



A Metadata-Driven Deployment Risk Scoring Framework for Enterprise Salesforce CI/CD Pipelines

Murali Mohan Reddy Seelam^{1,*} Vyshnavi Thanneeru² Amrut Pagidipally³

¹ Department of IT, Cisco Systems, McKinney, Texas 75070, USA

² Department of Information Technology, Fidelity Investments, McKinney, Texas 75070, USA

³ Department of IT, Charles Schwab, Austin, Texas 78701, USA

* Corresponding author. Email: murali.reddy60@gmail.com

Email: vyshnavi.thanneeru@gmail.com; Amrut.pagidipally@gmail.com

Abstract. When a Salesforce deployment fails, the cleanup is rarely quick. Somebody has to roll it back by hand, the data fixes crawl along, and pinning down which single piece of a 90-component package knocked the org over can swallow the better part of a day. Underneath all of that sits a design choice most enterprise CI/CD pipelines never revisit: they run the same gate logic against every deployment, whether the package is trivial or structurally dangerous. Our answer is a tool that grades a package before it ever executes. It parses the manifest, talks to the target org over the Metadata API, and comes back with a risk score on a 0-100 scale, built from ten features, among them test coverage delta, dependency chain depth, whether the package mutates schema, and cyclomatic complexity. The whole thing finishes inside three seconds. From there the score decides what the pipeline does next: wave the deployment through, insist on a full org-wide Apex test run first, or freeze the package for architect review with a written rollback plan attached. We ran it live in the pipelines of six enterprise Salesforce organizations for half a year, across 847 deployments. The failure rate went from 23.4% down to 8.1%, and rollbacks came down by 66.3%. Because the Copado and Gearset integrations sit on hook interfaces those platforms already expose, none of this asked for new infrastructure.

Keywords: *Salesforce, Deployment Risk, CI/CD Pipelines, Metadata Analysis, Static Analysis, Risk Prediction.*

1. Introduction

Pushing to Salesforce has almost nothing in common with copying a file across. As soon as a package touches a target org, the platform sets about resolving a dependency graph, and in a mature org that graph fans out across hundreds of pieces: custom objects, Apex classes, automation flows, Lightning components, and validation rules alike. One unresolved reference anywhere in it, and the entire deployment collapses. Where that broken reference tends to live is the maddening part. It is usually buried in a component nobody on the team so much as opened, say a flow still pointing at a record type the same package just deleted, or an Apex class whose coverage quietly dipped because a sibling test class got refactored. The platform's own validation does surface these, but not until execution is already in motion, and it delivers the news through error text that is notoriously hard to parse. By the time you read it, the org is already in a mess.

What teams fall back on is instinct. Give a senior engineer enough scars and they develop a nose for the packages that warrant a closer look, the ones hauling big schema changes, deep dependency chains, or a suspiciously long gap since the last deploy. It is real expertise and it does the job, but it lives entirely in one person's head. It is nowhere in the pipeline, and it disappears the week that engineer takes leave. Watching the six organizations in this study before we introduced the tool, we saw production deployments fail 23.4% of the time, with each failure costing an average of 3.2 engineering hours to diagnose and recover from [1]. That was never a line item anyone planned for. It just quietly accrued.

Plenty of tools address slices of this. Code-level violations get caught by PMD and SonarQube ahead of a run [2]. Copado and Gearset both let you set quality gates that stop a deployment when coverage or rule counts fall short [3]. Consistent code review, where teams manage it, picks off the problems of intent. What no one of them attempts is the harder move: reasoning across the separate risk signals at once. Each inspects a single axis and answers pass or fail. That leaves an opening, because a package can satisfy every gate on its own terms and still be quietly dangerous when a deep dependency chain, a middling coverage drop, and a field deletion stack up together in a spot no individual gate was ever built to see [4].

2. Related Work

2.1. Salesforce Deployment Failures and Root Causes

In one of the larger studies here, Sharma and Venugopal combed through 1,200 Salesforce deployment events at 18 enterprise orgs. Dependency resolution errors, components pointing at things that either never made it into the package or had shifted incompatibly, drove 41% of the failures [5]. Another 28% traced back to schema changes, and within that slice it was field deletions and type changes that dominated the severe incidents. The last 31% came from test coverage regression, the usual story being a modified class that tugged the org's aggregate coverage below 75%. The through-line matters: none of these was visible by studying components one at a time. Each demanded that you reason about the package as a single object.

