PANDORIAN RESEARCH

Authoring Guidelines for LLM Enforcement

Observable Decision Boundaries in Code Governance

Pandorian Research

June 2026

ABSTRACT

We report observations from 1,105 findings produced by an LLM-based code governance system across 13 production guidelines deployed across three large software organizations. Finding precision varied dramatically across guidelines enforced by the same underlying model, ranging from above 90 percent to below 30 percent. The variation was not explained by model choice, because the model was held constant across the analyzed deployments. Nor was it explained by domain complexity alone, because guidelines of similar conceptual difficulty produced different outcomes. The strongest pattern we observed was whether the guideline could be converted into an explicit enforcement decision procedure. We characterize an enforceable guideline as a specification with four components: applicability, violation signature, exception semantics, and developer actionability. Guidelines fail when one of these components is left implicit. Ambiguous applicability produces findings in the wrong context. Weak violation signatures produce speculative findings. Conceptual exceptions produce false positives on legitimate code paths. Vague remediation produces findings that may be technically correct but operationally unhelpful. This paper reframes guideline authoring for LLM-based governance as the discipline of translating tacit engineering standards into observable decision boundaries. We propose a rubric for assessing guideline readiness before enforcement, show how noisy compound guidelines can be decomposed into enforceable units, and describe the failure modes associated with each missing component. Our conclusion is practical: for organizations deploying LLM-based code governance, the bottleneck is often not model capability alone, but the quality of the specifications the model is asked to enforce.

1. Introduction

Large language models have made a previously difficult governance problem newly tractable: enforcing engineering standards expressed in natural language across every pull request in a codebase. Traditional static analysis requires formal rule languages, AST-specific implementation, and brittle pattern matching. An LLM-based governance system can instead read a guideline written in ordinary engineering language, inspect the code under review, and produce findings that approximate what an experienced reviewer might flag.

The technology is useful, but it does not remove the need for precision in the standard being enforced. In production deployments, practitioners encounter a consistent pattern: two guidelines written by comparable engineering teams, enforced by the same model, and applied within large production codebases can produce dramatically different finding quality. One guideline produces findings that developers accept quickly. Another produces findings that are debatable, inconsistent, or ignored.

The simplest explanation would be that the lower-performing guideline addresses a harder technical domain. That explanation is incomplete. Some difficult rules perform acceptably when their enforcement conditions are explicit. Some apparently simple rules perform poorly when their scope, exceptions, or violation shape are left to interpretation. The decisive question is not merely whether the rule is hard. It is whether the rule gives the model enough structure to decide consistently.

This paper reports observations from 1,105 findings across 13 production guidelines deployed across three large software organizations with active code governance programs. We find that guideline quality is best understood not as a list of independent writing properties, but as a question of enforceability: can the guideline be applied as a decision procedure over code?

We use the term decision procedure in a practical rather than formal sense. A guideline does not need to be reducible to deterministic static analysis to be useful. But it does need to answer four questions explicitly enough that an LLM does not have to guess:

Applicability: when does this guideline apply?

Violation signature: what observable code shape constitutes a violation?

Exception semantics: what apparent violations should be suppressed?

Developer actionability: what should the developer do to fix the issue?

When these components are explicit, findings are more consistent, more defensible, and easier for developers to resolve. When they are implicit, the model fills gaps with plausible but unstable interpretation. That gap-filling is the source of many false positives, weak findings, and missed violations.

2. Dataset and Context

The dataset consists of 1,105 findings produced by an LLM-based code governance system deployed in production across three large software organizations over approximately three months. The system

reads engineering guidelines authored by each organization and applies them to pull requests, producing findings that are surfaced to developers before merge.

The 13 guidelines covered a representative cross-section of governance concerns: security, performance, reliability, correctness, component conventions, and architectural prescription. The code under review spanned Go, Python, JavaScript, TypeScript, and Vue.

