record (proposed)	Generalized architecture

Table 1. Provisioning profile - illustrative control mapping across cloud platforms.

3.2 Provisioning architecture

A standard database-service request captures workload type, expected scale, criticality tier, and target platform. The framework translates that request into the specific sequence of infrastructure and configuration steps the target platform's provisioning profile defines, replacing the manual runbook an engineer previously worked through platform by platform. The request format itself does not vary by target platform - a requester specifies their workload's requirements, not the mechanics of any particular cloud database service. Figure 1 traces this flow end to end, including the generalized architecture extensions discussed in Section 3.4.

Naming, tagging, and capacity get enforced as policy, not left to convention. An environment that fails to satisfy every mandatory control is not marked complete and is not released for use, even in cases where some underlying cloud resources have already been created. That is a release-gate guarantee, and it is worth being precise about what it does not claim: partial failures, external changes made outside the framework, defects in a profile's own implementation, and approved exceptions can all still leave the framework having created resources that are not yet compliant. Framed this way, governance shifts from something applied after an environment exists into a property built into how the environment is allowed to come into existence at all.



* Generalized architecture extension (Section 3.4) -- not uniformly deployed and measured.

Figure 1. Policy-driven provisioning architecture flow.

3.3 Security, resilience, monitoring, and compliance controls

Security policy - encryption requirements, access-control rules, network configuration - gets expressed as code rather than as documentation each engineer interprets independently at request time. Every environment provisioned under an approved profile satisfies the same approved objectives for encryption, key management, access control, and network isolation; the specific mechanism achieving each objective differs by platform, exactly as the profile defines. Reviewing the policy definition once, rather than reviewing each individually-provisioned environment against the same requirements over and over, substantially reduces the volume of repetitive security review work. It does not eliminate review entirely - a profile can be syntactically complete

and still implement its assigned policy incorrectly, so each profile's own implementation, each policy change, and the resulting environments themselves still need independent verification.

Backup configuration, monitoring, and logging get applied by default at provisioning time and validated as mandatory steps before an environment is released, rather than left as follow-up tasks an engineer may or may not get back to. Restore testing and full disaster-recovery validation stay explicitly out of scope, consistent with the scoping boundary stated in Section 1.1. Treating these controls as part of provisioning itself matters most exactly when they are easiest to skip - under deadline pressure, once the immediate functional request has been satisfied and the non-functional pieces are the ones most likely to get deferred if they are not structurally built into what provisioning means.

3.4 Approval, evidence, failure handling, and drift management (generalized architecture)

The mechanisms in this subsection describe a generalized architecture extending the implemented provisioning core. They are flagged here explicitly as such, consistent with Section 1.1's scoping statement, and are not all claimed as uniformly deployed and measured.

Higher-risk requests - production-tier environments, requests above a defined capacity threshold, or requests carrying a sensitive data classification - would route through an explicit approval gate before provisioning proceeds; lower-risk requests could proceed automatically. Post-provisioning validation would independently confirm the resulting environment matches policy, rather than simply trusting that the provisioning process executed correctly because it completed without error. The architecture would generate an evidence record identifying the policy version applied, the specific controls verified, and the approvals tied back to the originating request.

Failure handling gets treated as a first-class design concern rather than an afterthought, because provisioning spans many resources and steps and can fail partway through - and a partially-provisioned environment is frequently worse than none at all, since it can look usable while actually missing backup configuration or network isolation underneath. Rollback in infrastructure automation is, in practice, usually compensation rather than true transactional undo: a database that has already accepted connections, or already produced persistent diagnostic state, cannot always be deleted cleanly without side effects of its own. A framework that treats failure as first-class chooses among a small set of responses - retry an idempotent step, resume from the last successfully completed step, execute a compensating action to undo one specific side effect, or quarantine a partially-created resource for manual review when automated cleanup itself carries risk.