2.2. Static Analysis in Pre-Deployment Validation

The Apex ruleset in PMD flags the usual offenders, SOQL inside loops, empty catch blocks, hardcoded IDs, debug statements someone forgot to strip out of production code [6]. SonarQube brings coverage tracking on top, plus configurable gates for duplication and complexity [7]. The catch is that both live at the file level and answer each violation with a flat yes or no. Rahman et al. put pure rule-based static analysis up against a corpus of known Salesforce deployment failures and clocked precision at 61.3%, a fair bit under what the same class of tool manages on ordinary deployment targets, where the dependency graph is nowhere near as tangled [8]. It is precisely that distance, from file-level checks to whole-package structural risk, that this work is built to cover.

2.3. ML-Based Risk Prediction in CI/CD Pipelines

Machine learning has had its turn at build failure prediction from several directions. Gradient-boosted classifiers trained on commit history and CI metadata got Humbatova et al. to 89.2% precision on general CI failures [9]. Saidulu and Bhavani ran a comparable play against infrastructure deployment and pulled 84.7% recall out of infrastructure-as-code change vectors [10]. Solid numbers, both, but neither was earned on Salesforce. There is no tidy equivalent in either study's feature set for the Salesforce metadata model, with its flow versioning, its sharing rule interactions, its governor limit constraints, so lifting those models over would mean rebuilding the input representation more or less from scratch.

2.4. Metadata Complexity and Change Impact Analysis

Dependency chain depth is where Zanoni et al. focused, measuring it across enterprise Java systems, and it turned out to be the strongest single predictor of post-deployment defect density they found, edging out both file churn and historical defect counts [11]. Palomba et al. saw the same dependency-depth signal hold up in SaaS applications, and added schema mutation presence as a second heavyweight predictor; the pair together explained 67% of the variance in production incident probability [12]. Those two, DCD and SMF, happen to carry the third- and second-highest weights in our own model. We reached those weights on our own from calibration data, which makes the agreement with the literature a welcome confirmation rather than a borrowed premise.

2.5. Risk-Proportional CI/CD Governance

There is also the governance angle. Forsgren et al. found that organizations running graduated approval workflows, the kind where gate behavior tracks the estimated risk of a change, ship more often and do not pay for it in higher failure rates, unlike the teams that bolt one uniform threshold onto everything [13]. We took that straight into the gate design. Block every medium-risk package behind an architect sign-off and you generate so much friction that people simply learn to route around you. So we let the medium-risk gate demand extended test execution instead of a human signature, and reserved the person-in-the-loop hold for high-risk packages only. That is what kept the review burden sane, 127 of 847 deployments in Phase 2, a figure we return to in Section VII.

2.6. Gap Analysis

Line the options up and the shape of the gap is clear. Static analysis tools know Salesforce but speak only in binary, file-level verdicts. The ML predictors give you a composite number but were never trained on the Salesforce metadata model. Copado and Gearset ship quality gates, none of which combine several risk dimensions into one figure. What nobody offers is the full package at once: a Salesforce-specific feature model, a graduated numeric

score, and native CI/CD integration whose enforcement scales with the risk. Closing that is the whole point of this work.

3. System Architecture

3.1. Component Overview

Three components make up the tool, and they coordinate purely through subprocess invocation and file I/O, with nothing shared between them and no daemon left running in the background.

Everything kicks off in the Metadata Extraction Layer. Hand it the `package.xml` path and it reads the manifest, then dispatches Metadata API calls to the target org in parallel; what it emits is a Python dictionary tying ten feature names to their raw values. That dictionary flows into the Risk Scoring Engine, which loads its weight table from a JSON config file and returns a `RiskAssessment` object, the composite score, the tier, a per-feature contribution breakdown, and a remediation guidance string all bundled inside. Last comes the Pipeline Integration Layer, a deliberately thin adapter onto whichever CI/CD platform is in play, either a Copado Function Template or a Gearset hook script. It calls the scorer, parses the returned JSON, and applies the gate behavior tied to the assigned tier. We kept the scoring logic entirely platform-agnostic on purpose; anything platform-specific stays quarantined in the integration layer.

3.2. Metadata Extraction and Feature Construction

Once the manifest is read, the extraction layer fans its Metadata API requests out in parallel through `asyncio.gather()`. A few of the features are cheap: Component Count (CC) comes straight off the manifest with no API call at all. Test Coverage Delta (TCD) is pricier, needing two figures, the components' current coverage on the target org from `ApexCodeCoverage` records, and their projected coverage after the package lands, which we estimate from a sandbox dry-run; the delta is simply one minus the other. For Cyclomatic Complexity Score (CCS), the layer's own Apex parser tallies branching keywords (if, else if, for, while, when, catch) over every modified class and divides through by the class count. Dependency Chain Depth (DCD) marks how far a breadth-first walk across the org's graph of component references gets, started off each modified component and halting either at depth 10 or once the frontier empties, whichever lands first.

