Computer sciences

UDC 004.056.5:004.414

Shkliar Kostiantyn

Master's Degree in Software Engineering

National Technical University of Ukraine

«Igor Sikorsky Kyiv Polytechnic Institute»

ORCID: 0009-0005-8321-0920

DOI: https://doi.org/10.25313/2520-2057-2026-6-12079

ACCESS MATRIX-AS-CODE: AUTOMATED GENERATION OF PERMISSION VALIDATION SCENARIOS FROM CODIFIED ACCESS SPECIFICATIONS IN ENTERPRISE CRM PLATFORMS

Summary. *Enterprise CRM platforms accumulate thousands of permission boundary definitions across roles, objects, and fields, yet the test suites that validate these boundaries are typically maintained through manual mappings that grow stale as configurations change. This paper introduces Access Matrix-as-Code (AMS), an approach in which the complete permission state of a Salesforce CRM deployment is extracted via Tooling API SOQL queries, codified as a versioned YAML specification, and used to automatically generate both positive and negative test scenarios for every role-object-field combination. A diff engine compares successive AMS snapshots to produce incremental scenario updates targeting only permission boundaries affected by each release. Evaluated across six release cycles in a financial services CRM environment with 27 roles and over 3,000 permission boundary definitions, the pipeline achieved 100% specification completeness, detected 14 unintended configuration drifts that the prior manual process missed, and reduced scenario generation time to under four minutes per release through delta-based updates. By converting*

permission validation from a periodic audit activity into a continuous, specification-driven engineering practice, the AMS approach eliminates the detection latency that allows unauthorized access configurations to persist between review cycles.

Key words: *access matrix-as-code, permission specification, automated test generation, CRM access control, Tooling API, policy-as-code, configuration drift detection.*

Introduction. Enterprise CRM deployments in regulated industries operate under a dual constraint: the permission model must be granular enough to enforce least-privilege access across dozens of roles, yet flexible enough to accommodate frequent changes driven by new business requirements, regulatory updates, and organizational restructuring. A Salesforce deployment serving a financial advisory network, for instance, may define 25 or more distinct roles, each governing access to hundreds of fields distributed across custom and standard objects. With 27 roles, 43 objects, and 1,840 field-level security entries in the deployment studied in this paper, the total number of individual access decisions exceeds 3,000 distinct permission boundaries. Verifying that these decisions are correctly implemented after every release cycle is a prerequisite for regulatory compliance under frameworks such as the Gramm-Leach-Bliley Act safeguards provisions and SOX internal control requirements.

Most organizations record the intended state of their permission configuration in spreadsheets, wiki pages, or Confluence matrices that an administrator populates at deployment time and updates when changes occur. These static documents diverge from the platform's actual configuration within weeks: a permission set adjustment made to resolve a user-reported access issue, a sandbox refresh that overwrites permission boundaries, or a deployment script that introduces unintended side effects can each create a gap between what the matrix declares and what the platform enforces. When such a gap involves

granting access that should be restricted, the organization operates with an undetected policy violation whose exposure window extends until the next scheduled audit. In the deployment studied here, audits were conducted quarterly, leaving a potential detection latency of up to 90 days per cycle.

Policy-as-Code (PaC) addresses an analogous drift problem in infrastructure management. HashiCorp Sentinel [1] evaluates Terraform plans against codified policy rules before infrastructure is provisioned, Open Policy Agent [3] applies its Rego language to Kubernetes admission control and cloud resource governance, and AWS CloudFormation Guard [2] validates infrastructure templates against declarative guard rules. Palo Alto Networks [15], SentinelOne [9], and Spacelift [10] document growing enterprise adoption of PaC for RBAC and IAM validation in cloud environments, reporting significant reductions in configuration violations when policies are codified and enforced within CI/CD pipelines rather than reviewed manually on a periodic schedule.

Despite this momentum, PaC has not been systematically adapted to CRM access control testing. Infrastructure PaC tools answer a compliance question: does this deployment configuration match the declared policy? They do not generate executable test scenarios that verify whether the deployed permission model behaves correctly at runtime when a user with a specific role attempts a specific operation on a specific field. Salesforce exposes its permission model through the Tooling API [7], a REST and SOQL-based interface that provides query access to PermissionSet, FieldPermissions, ObjectPermissions, and SetupEntityAccess sObjects. Baskar [13] and Salesforce Ben [12] have demonstrated extraction of profile and permission set metadata through these interfaces for administrative purposes. An open question remains: can the extracted permission state serve as a formal specification from which test scenarios are generated automatically, closing the loop between access policy definition and test execution?

