

Guardrails at the Admission Boundary: The Latency and Reliability Cost of Policy-as-Code Enforcement

Siri Chandana Chava

Sr. Cloud Engineer, Independent Researcher

sirichava.09@gmail.com

Abstract—Policy-as-code has become the default mechanism for enforcing security, compliance, and operational rules on cloud-native platforms, and its strongest enforcement point is the admission boundary, where an engine evaluates every proposed change before it takes effect. Admission is attractive because it offers complete mediation. For exactly that reason it is also a synchronous dependency on the critical path of every deployment, and platform teams rarely measure what it costs them. This article treats that cost as a first-class engineering concern. We order the available enforcement points, from commit through continuous integration to admission and runtime, by the coverage each provides and the feedback latency each imposes. We then analyze admission as a synchronous dependency whose tail latency is added to every change and whose availability is coupled to the control plane it guards, and we work through the failure modes that follow, including the choice between failing open, which opens a security gap, and failing closed, which opens an availability gap. From that analysis we develop a coverage-cost model in which marginal enforcement coverage is purchased with marginal latency and reliability budget, and we give a benchmarking method for locating a particular platform on that curve. The argument is that an admission-time policy engine is a reliability dependency as much as a security control, that its cost is a curve rather than a constant, and that placing policy well means measuring the curve instead of assuming it is flat.

Index Terms—policy-as-code, admission control, reference monitor, complete mediation, guardrails, Kubernetes, control plane, tail latency, fail-open, fail-closed, reliability, platform engineering.

I. INTRODUCTION

Policy-as-code replaces prose with programs. Instead of a wiki page stating that production workloads must not run as root, a platform team writes a rule and a policy engine evaluates it on every change. The rules become reviewable, testable, and version-controlled, and they are applied uniformly rather than whenever a reviewer happens to notice. The practice is sound and its rapid adoption is deserved. The open question is not whether to express policy as code, but where to enforce it, and what the strongest answer costs.

The strongest answer is the admission boundary: the point at which the control plane evaluates a proposed change before persisting it. Its appeal has a long pedigree. Anderson's reference monitor had to be tamperproof, small enough to verify, and, the property at issue here, always invoked [1]. Saltzer and Schroeder generalized that property into complete mediation and made it foundational, on the grounds that a check which can be bypassed yields a probability rather than a guarantee [2]. Admission control in a modern orchestrator [3] is complete mediation applied to configuration change.

The same property is what makes it expensive. Because every change passes through it, the engine is a synchronous dependency on the critical path of every deployment. What it adds to each change is its tail latency rather than its mean, since the caller waits for the slowest response and not the average one [4]. And its availability bounds the availability of the operations it mediates: nothing is admitted while the admitter is down.

A component with those properties is a reliability dependency of the platform. It is seldom treated as one. Policy engines arrive through the security door, so they tend to be evaluated on their security properties, deployed like ordinary workloads, and given none of the redundancy, latency budgeting, or dependency analysis a control-plane component would receive. The assumption underneath is that evaluation is fast and the engine is reliable. Neither is guaranteed, and when the assumption fails it fails at the worst possible moment: during an incident, when deployments are urgent and the engine, if it shares fate with the platform it guards, is most likely to be impaired.

This article treats that cost as something to engineer rather than something to discover. It contributes four things. First, a taxonomy of enforcement points, ordered by coverage and feedback latency, which locates admission as the point of strongest coverage and highest coupling. Second, an analysis of the admission boundary as a synchronous dependency: how its latency is added to every deployment, how its availability couples to the control plane, and which failure modes follow. Third, a coverage-cost model in which enforcement coverage is bought with marginal latency and reliability budget, and which is a curve rather than a constant. Fourth, a benchmarking method for measuring a particular policy layer instead of assuming its cost.

The examples are drawn from the Kubernetes admission model because it is the one most readers will have in hand, but the reasoning is platform-neutral. The tension between complete mediation and critical-path cost is a property of synchronous enforcement, not of any one system. The numbers, by contrast, do not travel at all. The cost of a policy layer depends on its rules, its data dependencies, and its deployment topology, and the decision to adopt policy-as-code says nothing about where on the curve a platform has landed.