Drift detection is treated as a separate problem from provisioning-time correctness, not a continuation of it. Environments would be rescanned on a regular cadence against the same provisioning policy that governed their creation, with detected drift classified by severity - a missing tag is not the same finding as a disabled encryption setting - and routed accordingly, as summarized in Table 2. Some drift categories would be auto-corrected; others, particularly anything whose automated correction risks downtime, require approval before remediation proceeds. A workable exception-governance model needs waivable and non-waivable control categories decided in advance, a named approver for anything waivable, a required justification, an expiration date rather than a permanent waiver, and re-validation whenever the underlying policy itself changes.

Drift Category	Example	Remediation Routing	Waivable?
Low severity	Missing or inconsistent resource tag	Auto-corrected	Yes, time-bound
Medium severity	Monitoring hook disabled	Auto-corrected, logged	Yes, time-bound
High severity	Network isolation rule loosened	Approval required before remediation	Named-approver only
Critical severity	Encryption at rest disabled	Immediate escalation, approval required	Non-waivable

Table 2. Drift classification and remediation routing.

3.5 Evaluation method

The applied evaluation compares the prior ticket-driven provisioning process against the automated workflow using two practical indicators: provisioning turnaround time and repetitive manual engineering effort. This

comparison is explicitly not a controlled statistical study. What evidence is available supports order-of-magnitude, generalized claims about direction and rough scale of improvement - not precise, statistically validated figures - and this article states that limitation directly rather than presenting practitioner observation as something more rigorous than it actually is. A methodologically complete account would specify the measurement period, the approximate volume of requests reviewed, the operational definitions used for the start and completion of a provisioning request, and whether the manual-effort estimate derives from workflow-system records, ticket history, or practitioner assessment. Those specifics were not available for this intake and should be verified against workflow and ticket records before the underlying figures are treated as final for publication.

4. Results

Operational experience under the automated framework indicated that provisioning requests once measured in days could complete in minutes once platform-specific provisioning steps were automated end to end, illustrated in order-of-magnitude terms in Figure 2. Alongside the turnaround change, engineers reported a substantial reduction in repetitive manual engineering work - the platform-specific runbook execution that previously consumed disproportionate time relative to the actual decision content of a provisioning request.