Research on specification-driven test generation [8] has established the principle that each element of a formal model can produce corresponding test cases, achieving systematic coverage by construction. Whether this principle can be applied to CRM access matrices, and whether the generation can operate incrementally to scale with the magnitude of each change rather than the size of the overall system, has not been investigated.

This paper addresses these questions by formalizing and evaluating an Access Matrix-as-Code (AMS) pipeline. Three objectives guide the investigation: (1) to determine whether programmatic extraction of the CRM permission state can produce a specification complete enough to serve as the sole authoritative input for test generation; (2) to measure the accuracy of generated scenarios and the rate at which the pipeline detects unintended configuration drifts across six production release cycles; and (3) to evaluate whether diff-based incremental generation achieves sufficient efficiency for integration into a CI/CD pipeline, and whether the approach is portable across organizational boundaries, assessed through a secondary deployment at East Cloud Solutions, a Salesforce consulting practice founded by the author.

Literature Review. PaC frameworks express governance rules in domain-specific languages designed for policy authors rather than application developers. OPA's Rego [3] is a declarative query language that evaluates structured input documents against rule sets and returns allow/deny decisions; policies are organized as packages that can be unit-tested independently and composed into hierarchical bundles. Sentinel [1] follows a similar declarative model but embeds enforcement directly into Terraform's plan-apply lifecycle, so that a failing policy halts the provisioning operation before any resource is modified. CloudFormation Guard [2] takes a third path, defining rules as typed clauses that pattern-match against JSON and YAML template nodes. Check Point [14] documents how these frameworks integrate into CI/CD pipelines through pre-deployment gates that evaluate each infrastructure change against the active policy set and block non-

compliant commits. Critically, none of these languages includes constructs for emitting executable test scenarios, which confines PaC to compliance verification and prevents its direct application to runtime access validation.

Formal access control modeling provides the vocabulary for specifying permission boundaries. Ferraiolo et al. [4] defined the NIST RBAC standard with four components ranging from Core RBAC (user-role-permission mappings) to Dynamic Separation of Duty constraints. More directly relevant to test generation is NIST SP 800-53 Revision 5 [11], which operationalizes access control through 25 security requirements. AC-3 (Access Enforcement) mandates that the system enforce approved authorizations at every access point, and AC-6 (Least Privilege) mandates that each user receive only the minimum access necessary for their function. For an automated validation pipeline, these controls translate into a concrete criterion: every granted permission must succeed when exercised (AC-3 positive path), and every non-granted permission must be denied when attempted (AC-3 negative path combined with AC-6 verification).

Salesforce implements a layered permission resolution model whose complexity exceeds the flat role-permission mapping described in Core RBAC. Profiles [6] define baseline object and field access, system permissions, and login restrictions for every user assigned to them. Permission sets [5] grant additive access on top of the profile baseline, and permission set groups bundle multiple sets into a single assignable unit whose effective permissions are the union of all constituent sets. Within groups, muting permission sets selectively revoke specific permissions that would otherwise be inherited, introducing a subtractive layer absent from the standard RBAC model. Sharing rules govern record-level visibility by extending read or read/write access to records owned by users in one role to users in another role, operating independently of object- and field-level permissions. This multi-layered resolution means that the effective permission for a given user on a given field depends on the profile assignment, all active permission set assignments, the resolution order within any assigned permission

set groups, any muting overrides, and the applicable sharing rules. Extracting this composite state requires querying multiple metadata structures and merging their outputs according to the platform's resolution precedence.

Salesforce's Tooling API [7] provides query access to the metadata objects that encode each layer of the permission model. FieldPermissions records store boolean read and edit flags per field per permission set; ObjectPermissions records store CRUD flags (create, read, update, delete) plus ViewAllRecords and ModifyAllRecords per object per permission set; SetupEntityAccess records store access grants to connected apps, Apex classes, and Visualforce pages per permission set. Each record includes the parent PermissionSet identifier, enabling the extraction layer to reconstruct the full permission tree by joining query results across these four sObjects. Baskar [13] demonstrated a package.xml retrieval technique that pulls profile and permission set metadata as XML bundles, but this approach returns the raw deployment representation rather than a queryable dataset. Salesforce Ben [12] documented Metadata API and Tooling API usage for administrative bulk edits, confirming that the Tooling API's SOQL interface offers more granular query control than the XML-based retrieve operation used for deployment packages.