II. BACKGROUND: COMPLETE MEDIATION AND THE COST OF ENFORCEMENT

A. The reference monitor and its abandoned third requirement

Anderson's 1972 planning study defined the reference monitor as the mechanism that mediates every access by a subject to an object, and set three requirements on it: it must be tamperproof, it must always be invoked, and it must be small enough to be subjected to analysis and test [1]. Saltzer and Schroeder restated the second requirement as complete mediation, one of their design principles for protection, reasoning that a check which can be skipped protects nothing [2].

The cost follows from the same definition. A mechanism invoked on every access sits on the critical path of every access, and the mediated operation can be no faster and no more available than its mediator. This is not an implementation defect waiting to be optimized away. It is the shape of the guarantee, and the trade it forces is the subject of this article.

What has changed since 1975 is the third requirement. Anderson's monitor arbitrated memory references, so the check was small, local, and bounded by construction, and the requirement that it be small enough to verify kept it that way. A contemporary admission controller evaluates arbitrary operator-supplied rules, written in a general-purpose policy language, over an object that may run to tens of kilobytes of serialized configuration, and it sometimes consults external services before deciding. The mediation has been faithfully preserved; the bound on the mediator has quietly been dropped. That is why performance, which the early literature could treat as a constraint to respect, becomes a first-order design concern in this setting.

B. Policy evaluation is computation

The line from Lampson's access matrix [5] through role-based [6] and attribute-based [7] models, to logic-based authorization languages such as Binder [8] and SecPAL [9], and to standards such as XACML [10], is a line of increasing expressiveness. Expressiveness is bought with evaluation cost, and the bill is presented on every mediated operation.

Comparing a string field is cheap. Iterating every container in a pod specification, cross-producting the images against a list of permitted registries, and then resolving a digest against a remote registry is not, and the difference is paid on every change the rule mediates, indefinitely. This is why the cost of a policy layer is properly a curve rather than a constant, and why it must be measured for the rules actually in force rather than inferred from the choice of engine. Two platforms running the same engine can sit at opposite ends of the curve.

C. Where a function belongs

Saltzer, Reed, and Clark argued that a function should be placed where it can be completely and correctly performed, and that implementing it at a lower level is an optimization rather than a substitute for correct placement [11]. Applied to policy the argument cuts in both directions. A rule that can be

decided correctly only with full knowledge of the change as it will finally exist belongs at admission, because that is where the knowledge exists and where mediation is complete. A rule that can be decided just as correctly earlier belongs earlier, where feedback is faster and the cost is off the deployment path. Section III is that argument applied to policy placement.

D. The cost of a synchronous dependency

Dean and Barroso showed that a synchronous dependency contributes its tail, not its mean, to the latency of the operation depending on it, and that with enough dependencies the tail comes to dominate [4]. A policy layer of many rules evaluated per admission contributes the tail of the slowest, not the average of all. Gray's case for failing fast [12] and Nygard's treatment of timeouts and circuit breakers [13] apply here without modification: a synchronous dependency on a critical path needs a bounded response time and a defined behavior on exceeding it, or a slow dependency becomes a platform-wide stall.

Kubernetes does bound the wait, with a per-webhook timeout capped at thirty seconds and defaulting to ten. That is a bound, but a permissive one. A ten-second stall on every deployment is not a degraded platform; it is an outage with a slow clock. The timeout exists to stop the API server hanging indefinitely, not to express anyone's latency budget, and treating the default as though it were a budget is one of the more common ways a team inherits a cost it never chose.

III. WHERE POLICY IS ENFORCED: A TAXONOMY OF ENFORCEMENT POINTS

Policy can be enforced at four points along the path from an authored change to its effect on the running system, and the points are not interchangeable. Each trades coverage against feedback latency and critical-path cost, as summarized in Fig. 1 and Table I.