Rule	Domain	Findings	Primary language
PII in logs	Security	11	Go
Lodash imports	Performance	7	JavaScript
HTTP status code usage	Correctness	6	Go
Fixed type usage	Correctness	15	Go
Idempotency keys	Correctness	6	Go
React patterns	Component conventions	17	TypeScript
Design system component usage	Component conventions	16	Vue / TypeScript
API error response shape	Correctness	5	Go
Return after error	Correctness	33	Go
Internationalization keys	Component conventions	192	Vue
Go error wrapping	Symbol enforcement	78	Go
Vuex state management	Architectural prescription	423	Vue
Airflow input sanitation	Security / data flow	296	Python

The volume distribution is uneven. Two rules account for nearly two-thirds of the dataset. We treat this as useful rather than disqualifying: high-volume rules expose aggregate behavior and recurring failure modes, while low-volume rules show whether the same patterns appear in smaller samples.

Findings were assessed by reading the guideline, the code, and the model-generated explanation. We classified a finding as a true positive when a domain expert would likely accept it as a legitimate violation. We classified it as a false positive when the finding applied the rule incorrectly. We treated borderline or hedged findings as operationally weak, even when they could be defended under a broad interpretation of the guideline.

This is an experience report, not a controlled empirical study. We do not claim statistically defensible precision estimates. We claim that the differences between rules were large, recurring, and tied to specific guideline structures in ways that practitioners can use.

3. The Core Observation: Guidelines Fail Where They Require Guessing

Across the dataset, noisy guidelines did not fail randomly. They failed where the model had to infer something the guideline did not state. Sometimes the missing information was scope: does this rule apply to error-level logs or only to info and warning logs? Sometimes it was category boundary: is a tenant identifier personal data, organizational data, or neither? Sometimes it was exception handling:

what concrete code marker tells the model that a log statement is an audit log? Sometimes it was remediation: what is the acceptable fix when the diagnostic value of the log must be preserved?

These are not merely documentation gaps. They are enforcement gaps. A human reviewer can often resolve them by applying tacit organizational knowledge. An LLM at pull-request time has less institutional context and must apply the guideline as written. If the guideline is incomplete, the model will still produce an answer. The answer may be plausible. It may even be defensible. But it will be less consistent, less predictable, and easier for developers to argue with.

The highest-precision guidelines in the dataset shared a simple property: they gave the model an observable target. The Lodash import rule prohibited specific import shapes. The Go error wrapping rule required a specific helper. The HTTP status code rule looked for recognizable constants in recognizable contexts. These guidelines left relatively little to infer.

The lowest-precision guidelines asked the model to enforce concepts without sufficient operational definition. The Vuex state management rule asked the model to detect architectural over-reliance. The PII logging rule combined personal identifiers, organizational identifiers, audit-log exceptions, log-level distinctions, and pseudonymization expectations in a single document. The model was not merely applying a rule; it was reconstructing the rule.

The practical implication is that guideline authoring should be evaluated by how much reconstruction it requires. A good guideline does not merely express intent. It exposes decision boundaries.

4. An Enforceable Guideline Has Four Components

We propose that an enforceable guideline has four components: applicability, violation signature, exception semantics, and developer actionability. These components are separable enough to diagnose different failure modes, but connected enough to form a single authoring discipline.

4.1 Applicability: When the Rule Applies

The first question an enforcement system must answer is whether the rule applies to the code under review at all. Many noisy findings in the dataset arose not because the model misunderstood the prohibited behavior, but because the guideline left scope underspecified.

Applicability includes language, framework, file path, environment, code-path category, log level, data sensitivity, trust boundary, and severity tier. If these dimensions are implicit, the model fills the gaps. Those gap-filling decisions are often reasonable in isolation but inconsistent across findings.

The PII logging guideline illustrates this failure mode. The guideline expressed a broad preference against logging PII while also naming specific log levels where the rule was especially strict. It did not fully specify the treatment of debug, error, or critical logs. As a result, the model applied the rule inconsistently across severity levels. A developer could reasonably read one finding as legitimate and another as overreach, even though both came from the same guideline.

The authoring fix is to make applicability explicit. For level-dependent rules, prose is usually inferior to a matrix.

Level	Applies?	Enforcement treatment
Debug	No	Local development only; not retained in production.
Info	Yes	Strict enforcement unless an explicit audit marker is present.
Warning	Yes	Strict enforcement unless an explicit audit marker is present.
Error	Yes	Advisory or pseudonymization-required, depending on the rule.
Critical	No or advisory	Permitted for recovery diagnostics if no safer signal exists.