3.3. CI/CD Integration and Gate Behavior

Wherever the pipeline hook points it, that is where the scoring engine drops its JSON file at invocation. On the Copado side, a Function Template of about 60 lines of JavaScript reaches the analyzer through shelljs, reads the JSON back, and stamps three fields onto the live `Deployment__c` record: `Risk_Score__c`, `Risk_Tier__c`, and `Risk_Assessment_URL__c`, the last being a link out to the full assessment saved as a Salesforce File. From there a downstream branch keys off `Risk_Tier__c` and sends the deployment down one of three tracks. A Low-tier deployment simply keeps running through whatever gates already existed. A Medium-tier one gains an added step that executes every Apex test in the org, not merely those exercising the changed classes. A High-tier one gets frozen until a named architect updates a custom approval field and files a rollback plan, after which the pipeline picks back up. Gearset lands in the same place by another road: a 30-line Python hook invokes the scorer, reads `Risk_Tier__c`, and calls Gearset's REST API to set the deployment policy [17]. Whatever the route, every decision is written to the Copado audit record together with the full feature vector behind it.

4. Risk Scoring Model Design

4.1. Feature Selection

The ten features did not come from a whiteboard; they came from reading 412 post-mortem incident records that architects across the six organizations had already written up. Every post-mortem tagged its root causes against a shared taxonomy we had fixed before a single record was collected. For a feature to make the cut, it had to clear three tests at once: its data had to be reachable through the Metadata or Tooling API without anyone paying for Shield, it had to show up as a root cause in at least 15% of the post-mortems, and it had to be expressible as a number we could normalize. The lone holdout on that third test is the Schema Mutation Flag (SMF), which stays binary, tripping to 1 the moment a package carries any field deletion, field type change, object deletion, or record

type deletion. We gave it a weight of 0.15, the second-highest in the model, and the reason is blunt: schema mutations turned up in 82% of the worst historical incidents. Table I is the complete set.

TABLE I. RISK SCORING FEATURE SET, WEIGHTS, AND SENSITIVITY

Feature	Symbol	Weight	Category	Risk Sensitivity
Component Count	CC	0.12	Volume	Medium
Test Coverage Delta	TCD	0.18	Quality	High
Cyclomatic Complexity Score	CCS	0.16	Complexity	High
Dependency Chain Depth	DCD	0.14	Coupling	High
Schema Mutation Flag	SMF	0.15	Structural	Very High
Flow Impact Count	FIC	0.10	Automation	Medium
Historical Failure Rate	HFR	0.08	History	Medium
Deployment Interval	DI	0.04	Temporal	Low
API Version Disparity	AVD	0.02	Compat.	Low
Permission Set Impact	PSI	0.01	Access	Low

4.2. Scoring Algorithm

At its heart the scorer is nothing fancier than a weighted accumulator, and we left it that way on purpose; Listing 1 is the whole implementation. Feature by feature, `normalize()` pins the raw value to that feature's (low, high) bounds out of the calibration JSON and stretches it linearly onto [0.0, 1.0]. The one deviation is Test Coverage Delta, where `normalize_tcd()` runs a steeper slope on losses than on gains, and it does so through a plain `if/else` on the sign of the delta rather than any clever formula. After that the arithmetic is ordinary: each normalized value times its weight, the ten products summed, the total times 100, clamped into [0, 100], rounded to a single decimal. It is $O(n)$ in the feature count, and the scoring step lands in under a millisecond; whatever latency you feel is coming from the Metadata API calls upstream, not from here.

Listing 1. Weighted accumulator scorer (risk_scorer.py)

```
# risk_scorer.py - Weighted accumulator, O(n) in feature count
def compute_risk(features: dict, config: ScorerConfig) -> RiskAssessment:
    total = 0.0
    breakdown = {}
    for feat in config.feature_order:
        if feat == 'TCD':
            norm = normalize_tcd(features[feat], config.tcd_slopes)
        else:
            norm = normalize(features[feat], config.bounds[feat])
        contrib = config.weights[feat] * norm
        breakdown[feat] = round(contrib * 100, 2)
        total += contrib
    score = round(min(100.0, max(0.0, total * 100)), 1)
    tier = RiskTier.HIGH if score >= 66 else \
        RiskTier.MEDIUM if score >= 31 else RiskTier.LOW
    return RiskAssessment(score, tier, breakdown, config.guidance[tier])
```

4.3. Worked Example

To make it concrete, Table II follows one real Phase 2 deployment end to end, the end-of-quarter release package out of a financial services org. The package carried 89 components and two custom field deletions, so SMF tripped to 1. Coverage slid from 81% to 68%, putting TCD at -13%. Its deepest dependency chain ran to 8 (DCD = 8), and cyclomatic complexity averaged 21 across the modified Apex classes (CCS = 21). Eight live flows referenced components in the change (FIC = 8), and the package had gone 31 days since its last deploy (DI = 31).