Utting, Pretschner, and Legeard [8] published a taxonomy that classifies model-based testing approaches along three dimensions: model notation, test generation algorithm, and coverage criterion. Specification-driven generation, the category most relevant to access matrix validation, produces one or more test cases for every element of the formal model, achieving coverage by construction rather than by sampling. When applied to a permission matrix, this means each role-object-field entry with a defined access level yields a scenario pair: a positive scenario exercising the granted path and a negative scenario exercising the denied path. Prior applications of specification-driven generation have focused on state machine models and protocol specifications; no published work applies this

technique to access control matrices extracted from a CRM platform's metadata layer.

Table 1 positions the proposed AMS approach against two established alternatives for managing permission validation.

Table 1

Comparison of permission validation approaches

Characteristic	Manual access matrix	Infrastructure PaC	AMS (proposed)
Spec format	Spreadsheet / wiki	Rego / Sentinel rules	Versioned YAML
Scope	CRM access control	Infrastructure / cloud IAM	CRM access control
Update trigger	Manual edit	Policy code by operators	Auto-extracted per release
Test generation	Manual authoring	None (verdict only)	Automated pos. + neg.
Drift detection	Periodic manual audit	Pre-deployment gate	Diff snapshot per release
Incremental	Full re-review	Full re-evaluation	Delta for changed only
Feedback loop	None	None	Failures trigger spec fix

Source: synthesized by the author based on [1; 2; 3; 5; 7; 8]

Manual matrices depend on human maintenance and produce tests only through manual authoring, creating a bottleneck proportional to permission complexity. Infrastructure PaC tools automate compliance evaluation but confine their output to binary verdicts without producing executable scenarios, and target cloud resource configurations rather than CRM-specific permission models with layered resolution logic. No peer-reviewed studies address automated scenario generation from CRM permission metadata; the closest related work remains the model-based testing literature [8], which has not been applied to access control matrices extracted from a platform's metadata layer. Neither existing approach generates test scenarios from the platform's own permission state or supports incremental updates scoped to changed entries.

Materials and Methods. Evaluation was conducted within a Salesforce-based CRM deployment at a financial services organization managing client

assets in excess of $130 billion. Outcomes are compared across six consecutive release cycles during which the AMS pipeline was operational, with baseline comparison drawn from six preceding cycles during which permission validation relied on manually maintained matrices and manually authored scenarios.

Designed by the author, the AMS pipeline operates as a three-phase process: Extract, Codify, and Generate. In the Extract phase, a scheduled job authenticates via OAuth 2.0 client credentials flow and executes four Tooling API SOQL queries against distinct sObjects: PermissionSet (retrieving all active sets, including those embedded within profiles, filtered by IsActive = true), FieldPermissions (retrieving per-field read and edit grants for each set), ObjectPermissions (retrieving per-object CRUD, ViewAllRecords, and ModifyAllRecords grants for each set), and SetupEntityAccess (retrieving entity-level access to connected apps, Apex classes, and Visualforce pages for each set). Parent PermissionSet identifiers in each result set enable the pipeline to assemble the complete access hierarchy through cross-sObject joins. Dormant permission sets with zero user assignments are excluded to avoid generating scenarios for unreachable configurations.

In the Codify phase, normalized extraction output is transformed into an Access Matrix Specification (AMS) document formatted in YAML. Each entry follows a hierarchical key structure: role identifier, object API name, field API name (for field-level entries) or the literal "object" (for object-level entries), and a set of boolean access flags (readable, creatable, editable, deletable, viewAllRecords, modifyAllRecords). Sharing rule entries record the sharing model (private, readOnly, readWrite) per object per role pair. Permission set group resolution is handled by merging the permissions granted by each member set and then subtracting any flags revoked by muting sets assigned to the group, replicating the platform's own resolution precedence within the specification. Each AMS document is committed to version control with a release-cycle timestamp. Schema design decisions for platform-specific constructs such as

muting set resolution order and record-level sharing propagation to child objects were informed by the author's eight Salesforce certifications spanning administration, development, and architecture domains.

In the Generate phase, the generation engine iterates over each AMS entry and emits one scenario per entry. Entries where access is granted produce a positive scenario that authenticates as a user assigned to the corresponding role and asserts successful execution of the permitted operation. Entries where access is denied produce a negative scenario that authenticates as a user assigned to the corresponding role and asserts that the operation is blocked by the platform. When executing after the initial baseline cycle, a diff engine compares the current AMS snapshot with its predecessor and identifies entries whose access flags differ. Only changed entries generate new scenarios; unchanged entries retain existing scenarios without regeneration. Each changed entry is further classified as an intentional modification (matching an approved change request in the release manifest) or an unintended drift (no corresponding change request found).