For security rules, applicability must include trust boundaries. A SQL injection guideline should not merely say to sanitize input. It should state which sources are untrusted, which sources are trusted, and whether defense-in-depth cases are blocking or advisory. Without that boundary, the rule will fire on every dynamic query construction pattern, including cases built entirely from compile-time constants or validated allowlists.

The applicability test is simple: before asking whether the code violates the rule, ask whether the model can determine that the rule applies.

4.2 Violation Signature: What Code Shape Constitutes a Violation

Once applicability is established, the model must identify the violating code shape. We call this the violation signature: the observable pattern that should trigger a finding.

The strongest signatures are concrete symbols, function calls, imports, constants, decorators, file paths, AST shapes, or data-flow patterns. The weakest signatures are intentions, philosophies, or terms whose boundaries are implicit. The difference matters because an LLM can match concrete structure more reliably than it can infer unstated intent.

The Lodash import rule is a high-quality violation-signature rule. It prohibits full-bundle imports and points to specific import forms. The Go error wrapping rule similarly anchors enforcement on a known helper. The model does not need to infer the project philosophy behind the rule. It only needs to inspect whether the prohibited or required symbol appears in the relevant location.

By contrast, a rule such as "avoid over-reliance on singleton sources of data" asks the model to transform architectural intent into an enforcement target at runtime. The model must infer what counts as over-reliance, what counts as singleton state, and which contexts justify exceptions. Each inference creates variance.

A recurring variant of weak violation signatures is container-based detection. A guideline may intend to prohibit regulated content, but the finding fires on a container that might hold that content. For example, flagging every variable named URI because a URI could contain a user identifier is weaker than flagging a URI that demonstrably contains an email, user ID, or other regulated identifier. The former produces speculation. The latter produces evidence.

The authoring principle is to move from intent to observable evidence. Not "avoid unsafe URL handling," but "wrap untrusted path segments with url.PathEscape before interpolation." Not "use the proper error helper," but "return errors through goisms.WrapErrorFunc rather than returning err directly." Not "avoid hardcoded user-visible strings," but "Vue templates, label fields, formatter return values, and menu item definitions must reference i18n keys rather than string literals."

4.3 Exception Semantics: What Apparent Violations Should Be Suppressed

Most useful guidelines have legitimate exceptions. The question is not whether exceptions exist, but whether the model can recognize them in code. Conceptual exceptions are fragile. Code-grounded exceptions are enforceable.

The clearest contrast in the dataset is between the Airflow input sanitation guideline and the PII logging guideline. The Airflow guideline included a concrete exception: do not flag Variable.Get(). Because Variable.Get() is a named symbol, the model could recognize and honor it. The PII guideline included the conceptual exception "audit logs." But the codebase represented audit logs with concrete prefixes such as [AUDIT]. Because the guideline did not name those markers, the model flagged log statements that developers would likely regard as exempt.

The deeper lesson is that exception semantics should be stated in the same language the codebase uses. If an exception is represented by a prefix, name the prefix. If it is represented by a decorator, name the decorator. If it is represented by a file path, name the path pattern. If it is represented by a helper, name the helper. If multiple markers exist, enumerate them.

Exception semantics also include exclusion sets. Category-based rules need both inclusions and exclusions. A PII rule should define which identifiers are personal identifiers and which look related but are not. Without exclusions, the model will flag ambiguous cases with hedged language such as "may be considered PII." Those findings are not operationally useful. A developer cannot confidently fix a finding that the model itself presents as uncertain.

A strong guideline therefore includes near-misses: examples that resemble violations but are explicitly allowed. Near-misses are often more valuable than positive examples because they prevent false positives at the boundary of the rule.

4.4 Developer Actionability: What the Developer Should Do

The first three components determine whether the model can identify violations accurately. The fourth determines whether the finding is useful once identified. A guideline that finds a real problem but does not prescribe a fix leaves developers to invent remediation. That invention may be wrong, inconsistent, or unnecessarily costly.