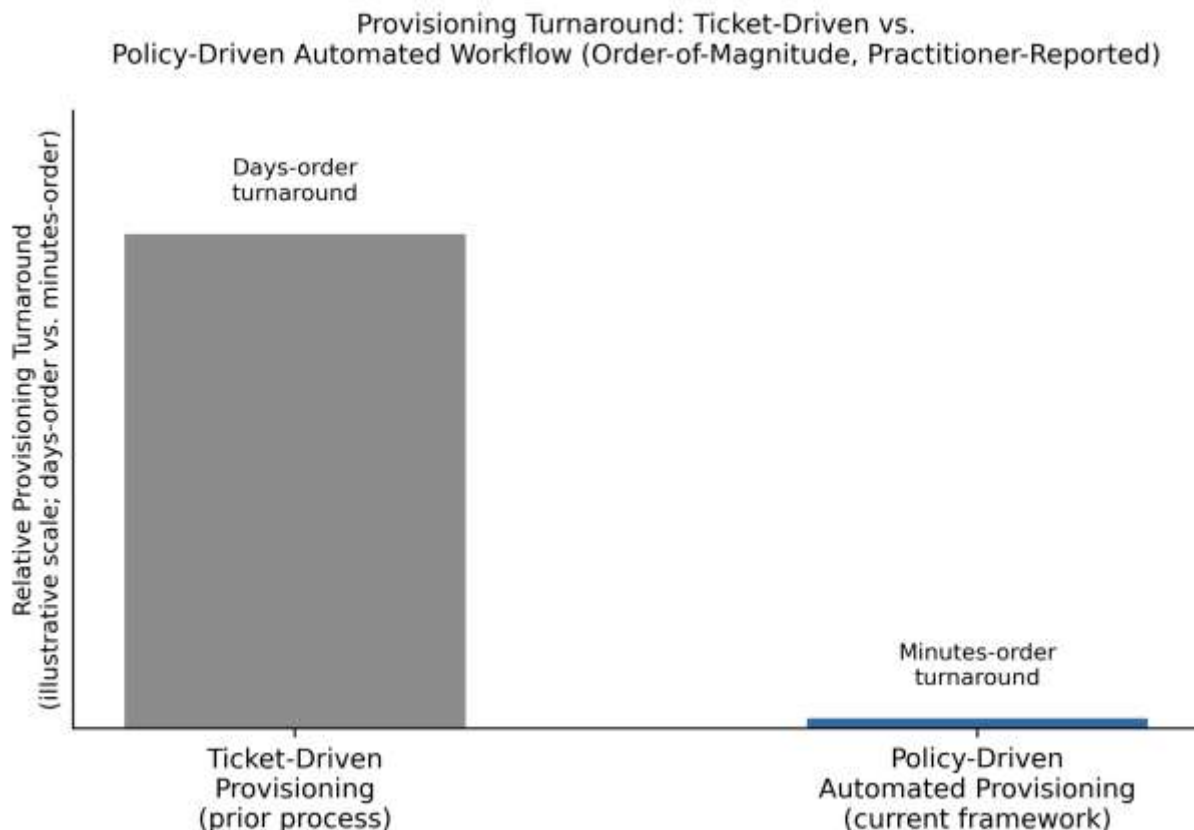


Figure 2. Provisioning turnaround: ticket-driven vs. policy-driven automated workflow (order-of-magnitude, practitioner-reported).

A related but analytically distinct data point concerns adoption of the underlying automation foundation, rather than database-environment throughput specifically: the associated AWS-native self-service deployment platform is reported to support more than 150 developers across thousands of deployment operations. This figure should not be read as a count of database environments provisioned per day. It is evidence of organizational adoption of the automation foundation the provisioning profiles run on, and the two should not be conflated with each other.

Both categories of practitioner-reported outcome above require verification against available workflow and ticket-system records before they can be presented in a final publication as anything beyond order-of-

magnitude, practitioner-reported estimates. That verification had not been completed at the time of this intake. Stating that plainly is itself part of the result here, not a caveat to be minimized.

5. Discussion

5.1 What policy-driven actually buys an enterprise

The value of this architecture is not that it automates faster typing. It is that the same enterprise requirement gets enforced identically no matter which engineer, which team, or which cloud platform handles a given request - a property manual, ticket-driven provisioning structurally cannot guarantee, no matter how well-documented the runbooks happen to be. Extending to a new cloud platform becomes writing one new provisioning profile against an existing policy interface, rather than re-deriving security, backup, and monitoring policy from first principles all over again for every new platform the enterprise adopts.

5.2 Lessons learned: exceptions are the real threat surface

The framework's practical value scales inversely with how many exceptions engineers feel they need to make. Every manual override, every just this once exception granted under deadline pressure, erodes two things at once: the consistency guarantee the framework exists to provide, and - more slowly, more corrosively - engineers' trust that the automated path is actually the fastest and easiest path available to them. Keeping legitimate exceptions few, well-justified, time-bound, and visibly governed matters as much to the framework's long-term success as the automation itself does (Rekulapalli et al. 2026).

5.3 Reading the results honestly

The order-of-magnitude turnaround improvement reported here is consistent with, though not directly comparable to, the general finding that provisioning-tool choice carries measurable performance consequences (Regvart et al., 2026) - different scope, but the same underlying point that provisioning mechanics are not performance-neutral. The adoption evidence for the underlying deployment platform matters precisely because policy-as-code adoption research shows that security and compliance use cases dominate why organizations invest in this pattern in the first place (Foalem et al., 2026). That said, provisioning speed is not itself evidence of security or reliability. It measures how quickly a compliant environment can be created - and that claim depends entirely on the underlying policy being correct, which is a separate question from how quickly it gets enforced (Bhatnagar & Bonagiri, 2026).

6. Limitations and Generalizability

Three limitations deserve to be stated directly rather than buried in a closing paragraph.

First, results come from one enterprise context. Financial-services production requirements, regulatory posture, and organizational scale shaped both the policy model and the operational results reported here; a different industry or regulatory environment might see different costs and different benefits from the same architectural pattern.

Second, not every enterprise control normalizes cleanly across every cloud platform. Provider-level variation is not hypothetical - measured differences in trustworthiness posture, and in which security controls providers implement well versus poorly, are documented in the empirical literature (Akhtar et al., 2024; Verdet et al., 2025). Some requests will always need specialist human review rather than fully automated handling.