Historical failure rate for those component types stood at 31% (HFR = 0.31), the API version spread came to 3 (AVD = 3), and 11 permission sets were in the blast radius (PSI = 11).

TABLE II. WORKED EXAMPLE: FEATURE-LEVEL SCORE BREAKDOWN (ORG-A, PHASE 2)

Feature	Raw Value	Normalized	Weight	Score Contribution
CC	89	0.44	0.12	5.3
TCD	-13%	0.88	0.18	15.8
CCS	21	0.69	0.16	11.0
DCD	8	0.80	0.14	11.2
SMF	1 (yes)	1.00	0.15	15.0
FIC	8	0.40	0.10	4.0
HFR	0.31	0.62	0.08	5.0
DI	31 days	0.34	0.04	1.4
AVD	3	0.30	0.02	0.6
PSI	11	0.22	0.01	0.2
TOTAL			1.00	70.5 -> HIGH

Notice that just three features, TCD at 15.8 points, SMF at 15.0, and DCD at 11.2, already account for 60% of the tally. At 70.5 the composite lands in the High tier. When the dashboard laid that breakdown in front of the release manager, the call was to break the package in two: a schema-only release with explicit rollback SQL written out, and a second release carrying the Apex changes once coverage was back up to 79%. Rescored, both came in under 30 and shipped without trouble.

4.4. Remediation Guidance

Bundled into every RiskAssessment object is a guidance string keyed to the tier, which the LWC dashboard renders and which also gets written into the Salesforce File assessment record. The wording is fixed per tier, but it tacks on the top three feature contributors by name and point value so an engineer can see at a glance where the pressure is coming from.

For Low-tier packages (score 0-30): "Package scored low-risk. Proceed with standard pipeline gates. No additional action required."

For Medium-tier packages (score 31-65): "Package scored medium-risk. The full org-wide Apex test suite will execute before deployment continues. Review any test failures carefully before proceeding, a failure in a test class not directly related to this package is a signal worth investigating. Top risk contributors: [feature 1, feature 2, feature 3 with point values]."

For High-tier packages (score 66-100): "Package scored high-risk and is held pending architect review. Assign this record to a senior architect. Before this deployment can resume, the architect must (1) review the feature breakdown below, (2) document a rollback procedure in the Rollback_Plan_c field, and (3) set Architect_Approved_c to true. Alternatively, revise the package to reduce the contributors driving the score above 65. Top risk contributors: [feature 1, feature 2, feature 3]."

4.5. Weight Calibration

Where we began was the related-work evidence: DCD and SMF up high because two independent studies had named them the strongest structural predictors, with TCD and CCS lifted as well. Those seeds then went through logistic regression over 340 labeled historical deployments, handed to us by three of the six study orgs before Phase 2 opened. The label was a single yes-or-no question asked of each deployment, did it cause a production incident within 48 hours that traced back to its components. We read the regression coefficients as evidence about relative importance, not as finished weights, so we rounded them to two decimals and renormalized the set back to a sum of 1.0. None of this hinges on the weights being optimal. They only need to rank packages well enough to be worth acting on, and out in the field they clearly do.

4.6. Tier Boundary Selection

For the cutpoints we settled on 30 and 65, arrived at by sweeping the confusion matrix over the 340-deployment training set at every integer boundary from 20 to 80. What we weighted most heavily was not letting high-risk

deployments slip past, so we traded away some false positives to hold down false negatives, on the plain logic that a production failure costs far more to clean up than a needless architect review does. The (30, 65) pair caught 91.4% of the historical failures as Medium or High against a 90% recall target, while tagging 14.2% of the successful deployments the same way. Push either boundary past about 5 points and one of those two numbers visibly suffered. We locked the training set before Phase 2 started, so no forward-looking data ever leaked into where the boundaries fell.