Developer actionability should not be confused with precision. A finding can be a true positive and still be poor if the developer cannot tell how to resolve it. Conversely, remediation guidance does not necessarily improve detection accuracy. It improves adoption, consistency, and time to resolution.

The PII findings illustrate this distinction. A finding that correctly identifies a raw user identifier in a log statement may still be unhelpful if it offers no way to preserve the diagnostic value of the log. Developers then face two bad options: delete the signal entirely or argue with the finding. A better guideline states the acceptable remediation patterns: remove the identifier, hash it when correlation is needed, replace it with a request-scoped correlation ID, or move it to a scrubbed structured metadata field.

Actionability is especially important for judgment-dependent rules. Mechanical substitutions are easy to fix. Architectural or security findings require more guidance. The guideline should state not only what is wrong, but what the organization considers an acceptable fix.

5. Structural Rule Categories and Expected Difficulty

The four-component framework applies across guideline types, but not all guidelines are equally easy to enforce. Some categories naturally produce clearer violation signatures than others. The category of the rule should shape expectations for precision, enforcement tier, and authoring investment.

Category	Typical signature	Expected difficulty	Primary failure mode
Symbol enforcement	Named imports, helpers, constants, APIs	Low	Missing exception or outdated symbol list
Category-based rules	Data or entity classification	Medium	Ambiguous inclusion and exclusion boundaries
Structural correctness	Control-flow or multi-statement pattern	Medium	Context window or local reasoning limits
Component conventions	Prescribed component or helper in a known context	Medium	Overfitting to narrow examples
Data-flow and injection	Source-to-sink reasoning with trust boundary	High	Speculation about provenance or attacker control
Architectural prescription	Design judgment across modules	Very high	Subjective findings and weak local evidence

Symbol enforcement rules are the easiest. They tell the model to look for a named thing. Category-based rules are harder because the model must classify entities. Data-flow rules are harder still because they require provenance reasoning. Architectural prescription rules are the most difficult because they often ask the model to evaluate design intent from localized code context.

This does not mean hard categories should never be enforced. It means they should be tiered carefully. A symbol enforcement rule may be appropriate for blocking enforcement. An architectural prescription may be better as advisory feedback, or as a signal for human architectural review rather than automatic rejection.

6. A Readiness Rubric for Guideline Enforcement

We propose a scoring rubric that evaluates guideline readiness before deployment. The rubric is not a guarantee of precision. It is a discipline-forcing tool: it makes authors identify where the model would otherwise have to guess.

Dimension	0	1	2	3
Applicability	Scope absent or implicit	Scope described in prose only	Most scope dimensions explicit	All relevant scope dimensions explicit; matrices used where helpful
Violation signature	Pure intent or philosophy	Some examples but no stable signature	Concrete symbols or patterns named	Exact symbols, calls, AST shapes, or data-flow patterns stated
Exception semantics	No exceptions or conceptual exceptions only	Some exceptions named, but not code-grounded	Most exceptions tied to code markers	All exceptions and near-misses tied to observable code markers
Developer actionability	No remediation	General remediation direction	Concrete fix pattern	Multiple approved fix patterns by context or severity

A full score is 12 points. In practice, we recommend the following thresholds:

Score	Readiness	Recommended treatment
10-12	Production enforcement candidate	Deploy as blocking or high-confidence enforcement if the rule category supports it.
8-9	Monitored enforcement	Run advisory first, review findings, and refine scope or exceptions.
5-7	Rewrite before enforcement	The guideline has material gaps that will produce arguable findings.
0-4	Not currently enforceable	Use as documentation or architectural guidance, not automated enforcement.

The rubric should be interpreted in light of rule category. A 10-point symbol enforcement rule may be production-ready. A 10-point architectural prescription may still warrant advisory-only treatment because the underlying judgment is harder and less locally observable.

7. Worked Example: Rewriting the PII Logging Guideline

The PII logging guideline in the dataset combined several enforcement decisions: which identifiers count as personal data, which log levels are in scope, which audit-log patterns are exempt, how to treat organizational identifiers, and what pseudonymization patterns are acceptable. Each decision is reasonable. The problem is that they were bundled into one guideline, forcing the model to reconstruct the intended decision tree at finding time.