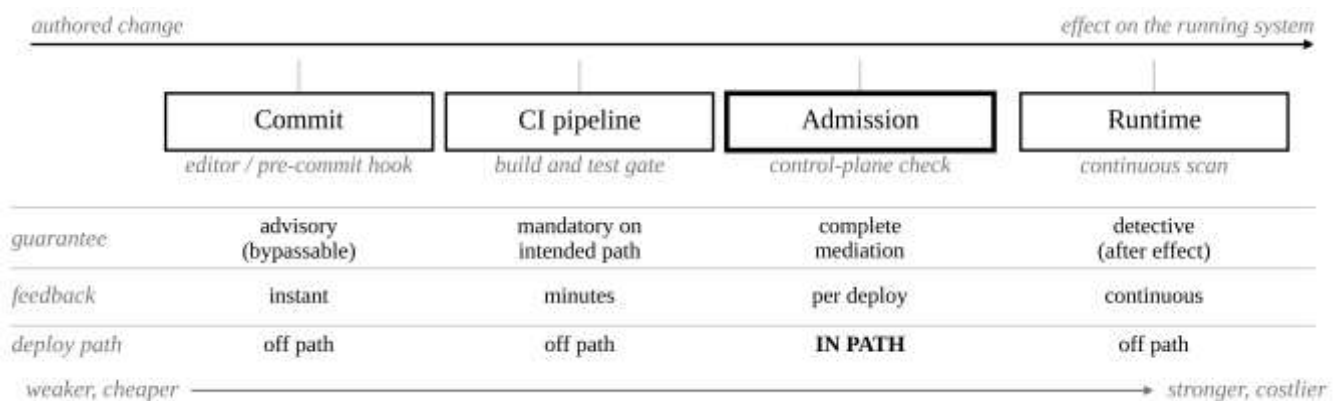


Fig. 1. Enforcement points along the path from an authored change to its effect. Earlier points give faster feedback and weaker guarantees; admission is the only point offering complete mediation, and the only one on the deployment path.

Commit-time enforcement lives in the editor, a pre-commit hook, or a local linter. Feedback is immediate and the cost falls on no shared path, which makes it the cheapest place to catch a mistake. It is also the weakest. A developer can skip a hook with a command-line flag, and the check sees the change without the context of the system it will run in. Commit-time policy is advisory: it reduces the number of violations that travel further, and it guarantees nothing.

CI enforcement lives in the build-and-test gate. Feedback arrives within the pipeline, still off the deployment critical path, and coverage is considerably stronger because the

pipeline can be made mandatory for the intended path. What it cannot cover is anything reaching the cluster by another route: a manual apply during an incident, a controller acting on its own, a manifest edited after the artifact was validated. A strong gate on the intended path is no gate at all on the paths around it.

Admission enforcement lives at the control-plane check, before the change is persisted. Coverage is strongest here: this is the one point every change traverses, and it sees the object in final form with the surrounding system state available. Which is precisely why it sits on the critical path of every

deployment and couples to the availability of the control plane. It is the only point that can guarantee a rule, and the only one whose cost is paid on every deploy.

Runtime enforcement scans the running system continuously. It catches what admission cannot: configuration drift, changes applied while a webhook was disabled, and violations arising from conditions that did not exist at admission time, such as an image that was clean when admitted and carries a critical vulnerability three weeks later. But runtime enforcement is detective rather than preventive. The violation has already taken effect, so what is on offer is eventual correction, not prevention.

TABLE I. ENFORCEMENT POINTS COMPARED

Point	Coverage	Feedback	On deploy path
Commit	Advisory; bypassable	Instant	No
CI	Mandatory on intended path only	Minutes	No
Admission	Complete mediation	Per deploy	Yes
Runtime	Detective; after effect	Continuous	No

The taxonomy yields a placement rule, which is the end-to-end argument [11] restated for policy: enforce each rule at the earliest point that actually guarantees it, and use admission only where admission is the earliest point that provides the guarantee. A rule that CI can check completely belongs in CI, off the critical path, duplicated at admission only when the guarantee genuinely requires mediation. Putting every rule at admission because admission is strongest is not the conservative choice it appears to be. It loads the deployment path with checks that could have run earlier at no loss of coverage, and it is the default outcome whenever placement is never decided deliberately.

IV. THE ADMISSION BOUNDARY AS A SYNCHRONOUS DEPENDENCY

The properties that make admission enforcement strong as a control are the same properties that make it load-bearing for reliability. They cannot be separated, and a team that adopts one has adopted the other whether or not it has noticed.