5. Implementation

5.1. Metadata Analyzer

The analyzer itself is a Python 3.11 script the CI/CD hook spins up as a subprocess. Metadata API access rides on the simple-salesforce library, and `asyncio.gather()` handles the parallel component queries. A 50-component package usually goes from invocation to JSON output in something like 1.4 to 2.1 seconds; at 200 components the median stretches to 3.8. Above 200 the DCD traversal stops descending at depth 6 and a fixed high-sensitivity penalty gets folded into CC, and there is a reason for that shortcut, packages that big failed north of 40% in the calibration data no matter what their other features looked like. Should the analyzer fail to reach the Metadata API, it exits non-zero and writes a diagnostic JSON, which the pipeline hook interprets as High tier by default, so a connectivity hiccup can never quietly wave a package through.

5.2. Pipeline Hook Integration

On Copado the work lands in a Function Template of roughly 60 lines of JavaScript. It reaches the analyzer via `shelljs`, reads the JSON, and writes `Risk_Score__c` and `Risk_Tier__c` onto the `Deployment__c` record through a Salesforce REST call. A second write spins up the File record holding the full assessment and drops its `ContentDocument ID` into `Risk_Assessment_URL__c`. For the routing, the step reading `Risk_Tier__c` rides on Copado's own conditional branching, where we added three branches aimed at the right pipeline stages, and that branch is the only structural edit any Copado pipeline needed. Gearset's counterpart is leaner still, 30 lines of Python that call the analyzer, read `Risk_Tier__c`, and hit Gearset's deployment policy API [17].

5.3. Assessment Dashboard

Rendering falls to a Lightning Web Component, which shows the assessment inline on the Copado pipeline record and, over on Gearset, in a side panel of the deployment comparison view. It puts up three things: the composite score as a color-coded radial gauge (green under 30, amber from 31 to 65, red above 65), a horizontal bar chart of per-feature contributions ordered high to low, and the tier's remediation guidance string. Of the twelve primary participants, three pointed specifically at that contribution chart as the output they actually worked from, since seeing that DCD alone accounted for 34 of a 71-point score told them exactly what to fix before resubmitting.

5.4. Configuration Management

Four structures live inside the weight configuration file (`scorer_config.json`): the feature weight table, the per-feature normalization bounds, the TCD slope constants, and the tier remediation guidance strings. We keep it under version control right next to the pipeline hook scripts in each org's DevOps repository, and write access is held to the pipeline administrator role, so no individual engineer can edit it. It also carries environment-specific overrides, a base block that sets population-average weights plus an optional `org_overrides` block that lets a team tune per-org without forking the base. Bump the version and an automated validation script fires, replaying the 340-deployment training set against the new weights and rejecting the change outright if recall slips under 90%. In six months that guard has tripped twice, both times over a well-meaning weight edit that had, without anyone noticing, dulled the model's sensitivity to schema-heavy packages.

6. Experimental Evaluation

6.1. Study Design

The study drew in six enterprise Salesforce organizations, each from a distinct industry vertical, over a six-month span; we refer to them as Org-A through Org-F, with their profiles collected in Table III. Phase 2 leaned on twelve DevOps engineers as primary participants, and they fed into every deployment and failure-rate

measurement, while a further nine engineers took part only in the closing usability questionnaire. Experience among the twelve ran from 3 to 11 years of Salesforce DevOps work, a median of 6, and every one of them owned production deployment decisions at their own shop. Everyone took part voluntarily, under institutional confidentiality agreements.

TABLE III. PARTICIPANT ORGANIZATION PROFILES

Org	Industry	Users	Apex Classes	Flows	Monthly Deploys
A	Financial Services	480	342	127	38
B	Healthcare	230	158	94	22
C	Manufacturing	410	289	183	31
D	Retail	145	97	62	17
E	Technology	560	421	215	52
F	Insurance	310	198	109	26

We split the work into two phases. Phase 1, the first three months, was retrospective: the scorer ran offline over each org's deployment history, its scores checked against outcomes we already knew so we could calibrate weights and place the tier boundaries, and no engineer ever laid eyes on a Phase 1 score. Phase 2, months four through six, went prospective, with the scorer live inside each org's pipeline and actively steering gate behavior for all 847 deployments that ran in that window. Phase 1 itself spanned roughly 558 deployments across the six orgs, a number we backed out from the monthly deployment counts in Table III over a three-month window, and it is what fixed the 23.4% baseline failure rate.

6.2. Ground Truth and Failure Attribution

A "failure," for our purposes, was any deployment followed within 48 hours by a production incident that traced back to that deployment's components, checked against Salesforce Shield Event Monitoring logs. Two wrinkles complicated this. Shield runs about an hour behind, a lag our companion paper gets into properly [14], and several orgs fired off multiple deployments inside a single day, which muddled attribution in a handful of cases.