Third, and most important architecturally: automation increases the blast radius of a defective policy. A manual, ticket-driven process can also repeat the same mistaken instruction across many environments - that risk is not unique to automation. What changes is speed and scale. Provisioning speed, worth restating once more because it bears repeating specifically in a limitations section, is not itself evidence of security or reliability; it measures enforcement speed for a policy whose correctness is a separate, prior question, this framework does not answer on its own (Gaggar, 2026).

7. Conclusion

Separating enterprise policy from platform-specific implementation converts database provisioning governance from something enforced inconsistently, platform by platform and engineer by engineer, into a shared model instantiated through profiles that can be tested, versioned, and extended at bounded cost. The applied evidence available here - order-of-magnitude, practitioner-reported, and explicitly not a controlled statistical study - indicates meaningful reductions in both provisioning turnaround and repetitive manual effort, alongside real organizational adoption of the underlying automation foundation. The generalized extensions proposed - explicit approval gating, automated evidence generation, continuous drift detection as its own control loop, and structured exception governance - describe where this architectural pattern goes next, not a claim that every piece is already deployed and measured. The framework's durability depends less

on any single automation capability than on organizational discipline around exceptions: a policy-driven system stays policy-driven only for as long as overrides remain rare, justified, and visible.

References

1. Bhatnagar and V. R. K. Bonagiri, "Operational Excellence through Cloud-Native Distributed Systems," *International Journal of Computational and Experimental Science and Engineering*, vol. 12, no. 2, 2026. doi: 10.22399/ijcesen.5207.
2. Aghili, R., Li, H., & Khomh, F. (2023). Studying the characteristics of AIOps projects on GitHub. *Empirical Software Engineering*, 28, Article 143. <https://link.springer.com/article/10.1007/s10664-023-10382-z>
3. Akhtar, S. I., Rauf, A., Abbas, H., Amjad, M. F., & Batool, I. (2024). Compliance and feedback based model to measure cloud trustworthiness for hosting digital twins. *Journal of Cloud Computing*, 13, Article 132. <https://link.springer.com/article/10.1186/s13677-024-00690-0>
4. Angermeir, F., Moyón, F., Fischbach, J., & Mendez, D. (2024). Towards automated continuous security compliance. *ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. <https://dl.acm.org/doi/10.1145/3674805.3690748>
5. Bessghaier, N., Ouni, A., Sayagh, M., Chouchen, M., & Mkaouer, M. W. (2025). Towards understanding code review practices for infrastructure-as-code: An empirical study on OpenStack projects. *Empirical Software Engineering*, 30(4), Article 106. <https://link.springer.com/article/10.1007/s10664-025-10654-w>
6. Bharany, S., Kaur, K., Badotra, S., Rani, S., Kavita, Wozniak, M., Shafi, J., & Ijaz, M. F. (2022). Efficient middleware for the portability of PaaS services consuming applications among heterogeneous clouds. *Sensors*, 22(13), Article 5013. <https://www.mdpi.com/1424-8220/22/13/5013>
7. Bulut, M. F., & Hwang, J. (2021). NL2Vul: Natural language to standard vulnerability score for cloud security posture management. *IEEE International Conference on Cloud Computing (CLOUD 2021)*. <https://ieeexplore.ieee.org/document/9582205>
8. Daraghmi, E., Zhang, C.-P., & Yuan, S.-M. (2022). Enhancing Saga pattern for distributed transactions within a microservices architecture. *Applied Sciences*, 12(12), Article 6242. <https://www.mdpi.com/2076-3417/12/12/6242>
9. Falazi, G., Harzenetter, L., Képes, K., Leymann, F., Ntontos, E., Zdun, U., Breitenbücher, U., Becker, M., & Heldwein, E. (2023). Compliance management of IaC-based cloud deployments during runtime. *IEEE/ACM International Conference on Utility and Cloud Computing (UCC 2023)*. <https://dl.acm.org/doi/10.1145/3603166.3632135>
10. Foalem, P. L., Khomh, F., Da Silva, L., & Merlo, E. (2026). An empirical study of Policy-as-Code adoption in open-source software projects. *Journal of Systems and Software*. <https://arxiv.org/abs/2601.05555>
11. H. Gaggar, "MemGov: Policy-Governed Memory Lifecycle Management for Enterprise LLM Agents," *Journal of Intelligent Decision Making and Information Science*, vol. 3, no. 11s, pp. 2199–2214, 2026.
12. H. Gaggar, "Architecting scalable enterprise AI agent infrastructure for distributed intelligent systems," *International Journal of Advances in Signal and Image Sciences*, pp. 193–208, 2025. doi: 10.29284/da9w0117.
13. K. Rekulapalli, S. R. Kagga, V. Ayyagari, A. Akula, and R. Akula, "AI in HRM: Enhancing Workforce Competency and Organizational Effectiveness," *SSRN*, 2026. Available: <https://ssrn.com/abstract=6762998>.
14. Liu, F., Tong, J., Mao, J., Bohn, R. B., Messina, J. V., Badger, M. L., & Leaf, D. M. (2011). NIST cloud computing reference architecture (NIST Special Publication 500-292). National Institute of Standards and Technology. <https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication500-292.pdf>
15. Nasiri, R., Kumara, I., Tamburri, D. A., & van den Heuvel, W.-J. (2024). Towards a taxonomy of infrastructure as code misconfigurations: An Ansible study. In *SummerSOC 2024 (CCIS vol. 2221*, pp. 83–103). Springer. <https://www.summersoc.eu/wp-content/uploads/summersoc-2024-Nasiri-IaC.pdf>
16. Nogueira, L., & Resende, A. (2026). Reusing policy-as-code across CI/CD and Kubernetes admission control: An empirical assessment of governance consistency. *Computers*, 15(7), Article 453. <https://www.mdpi.com/2073-431X/15/7/453>
17. Prates, L., & Pereira, R. (2025). DevSecOps practices and tools. *International Journal of Information Security*, 24, Article 11. <https://link.springer.com/article/10.1007/s10207-024-00914-z>