A better structure decomposes the concern into related but independently enforceable rules. The goal is not to weaken the security posture. It is to separate different decision boundaries so that each can be enforced with a clearer signature and triage tier.

Rule 1: Personal identifiers in non-audit logs

Applicability. Applies to Go backend services and to log.Infof, log.Warningf, and log.Errorf calls. Does not apply to debug-only logs, test fixtures, or critical recovery diagnostics unless explicitly configured otherwise.

Violation signature. The log format string interpolates a value whose name or field path matches the personal-identifier inclusion set.

Inclusion set. visitorId, userId, userEmail, userPhone, userIP, userName, firstName, lastName, and any field name ending in Email, Phone, or IP.

Exclusion set. Subscription, account, and tenant identifiers such as SubID, accountId, tenantId, and Tenant.ID; application identifiers such as app.Name, app.Key, and appId; correlation identifiers such as trace IDs, request IDs, and message IDs.

Exception semantics. Log statements whose format string begins with [AUDIT], [BILLING_AUDIT], or a matching [*_AUDIT] prefix are exempt.

Remediation. Remove the identifier, replace it with a request-scoped correlation ID, or apply an approved pseudonymization pattern.

Rule 2: Organizational identifiers in logs

Applicability. Applies to the same log calls as Rule 1, but only for organizational identifiers. This rule should be advisory unless the organization explicitly chooses to treat these identifiers as regulated data.

Violation signature. The log statement interpolates a tenant, subscription, account, or application identifier in a context where it is unnecessary for diagnostics.

Exception semantics. Audit logs and billing audit logs are exempt when marked by the approved prefixes. Operational logs are allowed when the identifier is required for customer support or incident response.

Remediation. Prefer a correlation ID or scrubbed structured metadata field. Do not remove the identifier when doing so would materially impair support or incident response without an alternative signal.

Rule 3: Pseudonymization for error-level diagnostic logs

Applicability. Applies to error-level logs in user-facing request paths where diagnostic correlation is needed.

Violation signature. The log interpolates a personal identifier without applying an approved pseudonymization pattern.

Approved patterns. Hash the identifier when cross-log correlation is needed. Use a request-scoped correlation ID when cross-service correlation is sufficient. Move the identifier to a scrubbed structured metadata field when the logging platform supports field-level controls.

This decomposition gives the governance program three rules, three severities, and three tuning loops. The security concern is the same, but the enforcement surface is clearer.

8. Precision, Recall, Trust, and Actionability

A recurring source of confusion in guideline evaluation is the tendency to treat all negative developer reactions as precision failures. They are not the same. A finding can fail in at least four ways.

Failure type	Meaning	Example
Precision failure	The finding is not actually a violation.	The model flags an audit log that is explicitly exempt.
Recall failure	The rule misses a real violation.	The i18n rule catches template strings but misses hardcoded labels in configuration objects.
Trust failure	The finding is debatable or inconsistent enough that developers lose confidence.	The model flags some tenant IDs as PII but ignores similar identifiers elsewhere.
Actionability failure	The finding is legitimate but does not tell the developer how to fix it.	A PII finding says to remove an identifier but gives no approved correlation alternative.

This distinction matters because authoring improvements affect different metrics. Clearer applicability and exception semantics usually improve precision. Broader examples may improve recall. Remediation guidance improves actionability. Consistency across findings improves trust. A mature governance program should monitor these separately rather than treating accepted findings as the only signal.

The dataset analyzed here is strongest on precision because it consists of generated findings. It does not directly reveal missed violations. A rule with very high precision and very low recall would look better in this dataset than it is operationally. Future evaluation should include sampled code regions with independent labeling of both violations and non-violations.

9. Limitations and Threats to Validity

This work is an experience report. Several limitations should constrain how strongly the conclusions are interpreted.

First, the dataset comes from three large software organizations rather than a single controlled environment. This strengthens the external validity of the observed patterns, because the findings span multiple codebases, engineering cultures, and internal conventions. At the same time, the deployments were not controlled experiments, and the exact precision numbers should not be treated as universal. The relevant claim is that the same failure modes recur across organizations, not that every organization should expect identical precision values.