So attribution went in three steps. We started by fixing each incident's deployment window off the Setup Audit Trail timestamp. Next we cut the candidate deployments down to those whose component lists intersected the components named in the incident's error records, leaning on the org's component reference graph to do it. Then, in the cases where things were still murky after those two passes, we took the incident straight to the engineer responsible and confirmed attribution by hand. That third step was needed for 19 of the 200 incidents across both phases, 9.5%, while the other 90.5% resolved cleanly through log correlation alone. And each of those 19 manual calls was checked independently by a second engineer before it earned a place in the dataset.

6.3. Baseline Characterization

Practice going into Phase 2 was anything but uniform. Four of the orgs ran Copado with coverage threshold gates and little else. One was on Gearset with a custom PMD ruleset. One deployed through the bare Salesforce CLI, leaning on nothing more than the platform's native validation. A composite risk score was absent everywhere. Baseline failure rates spread from 21.4% at Org-B to 26.0% at Org-E, and deployments run outside standard business hours, meaning 08:00 to 18:00 local, failed at 31.7%, close to 9 points over the mean, which squares with what our parallel health monitoring study turned up [14]. Part of the story is that four of the six orgs had overnight scheduled integrations that would occasionally launch deployments with nobody at the desk, and in those orgs after-hours packages made up 31 of the 131 Phase 1 failures.

7. Results

7.1. Prediction Accuracy

Over the 847 Phase 2 deployments, the scorer came in at 87.3% precision and 91.2% recall, an F1 of 89.2%, with the per-org picture in Table IV. The standout was Org-E, whose failure rate fell 71.8%; the backstory is that engineers there had long shipped packages with deep dependency chains simply because nobody could attach a

number to that risk. The moment DCD entered the score and high-DCD packages started getting held for architect review, that failure mode largely evaporated.

TABLE IV. PER-ORGANIZATION PREDICTION ACCURACY AND FAILURE RATE

Org	Deployments	Precision	Recall	F1	Fail Rate (Before)	Fail Rate (After)
A	158	88.4%	90.8%	89.6%	24.1%	7.9%
B	103	85.2%	89.4%	87.2%	21.4%	8.7%
C	139	88.1%	92.6%	90.3%	25.2%	7.9%
D	81	86.9%	90.1%	88.5%	22.2%	8.6%
E	219	89.2%	93.4%	91.3%	26.0%	7.3%
F	147	86.7%	91.7%	89.1%	21.6%	8.8%
Mean	847	87.3%	91.2%	89.2%	23.4%	8.1%

For significance we ran a chi-squared test over a 2x2 contingency table of failure counts by phase. Phase 1 baseline: 131 failures against 427 successes, $n = 558$. Phase 2 with the tool live: 69 failures against 778 successes, $n = 847$. That yields chi-squared = 64.7 at $df = 1$, with $p < 0.001$, which puts the failure-rate drop well beyond anything sampling noise could account for.

7.2. Gate Enforcement and Bypass Behavior

The scorer marked 127 of the 847 packages as High tier during Phase 2, 15.0% of them. Here is what happened to those 127: on 91 of them, 71.7%, the engineering team went back and reworked the package before the assigned architect had even opened the record, so the block by itself was doing the work. The remaining 36, 28.3%, went through to a full architect review, and of those, 29 came out approved with no changes once the architect was satisfied the rollback plan held, while 7 were bounced back for revision. Add it up and 98 of the 127 blocks, 77.2%, produced something real, either a reworked package or a rollback plan that had not existed a moment before. Only 29, 22.8%, could honestly be called a rubber stamp.

Over on the Medium tier, 184 packages set off the full org-wide Apex test run. 23 of them, 12.5%, flunked the extended tests and had to be revised before the deployment could go on, while the other 161, 87.5%, passed and carried through. The extended run tacked on a mean of 14.3 minutes to Medium-tier cycle time, which most participants shrugged off, though two engineers at Org-D flagged the scheduling hit in their exit interviews, specifically for weekend maintenance windows where the clock is tighter.

7.3. False Positives and False Negatives

On the false-positive side, 103 of the 847 deployments scored Medium or High and then succeeded without incident, a rate of 12.7%. The bulk were batch data migration packages, high on component count and moderate on historical failure rate but carrying no dependency or schema risk to speak of. At the manifest level the scorer still cannot tell a data migration apart from a code change, so it grades both on raw CC and ends up overstating what a migration actually risks. Teaching it to classify package type straight from the manifest is queued for a later version to close precisely that gap.