18. Regvart, D., Vlahović, I., & Balković, M. (2026). A controlled comparative evaluation of infrastructure as code tools: Deployment performance and maintainability across Terraform, Pulumi, and AWS CloudFormation. *Applied Sciences*, 16(6), Article 2971. <https://www.mdpi.com/2076-3417/16/6/2971>
19. Romeo, F., Arena, L., Blefari, F. A., Pironti, F. A., Lupinacci, M., & Furfaro, A. (2026). ARPaCCino: An Agentic-RAG for Policy as Code Compliance. In *ADBIS 2025* (pp. 467–481). Springer. <https://arxiv.org/abs/2507.10584>
20. Tang, C., Kooburat, T., Venkatachalam, P., Chander, A., Wen, Z., Narayanan, A., Dowell, P., & Karl, R. (2015). Holistic configuration management at Facebook. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)* (pp. 328–343). <https://dl.acm.org/doi/10.1145/2815400.2815401>
21. Verdet, A., Hamdaqa, M., Da Silva, L., & Khomh, F. (2025). Assessing the adoption of security policies by developers in Terraform across different cloud providers. *Empirical Software Engineering*, 30(3), Article 74. <https://link.springer.com/article/10.1007/s10664-024-10610-0>
22. War, A., Diallo, A., Habib, A., Klein, J., & Bissyandé, T. F. (2025). Vulnerabilities in infrastructure as code: What, how many, and who? *Empirical Software Engineering*, 30, Article 120. <https://link.springer.com/article/10.1007/s10664-025-10672-8>
23. Xu, Q., Gao, Y., & Wei, J. (2024). An empirical study on Kubernetes operator bugs. In *ACM International Symposium on Software Testing and Analysis (ISSTA 2024)*. <https://dl.acm.org/doi/10.1145/3650212.3680396>
24. Xu, T., & Zhou, Y. (2015). Systems approaches to tackling configuration errors: A survey. *ACM Computing Surveys*, 47(4), Article 70. <https://dl.acm.org/doi/10.1145/2791577>