Scenario execution is handled by the organization's existing test automation infrastructure, which authenticates as each role through dedicated test user accounts and performs CRUD operations against the target objects and fields via the Salesforce UI layer. A full suite of 3,087 scenarios completes execution in approximately 4.5 hours when distributed across parallel browser sessions, a duration compatible with overnight CI pipeline scheduling. Because the execution framework is external to the AMS pipeline, the pipeline's output is a structured scenario manifest (role, object, field, operation, expected outcome) that the framework consumes without modification.

Four metrics evaluate pipeline effectiveness. Specification completeness measures the percentage of active permission configurations (field-level, object-level, sharing rule) captured in the AMS document relative to the total returned by Tooling API queries. Accuracy of generated scenarios is measured as the percentage whose expected outcome (access granted or denied) matches the

actual platform behavior, verified through a full-suite execution after each extraction cycle. Configuration drift detection rate records the number of unintended permission changes identified through AMS snapshot comparison per release cycle. Incremental generation efficiency compares the wall-clock time for diff-based delta generation against full-suite regeneration.

Results

1. Specification Scope and Scenario Distribution

Table 2 reports the AMS dimensions for the final release cycle, representative of the stable deployment scope observed throughout the evaluation period.

Table 2

Access Matrix Specification dimensions and generated scenario counts (final release cycle)

Dimension	Count
Active roles (profiles + permission sets)	27
Custom and standard objects	43
Field-level security entries	1,840
Object-level permission entries	1,161
Sharing rule entries	86
Total AMS entries	3,087
Positive scenarios (access granted)	2,214 (71.7%)
Negative scenarios (access denied)	873 (28.3%)

Source: compiled by the author based on Tooling API extraction logs

Among the 873 negative scenarios, field-level denials constituted 612 (70.1%), object-level CRUD restrictions accounted for 198 (22.7%), and sharing rule restrictions comprised 63 (7.2%). Specification completeness across all six cycles was 100%.

2. Generation Accuracy and Refinement

Generation accuracy was evaluated by executing the full scenario suite after each extraction and comparing outcomes against AMS-predicted access levels. Across the six cycles, 18,522 total scenario executions (3,087 scenarios per cycle) produced 18,491 correct outcomes, yielding an aggregate accuracy of 99.8%.

All 31 mismatches concentrated in two cycles and traced to two distinct root causes. During cycle 2, 19 positive scenarios failed because a permission set group containing a muting set produced a denial that the AMS document did not predict. Investigation revealed that the Codify phase computed the group's effective permissions by applying muting sets after all granting sets, whereas the platform applied them in the order specified by PermissionSetGroupComponent records. Correcting the resolution order in the Codify logic eliminated these mismatches in subsequent cycles. During cycle 4, 12 negative scenarios unexpectedly passed because a sharing rule modification propagated read access to child objects through a master-detail relationship that the top-level sharing rule query did not capture. Adding a recursive traversal for master-detail child objects to the Extract phase resolved this class of errors. Cycles 5 and 6 each achieved 100% generation accuracy with no further refinements required.

3. Configuration Drift Detection

Under the prior manual audit regime, three unintended permission changes were detected across six baseline cycles, with an average discovery latency of 47 days. AMS snapshot comparison, executing at each release cycle boundary (every 3 to 5 days), produced a substantially different detection profile.

Figure 1 presents the configuration drift detection record across the six AMS-monitored release cycles.