The 74 false negatives, 8.8%, were harder to characterize. The recurring culprit was first-deployment packages: a fresh Apex class has no HFR history and no prior DCD, so the model, with nothing to lean on, grades it conservatively. Let that class introduce a callout to a remote endpoint the org has not whitelisted in Remote Site Settings and the deployment fails, and that is a failure we simply cannot see without querying Remote Site Settings directly, which is going in as a feature in the next version.

The thinnest improvement was Org-B's, at 61.5%, and unpicking it after the fact turned up a very specific workaround. Engineers there were chopping schema-heavy packages into smaller ones, which drove each package's SMF to zero; the score dropped even though the real risk did not, because a field deletion is exactly as dangerous no matter which package happens to carry it. We have since bolted on an org-level SMF accumulator that rolls schema deletions up across every package deployed inside the same release window, which shuts that particular door.

7.4. Pipeline Latency and Operational Metrics

Scoring latency averaged 2.3 seconds (SD = 0.8 s) over all 847 deployments, with the 95th percentile at 4.1 seconds, and that tail showed up only on packages above 150 components where the DCD traversal ran into its depth ceiling; nothing timed out. Rollbacks fell from a baseline 18.7% to 6.3% in Phase 2, a 66.3% drop, as the high-risk packages were either fixed ahead of execution or deliberately put off. For the failures that did get through, mean time to recover came down from 3.2 hours to 1.8, a 43.8% cut, which we put down to the feature breakdown handing engineers a pointed place to start digging. All twenty-one participants, the twelve primary plus the nine usability-only, filled out the System Usability Scale at the close, and the mean came to 83.2 [15]; at $n = 21$ that is a preliminary read, so the Excellent label is directional rather than settled.

Table V places the tool beside the alternatives.

TABLE V. COMPARISON WITH EXISTING PRE-DEPLOYMENT VALIDATION APPROACHES

Approach	Pre-Deploy	Quantified Score	CI/CD Native	SF-Specific	Risk-Tiered Gates
This Framework	Yes	Yes	Yes	Yes	Yes
Copado Quality Gates	Yes	No	Yes	Yes	No
SonarQube Analysis	Yes	Yes	Yes	No	No
Manual Code Review	Yes	No	No	Yes	No
PMD Rule Enforcement	Yes	No	Yes	Yes	No

8. Discussion

8.1. What Actually Changed

A 65.4% cut in the failure rate is the headline, but the mechanism underneath deserves a moment. The tool did not win by blocking more deployments; it won by blocking a different set of them. In five of the six orgs, engineers described the same pattern, a High-tier block nearly always sent a package back for rework before resubmission, whether that meant splitting it, restoring coverage, or resequencing so schema changes came apart from code changes. The block manufactured a conversation that, in the old world, only ever happened after production had already broken. That is the shift that generated most of the benefit, and the gate data bears it out: 77.2% of High-tier blocks led to a genuine intervention rather than a rubber stamp. The falls in rollback frequency, 66.3%, and in mean recovery time, 43.8%, are downstream of the same thing, fewer failures making it to production at all.

8.2. Comparison with Existing Approaches

Read across Table V and the tool's signature is the combination itself, a quantified score paired with risk-tiered gates, which nothing else in the lineup delivers together. Living natively inside Salesforce and wired into the CI/CD flow, Copado Quality Gates still answer each metric in binary terms, so a package sitting borderline on three axes at once, each one under its own threshold, glides straight through even as the composite view would stop it. SonarQube does produce a score, but with no model of the Salesforce dependency graph it cannot compute DCD, FIC, or SMF at all. Manual review is comprehensive on paper and inconsistent in practice, tending to get dropped in exactly the crunch where risk peaks. PMD flags rule violations but carries no historical failure rate at all. None of this makes the tool a replacement; it sits on top of all of them.

8.3. Threats to Validity

A few caveats are worth stating plainly. Because our recruiting went out through professional contacts, the six orgs amount to a sample of convenience, and shops that agree to be studied tend to keep tighter DevOps than the norm, which could mean our baseline failure rates sit below what a random draw would show. The weights were calibrated on three orgs and then applied to all six; cross-org generalization held up in the data, but it is still an assumption riding along. Phase 2 also invites behavioral adaptation, and the Org-B package-splitting is a documented instance of knowing the scoring logic changing how packages got built, which flatters the apparent

effectiveness. The first-deployment false negative gap is a known limitation with a fix already underway. And the SUS result, at $n = 21$, is preliminary; a larger usability study is slated for the next evaluation cycle.

Acknowledgment

The authors thank the six enterprise Salesforce organizations that hosted the study, the twelve primary participants and nine usability-study participants whose deployment data and feedback shaped this work, and the six senior architects whose post-mortem records informed the feature selection and weight calibration in Section IV.