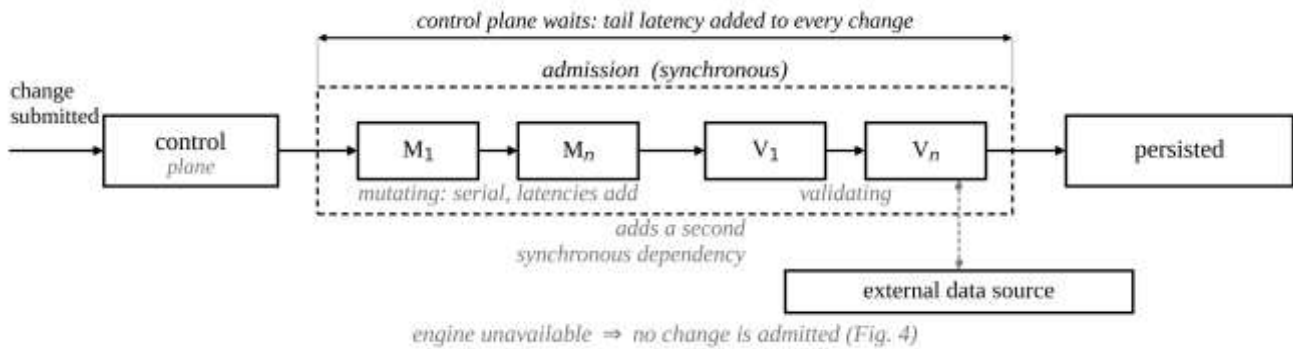


Fig. 2. The admission boundary on the critical path of a change. Mutating webhooks execute serially, so their latencies add; the control plane waits for the whole chain before persisting. A policy that consults an external source introduces a second synchronous dependency.

A. The two-phase path

A change submitted to the control plane traverses admission before it is persisted, in two phases. In the mutating phase, policy may modify the object: injecting a sidecar, setting a default resource limit, adding a required label. In the validating phase, policy accepts or rejects the object as submitted or as mutated. Both phases are synchronous, because the control plane cannot persist an object until it knows both whether the object is permitted and what the object should now be. That wait is the origin of everything else in this section.

The mutating phase deserves particular attention, because it is ordered. Mutating webhooks are invoked in sequence and each sees the output of the one before it, so their latencies add rather than overlap, and a mutation that a later validating hook then rejects has spent its cost for nothing. The admission path is, in the general case, a serial pipeline whose length is set by whoever registers webhooks, frequently several teams, none of whom sees the total.

B. The latency cost

The engine's response time is added to every change. For an operator applying a change interactively it is a pause. For a

controller reconciling continuously, or a rollout touching hundreds of objects, it is a throughput ceiling and a compounding source of tail latency.

Because the control plane waits, the number that matters is the tail. Consider a layer whose median evaluation takes two milliseconds and whose 99.9th percentile takes four seconds. That is not a fast layer with an occasional hiccup. It is a layer that stalls roughly one admission in a thousand for four seconds, and a rollout touching enough objects will hit that tail reliably rather than occasionally [4]. A mean that looks comfortable can conceal a tail that is not, which is the single most common reporting error we have seen in this area.

C. The reliability coupling

Because the control plane must consult the engine to admit a change, the engine's availability gates the availability of every operation requiring admission. Deployment availability is bounded above by engine availability, so an engine less available than the platform's stated target silently caps that target.

This requirement is rarely written down, and what is not written down is not engineered. A policy engine deployed as an ordinary workload, with a single replica, no disruption budget, and no anti-affinity, has an availability well below

what the control plane it guards promises. Cluster-management systems have long treated admission and scheduling as control-plane concerns rather than tenant workloads [14], and the operational literature is explicit that a dependency's availability target must be at least as strong as the target of whatever depends on it [15]. A policy engine given ordinary-workload treatment is a reliability liability in proportion to the criticality of what it mediates.

D. The circular dependency

The coupling can close on itself. If the engine runs on the cluster it guards, recovering the engine may require admitting a change, which requires the engine, which is down. Crash-only design treats recoverability as a property to be engineered rather than assumed [16], and this is a case where it is routinely assumed. The usual mitigation, excluding the engine's own namespace from its webhook configuration, works, but it is also a hole in the mediation, and it should be recognized as a trade rather than a free fix.

Correlated failure sharpens the problem. An engine sharing nodes, network, and control plane with the workloads it guards tends to fail when they do, which is exactly when its exemption matters most. Post-hoc studies of large-scale service outages find shared dependencies at the root of a disproportionate share of the largest incidents [17].