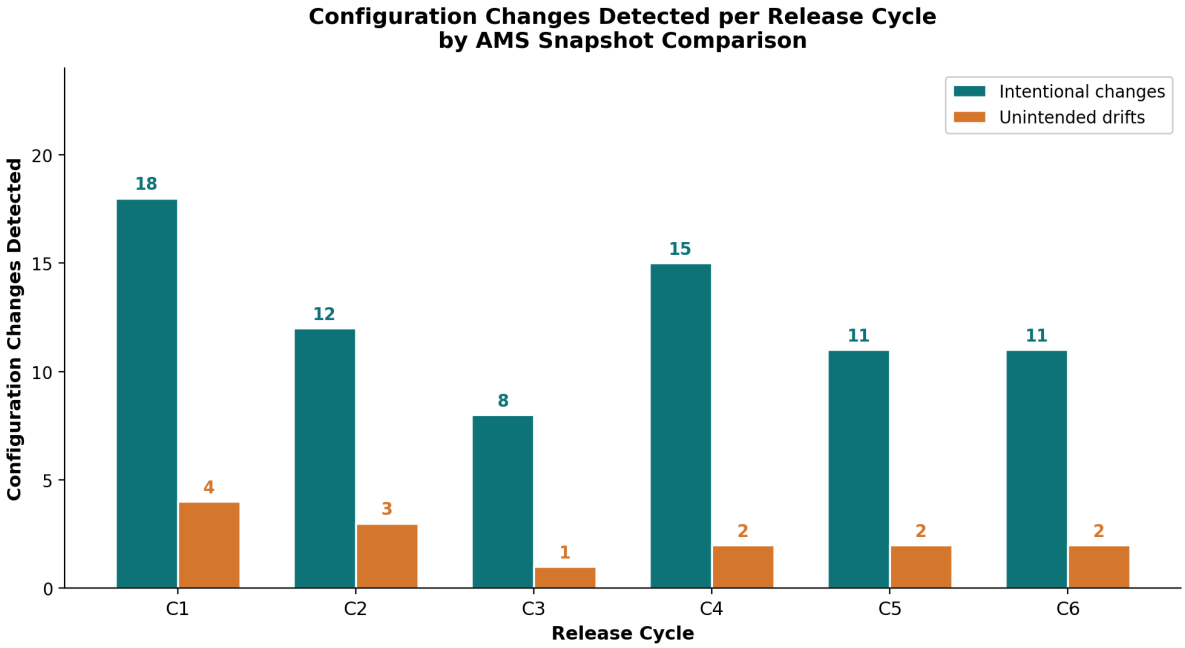

Fig. 1. Configuration changes detected per release cycle: intentional modifications versus unintended drifts

Source: compiled by the author

Across the six cycles, the diff engine flagged 89 total configuration changes: 75 intentional modifications matching approved change requests and 14 unintended drifts. Cycle 1 exhibited the highest drift count (4), attributable to residual configuration artifacts from the preceding sandbox refresh that had not been reconciled with the production change manifest. Nine of the 14 drifts involved field-level security grants that expanded access beyond the intended boundary, a pattern consistent with the administrative practice of granting additional field access as a quick fix for user-reported access errors without updating the formal change record. Four drifts involved object-level permission changes introduced by deployment scripts that modified permission sets as side effects of unrelated feature deployments. One drift involved a sharing rule modification triggered by a role hierarchy adjustment. All 14 were corrected at the release boundary before the next production deployment.

4. Incremental Generation Efficiency

Full-suite generation from a complete AMS snapshot required an average of 22.2 minutes across the six cycles for the 3,087-entry primary deployment; the East Cloud Solutions deployment (described in Section 4.5) completed full generation of its 640 entries in 7 minutes, consistent with generation time being proportional to entry count. Cycle 1 required full generation (24 minutes) because no prior snapshot existed for comparison. For cycles 2 through 6, the diff engine identified an average of 52 changed entries per cycle (range: 31 in cycle 3 to 82 in cycle 4), representing 1.7% of total AMS entries.

Figure 2 compares full-suite and incremental generation times.