Second, precision was assessed by the authors rather than by independent domain experts with measured inter-rater agreement across all three organizations. A stronger study would use multiple raters from each deploying organization and report agreement metrics.

Third, all findings were produced by a single underlying model. Different models may exhibit different failure modes. We expect the authoring principles to remain useful across models because they reduce ambiguity in the specification, but this remains an empirical question.

Fourth, rule difficulty and guideline quality are confounded. Symbol enforcement rules are intrinsically easier than data-flow or architectural rules. The framework helps explain differences within and across categories, but it does not eliminate the underlying difficulty gradient.

Finally, the dataset observes findings that were produced, not violations that were missed. The analysis therefore emphasizes precision and trust more than recall. This is appropriate for diagnosing noisy enforcement, but incomplete as a measure of total governance effectiveness.

10. Implications for Practitioners

The practical lesson is that LLM-based code governance is not primarily a prompt-writing exercise and not merely a model-selection problem. It is a specification discipline. The governance system can scale institutional knowledge only after that knowledge has been translated into a form the model can apply consistently.

Practitioners should invest in guideline authoring as a core operational function. Senior engineers and security leaders often carry the relevant standards tacitly. The work is to externalize those standards: define scope, name signatures, enumerate exceptions, and prescribe fixes. This work resembles documentation, but with a higher precision requirement because the immediate reader is an enforcement system rather than a forgiving human.

The most reliable improvement lever is decomposition, but decomposition should be understood carefully. The goal is not to split every complex sentence into separate rules. The goal is to separate independent decision boundaries. A single rule may contain several conditions if they combine into one violation type. But a guideline that mixes personal identifiers, organizational identifiers, audit exceptions, log-level policy, and pseudonymization expectations should almost certainly become multiple guidelines.

Enforcement tiering should reflect the rule category and rubric score. Symbol enforcement rules with clear signatures can often be blocking. Category-based and data-flow rules may begin as advisory until finding quality is measured. Architectural prescription rules should usually be treated as signals for human review unless they can be reduced to local, observable signatures.

A successful LLM governance program therefore looks less like autonomous AI judgment and more like operationalized expertise. The organization writes down what its best engineers know, in a form that can be checked consistently at pull-request time. The LLM scales that standard across contributors, repositories, and reviews. The leverage is real, but only when the standard is sufficiently explicit.

11. Conclusion

The central finding of this report is simple: LLMs enforce guidelines best when the guideline exposes observable decision boundaries in code. The important authoring question is not whether the standard

is written in natural language. It is whether the natural language contains enough structure for the model to decide consistently.

The four-component framework - applicability, violation signature, exception semantics, and developer actionability - offers a practical way to evaluate and improve guidelines before deployment. It explains why some rules produce trusted findings while others produce noisy or arguable ones. It also clarifies which failures are precision problems, which are recall problems, which are trust problems, and which are actionability problems.

Better models will improve LLM-based governance, but better models will not eliminate the need for better specifications. The bottleneck is often not intelligence. It is ambiguity. Organizations that want reliable enforcement should treat guideline authoring as the primary craft of LLM code governance.

Appendix A: Authoring Checklist

Before deploying a guideline, authors should be able to answer the following questions.

Applicability

Which languages, frameworks, directories, files, or code paths are in scope?

Which environments or tiers are in scope?

Which severities or levels are blocking, advisory, or exempt?

For security rules, which sources are trusted and untrusted?

Violation Signature

What exact symbol, call, import, constant, AST shape, or data-flow pattern should trigger the rule?

Does the rule fire on demonstrated evidence rather than theoretical possibility?

Can the model identify the violation from the available code context?

What common shapes of the violation exist in this codebase?

Exception Semantics

What code markers identify legitimate exceptions?

Are exclusions enumerated as clearly as inclusions?

Are near-miss examples provided?

Are exceptions tied to prefixes, helpers, decorators, file paths, or other observable markers?

Developer Actionability

What exact fix should the developer apply?

Are multiple approved fixes needed for different contexts?

Does the remediation preserve the legitimate engineering purpose of the original code?

Can the finding be resolved in minutes rather than requiring interpretation by the guideline author?