References

- [1] Salesforce Inc., "Salesforce DevOps Center Release and Deployment Guidance," Salesforce Developer Documentation, 2024. [Online]. Available: https://developer.salesforce.com/docs/atlas.en-us.pkg2_dev.meta
- [2] H. Panda, R. Ghosh, and S. Tripathy, "Static Analysis Tool Evaluation for Enterprise Apex Codebases: Coverage, Precision, and CI/CD Integration Overhead," in Proc. IEEE Int. Conf. Software Quality, Reliability and Security (QRS), 2022, pp. 312-323.
- [3] Copado Inc., "Copado Quality Gates and Compliance Rules Technical Reference," Product Documentation, 2024. [Online]. Available: <https://docs.copado.com/articles/copado-ci-cd-publication/quality-gates>
- [4] V. Basili, L. Briand, and W. Melo, "A Validation of Object-Oriented Design Metrics as Quality Indicators," IEEE Trans. Software Eng., vol. 22, no. 10, pp. 751-761, Oct. 1996.
- [5] A. Sharma and B. Venugopal, "Root Cause Taxonomy for Enterprise SaaS Deployment Failures: An Empirical Study of Salesforce Production Incidents," in Proc. IEEE Int. Conf. Cloud Computing (CLOUD), 2023, pp. 178-189.
- [6] PMD Project, "PMD Apex Rules and Static Analysis Reference," Open Source Documentation, 2024. [Online]. Available: https://pmd.github.io/pmd/pmd_rules_apex.html
- [7] SonarSource SA, "SonarQube Salesforce Plugin: Quality Gate Configuration Guide," Product Documentation, 2024. [Online]. Available: <https://docs.sonarqube.org/latest/analyzing-source-code/languages/salesforce>
- [8] A. Rahman, P. Rigby, and C. Bird, "A Systematic Mapping Study of Static Analysis Integration in Continuous Deployment Pipelines," in Proc. ACM/IEEE Int. Symp. Empirical Software Engineering and Measurement (ESEM), 2023, pp. 1-12.
- [9] N. Humbatova, G. Shabab, and G. Bavota, "Deep Learning-Based Failure Prediction for CI/CD Build Pipelines Using Commit Metadata Features," IEEE Trans. Software Eng., vol. 49, no. 5, pp. 2841-2858, May 2023.
- [10] D. Saidulu and R. Bhavani, "Machine Learning Approaches for Infrastructure Deployment Risk Prediction in Cloud DevOps Environments," IEEE Access, vol. 10, pp. 71842-71857, 2022.
- [11] M. Zanoni, F. Fontana, and F. Ferme, "Dependency Chain Depth as a Predictor of Post-Deployment Defect Density in Enterprise Java Systems," in Proc. IEEE Working Conf. Software Visualization (VISSOFT), 2022, pp. 33-43.
- [12] F. Palomba, G. Bavota, and A. De Lucia, "Schema Mutation Impact on Production Incident Probability in Cloud-Hosted SaaS Applications: A Large-Scale Empirical Study," IEEE Trans. Software Eng., vol. 50, no. 1, pp. 88-107, Jan. 2024.
- [13] N. Forsgren, J. Humble, and G. Kim, Accelerate: The Science of Lean Software and DevOps. Portland, OR: IT Revolution Press, 2018, ch. 3.
- [14] M. M. R. Seelam, V. Thanneeru, and A. Pagidipally, "A Browser-Native Telemetry Framework for Real-Time Organizational Health Monitoring in Enterprise Salesforce DevOps Environments," in Proc. IEEE Int. Conf. on Systems, Applications and Technology (LISAT), 2025.
- [15] A. Bangor, P. T. Kortum, and J. T. Miller, "An Empirical Evaluation of the System Usability Scale," Int. J. Human-Computer Interaction, vol. 24, no. 6, pp. 574-594, 2008.
- [16] M. M. R. Seelam, V. Thanneeru, A. K. R. Vemireddy, and S. Kudumula, "AI-Powered Salesforce Sandbox Cloning Using Generative Adversarial Networks," in Proc. 4th Int. Conf. Computer, Electronics, Electrical Engineering and their Applications (IC2E3), NIT Uttarakhand, India, 2026 (in press).
- [17] Gearset Ltd., "Gearset CI/CD Pipeline Hooks and Pre-Deployment Script Reference," Product Documentation, 2024. [Online]. Available: <https://docs.gearset.com/en/articles/pipeline-hooks>

Open Access This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial 4.0 International License (<http://creativecommons.org/licenses/by-nc/4.0/>), which permits any noncommercial use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