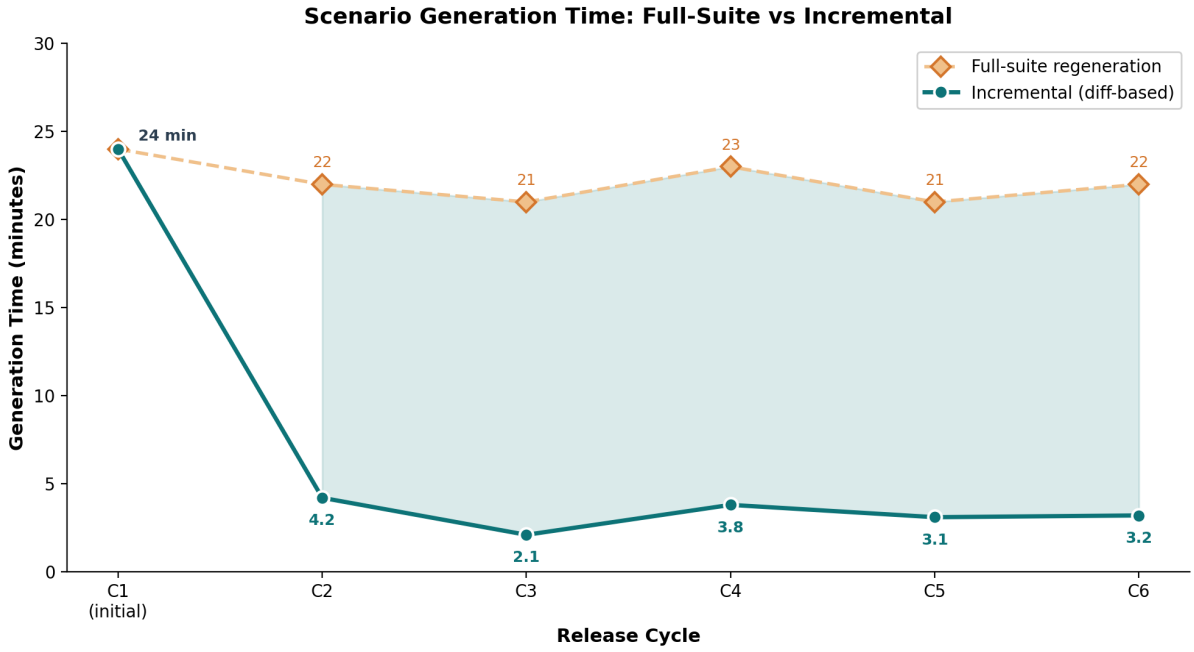

Fig. 2. Scenario generation time per release cycle: full-suite versus incremental

Source: compiled by the author

Incremental generation averaged 3.3 minutes for cycles 2 through 6. Correlation between generation time and the number of changed entries was strong (Pearson r = 0.94), confirming that the diff-based approach scales with change magnitude rather than system size. At 3.3 minutes per cycle, incremental generation fits within a single CI/CD pipeline stage without blocking downstream deployment steps.

5. Cross-Organizational Portability

To assess whether the pipeline generalizes beyond the primary deployment, the author applied it to a secondary Salesforce environment maintained by East Cloud Solutions, a consulting practice the author established to deliver Salesforce implementation services. This environment supports 12 roles across 18 objects with 640 field-level security entries, a structurally distinct deployment with different custom objects, permission set hierarchies, and sharing models. Extraction, codification, and generation completed successfully without modification to the pipeline logic; only Tooling API connection parameters and the AMS schema's object-field enumeration required reconfiguration. Full-suite generation completed in 7 minutes, and incremental generation averaged 1.2 minutes per cycle. Specification completeness was 100%. Measured accuracy was 99.5% in the first cycle (3 mismatches out of 640 scenarios, caused by the same permission set group resolution issue identified in the primary deployment), rising to 100% after applying the resolution order correction already developed during cycle 2 of the primary evaluation.

6. Consolidated Outcomes

Table 3 summarizes the principal evaluation metrics.

Table 3

Summary of evaluation metrics

Metric	Baseline (manual)	AMS pipeline
Spec completeness	Not measured	100% (all 6 cycles)
Generation accuracy	N/A	99.8%; 100% cycles 5–6
Unintended drifts	3 (quarterly audit)	14 (per-release)
Avg. detection latency	47 days	0 production-exposed days
Full generation time	N/A	22.2 min avg
Incremental gen. time	N/A	3.3 min avg (cycles 2–6)
Scenarios maintained	~180 (manual)	3,087 (auto-generated)
East Cloud Solutions	N/A	100%; 7 min full gen.

Source: compiled by the author based on pipeline execution logs and change management records

Discussion. Both accuracy drops observed during the evaluation originated not from flaws in the generation logic but from incomplete modeling of the platform's permission resolution behavior in the extraction and codification layers. Once corrected, the same refinements applied without modification to the secondary deployment, where the identical muting set issue surfaced independently. This convergence suggests that the set of platform-specific edge cases is finite and discoverable through progressive operational exposure: each edge case, once encoded into the pipeline, is permanently resolved across all deployments that share the same Tooling API schema. For organizations considering adoption, this means the initial cycles carry a higher correction burden, but the pipeline stabilizes as the edge case inventory approaches completeness.

A defining architectural property of the AMS pipeline is that it creates a bidirectional validation loop between the specification and the platform. In the forward direction, the specification produces scenarios that exercise each declared permission boundary against the live platform. In the reverse direction, scenario failures expose inaccuracies in the specification itself, either because the extraction queries missed a platform behavior (as with muting set resolution order) or because the codification logic applied an incorrect resolution precedence (as with child object sharing propagation). Each reverse-direction correction permanently improves the specification's fidelity to the platform, so that subsequent forward-direction validations operate against a progressively more accurate model. Infrastructure PaC tools lack this reverse path: when a Sentinel policy evaluation returns a compliance violation, the operator must determine independently whether the violation reflects a genuine infrastructure misconfiguration or a stale policy definition. In the AMS pipeline, scenario failures carry sufficient diagnostic detail (which role, which object, which field, which operation, expected versus actual outcome) to distinguish platform drift from specification drift without additional investigation.

Drift patterns reveal an asymmetry in how administrators interact with the permission model under time pressure. Nine of the fourteen unintended changes expanded access rather than restricted it, consistent with a response pattern in which granting additional field visibility is the fastest way to address an end-user complaint about missing data. Restricting access, by contrast, tends to be deferred because it risks disrupting existing workflows. From a security standpoint, this asymmetry is consequential: over-granting is precisely the class of drift that creates regulatory exposure, yet it is the class least likely to be noticed by users or reported through support channels. Automated snapshot comparison detects expansions and restrictions with equal sensitivity, neutralizing the asymmetry that manual audit processes inherit from the administrative behavior that produced the drift.

Beyond detection sensitivity, the AMS pipeline differs from manual auditing in two structural dimensions that compound to explain the gap between 3 and 14 detected drifts across equivalent six-cycle periods. First, audit frequency: manual audits occurred quarterly, leaving up to 90 days between observations during which multiple drifts could be introduced, compounded, and even reversed before the next review. Per-release snapshot comparison reduces this window to the length of a single release cycle (3 to 5 days), capturing each individual change event before it interacts with subsequent modifications. Second, coverage scope: a manual auditor reviewing a matrix of 3,087 permission entries under time constraints inevitably samples a subset of roles and objects, prioritizing areas flagged by user complaints or recent change requests. Automated comparison evaluates every entry in the specification exhaustively, with no sampling bias toward recently active areas. When frequency and coverage improve simultaneously, the combined effect exceeds what either factor would produce independently: high-frequency exhaustive comparison surfaces drifts that low-frequency sampled audits are structurally incapable of detecting.

Incremental generation's dependence on the volume of changed entries rather than the total specification size carries implications for organizations operating at scales larger than the deployment studied here. A deployment with 10,000 AMS entries experiencing the same 1.7% change rate per cycle would produce approximately 170 changed entries, yielding an estimated incremental generation time of roughly 11 minutes based on the observed rate of approximately 16 entries per minute. Full-suite regeneration for the same deployment would require approximately 73 minutes, making incremental generation the only viable option within a CI/CD pipeline constrained to sub-hour stage budgets. For multi-org enterprises managing parallel Salesforce instances across business units, the cumulative advantage compounds with each additional org.

Several constraints limit the scope of these findings. Tooling API SOQL queries are specific to Salesforce; other CRM platforms (Microsoft Dynamics 365, ServiceNow) expose permission metadata through different interfaces, and adapting the Extract phase to each platform would require rewriting the query layer. A separate boundary is that the pipeline produces scenario definitions (role, object, field, operation, expected outcome) rather than fully executable test scripts; the execution framework that authenticates each role and performs the access assertion was already present in the organization's automation infrastructure and is external to the AMS pipeline itself. Independent replication at unrelated enterprises would strengthen evidence for general portability beyond the two related deployments evaluated here. Although the East Cloud Solutions deployment provides a secondary data point, both environments share the same Tooling API schema and were configured by the same author, leaving external validity dependent on future replication by independent teams operating in different industry sectors.

Future research could apply machine learning to AMS snapshot histories to predict which permission configurations are most likely to drift, enabling

proactive test prioritization before changes occur. Extending the YAML specification to capture time-bound access constraints (permissions valid only during business hours or within approval windows) would address a category of access requirements that the current boolean-flag model cannot represent. Cross-platform portability, in which a unified AMS schema generates scenarios for Salesforce, Dynamics 365, and ServiceNow through platform-specific extraction adapters, would test whether the approach generalizes beyond a single vendor's metadata architecture.

Conclusions. Programmatic extraction via Tooling API captured the full active permission state without manual input, producing a specification that remained synchronized with the platform across every measured cycle. Where the specification initially fell short, in capturing how muting sets interact with group resolution sequences and how sharing rules propagate through master-detail hierarchies, the pipeline's own test results exposed the gaps and guided targeted corrections, after which no further discrepancies arose. This self-correcting property distinguishes the AMS approach from static documentation, which can harbor silent inaccuracies indefinitely because no automated mechanism exists to surface them. Once stabilized, the specification became the sole input from which both positive and negative scenarios were derived deterministically, eliminating the manual scenario authoring bottleneck that limited the prior process to approximately 180 hand-written scenarios covering a fraction of the deployment's 3,087 permission boundaries.

Drift detection delivered the most operationally consequential capability. Fourteen unintended permission changes surfaced automatically at the release boundary across six cycles, compared to three detected during an equivalent period of quarterly manual audits. Access expansions accounted for most detected drifts, precisely the category that creates regulatory exposure yet generates no user complaints that would trigger manual investigation. Scenario accuracy stabilized at 100% after two targeted extraction refinements, confirming that the

specification-to-scenario mapping produces reliable validation outcomes once the extraction layer accounts for platform-specific permission resolution edge cases.

Diff-based incremental generation reduced pipeline execution time from over 22 minutes (full-suite) to 3.3 minutes (delta), making specification-driven validation practical as a recurring CI/CD pipeline stage rather than an occasional batch operation. Execution time scales with the number of permission boundaries that changed between releases, not with the overall specification size, a property that becomes increasingly important as deployments grow beyond the scale evaluated here. Successful replication at East Cloud Solutions, a structurally distinct Salesforce deployment with different role hierarchies and object models, confirmed that the pipeline's core logic transfers across organizational boundaries when the underlying Tooling API schema is shared, and that extraction refinements developed in one deployment apply directly to others without requiring independent rediscovery of the same edge cases.

References

1. HashiCorp. (2024). *Policy as code, explained*. HashiCorp Blog. URL: https://www.hashicorp.com/en/blog/policy-as-code-explained (access date: 04.05.2026).

2. Amazon Web Services. (2023). *Governance at scale: Enforce permissions and compliance by using policy as code*. AWS Security Blog. URL: https://aws.amazon.com/blogs/security/governance-at-scale-enforce-permissions-and-compliance-by-using-policy-as-code/ (access date: 05.05.2026).

3. Open Policy Agent. (2024). *OPA documentation*. URL: https://www.openpolicyagent.org/docs/latest/ (access date: 05.05.2026).

4. Ferraiolo, D. F., Sandhu, R. S., Gavrila, S., Kuhn, D. R., & Chandramouli, R. (2001). Proposed NIST standard for role-based access control. *ACM Transactions on Information and System Security, 4*(3), 224–274. https://doi.org/10.1145/501978.501980

5. Salesforce Developers. (2024). *PermissionSet – Metadata API developer guide*. Salesforce. URL: https://developer.salesforce.com/docs/atlas.en-us.api_meta.meta/api_meta/meta_permissionset.htm (access date: 06.05.2026).

6. Salesforce Developers. (2024). *Profile – Metadata API developer guide*. Salesforce. URL: https://developer.salesforce.com/docs/atlas.en-us.api_meta.meta/api_meta/meta_profile.htm (access date: 06.05.2026).

7. Salesforce Developers. (2024). *Tooling API overview*. Salesforce. URL: https://developer.salesforce.com/docs/atlas.en-us.api_tooling.meta/api_tooling/intro_api_tooling.htm (access date: 07.05.2026).

8. Utting, M., Pretschner, A., & Legeard, B. (2012). A taxonomy of model-based testing approaches. *Software Testing, Verification and Reliability, 22*(5), 297–312. https://doi.org/10.1002/stvr.456

9. SentinelOne. (2025). *What is policy as code (PaC)? Examples, benefits & working*. URL: https://www.sentinelone.com/cybersecurity-101/cloud-security/policy-as-code/ (access date: 08.05.2026).

10. Spacelift. (2024). *What is policy as code (PaC) & how do you implement it?* URL: https://spacelift.io/blog/what-is-policy-as-code (access date: 08.05.2026).

11. National Institute of Standards and Technology. (2020). *Security and privacy controls for information systems and organizations (NIST SP 800-53 Rev. 5).* https://doi.org/10.6028/NIST.SP.800-53r5

12. Salesforce Ben. (2024). *Salesforce metadata API: Your complete guide*. URL: https://www.salesforceben.com/salesforce-metadata-api-your-complete-guide/ (access date: 10.05.2026).

13. Baskar, H. (2026). *Extract complete profile & permission set metadata in Salesforce*. Medium. URL: https://medium.com/@hari626007/extract-complete-profile-permission-set-metadata-in-salesforce-347225b2c50a (access date: 12.05.2026).

14. Check Point. (2025). *What is policy-as-code?* Check Point Cyber Hub. URL: https://www.checkpoint.com/cyber-hub/cloud-security/what-is-code-security/what-is-policy-as-code/ (access date: 13.05.2026).

15. Palo Alto Networks. (2025). *What is policy-as-code?* Cyberpedia. URL: https://www.paloaltonetworks.com/cyberpedia/what-is-policy-as-code (access date: 14.05.2026).