None of this argues against admission enforcement. It argues that adopting admission enforcement for its security properties without engineering for its reliability properties means accepting an unaccounted dependency, and that the bill arrives when the platform can least afford it.

V. A COST MODEL: LATENCY, RELIABILITY, AND COVERAGE

The cost of admission enforcement has three dimensions. Two of them are costs, and the third is the benefit that the first two purchase.

- **Latency cost:** the response time the engine adds to a change, reported at the tail. It decomposes into policy evaluation, external data fetches, and fixed per-call overhead. Because it is a distribution rather than a number, its tail is set by the most expensive rule and the slowest fetch, so a single data-dependent policy can wreck the tail while barely moving the average.
- **Reliability cost:** the reduction in deployment availability caused by the coupling to the engine. Deployment availability is at most engine availability, so an engine below the platform's target caps that target, and the failure-mode exposure discussed in Section VII is part of this cost rather than separate from it.
- **Coverage:** the benefit the two costs buy. Not the number of policies configured, but the fraction of the rules that actually matter which are enforced where they cannot be bypassed.

The relationship between them is a curve, not a constant. The first policies moved to admission cover the rules that matter most, at modest cost. As coverage approaches completeness, the marginal policies tend to be more expensive to evaluate, more likely to need external data, and less important than the ones already covered, so marginal cost rises while marginal benefit falls. Any platform running admission policy occupies some point on that curve. The design question is which point, chosen deliberately, rather than whether to enforce at all.

VI. THE COVERAGE-COST CURVE

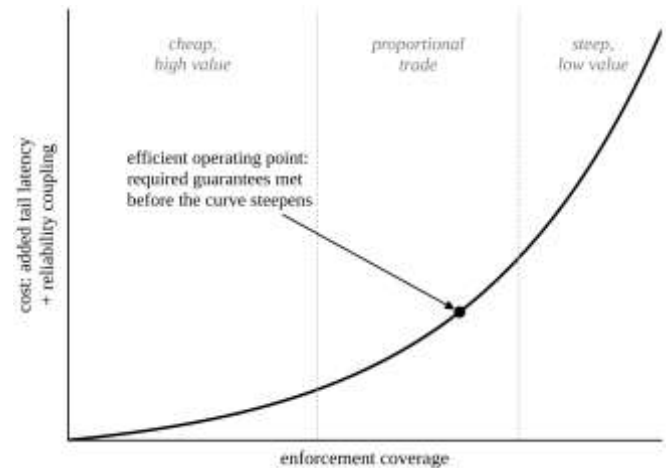


Fig. 3. The coverage-cost curve. Cost, in added tail latency and reliability coupling, rises faster than coverage as coverage approaches completeness. The efficient operating point is where the required guarantees are met, before the curve steepens.

The curve relates enforcement coverage to the cost of achieving it, where cost combines added tail latency with reliability coupling. It has three regions.

At low coverage, the first policies moved to admission are those whose guarantees matter most and whose evaluation is cheapest, so coverage rises quickly for very little cost. This region is unambiguously worth occupying. In the middle region, coverage and cost rise roughly together: a straightforward exchange of latency and reliability budget for guarantee. Approaching completeness, the marginal policies are the most expensive to evaluate, the most likely to need external data and the dependency that comes with it, and the least important. Cost climbs steeply while benefit flattens, and this region is usually not worth occupying.

The efficient operating point is where the required guarantees are met and before the curve steepens. Its location is specific to a platform, because it depends both on which guarantees genuinely need admission and on the cost of the particular policies providing them. What the curve buys is a change of question, from a binary one, should this policy be enforced at admission, to a marginal one, does this policy's guarantee justify its position on the cost curve. The marginal framing is what interrupts the usual outcome, in which a policy layer accretes indefinitely because each individual rule, considered alone, seemed worth enforcing strongly.

A. What sets the steepening

Two forces determine where the curve turns up, and both can be measured. The first is the distribution of evaluation cost. Uniformly cheap policies keep the curve flat until high coverage; a handful of expensive ones steepen it as soon as they are included, which is an argument for enforcing those earlier if their guarantee permits. The second is the prevalence of external data dependencies. A policy deciding from the change alone is cheap and reliable; a policy fetching external data adds both latency and a dependency, so the curve steepens sharply as data-dependent policies are added. Moving those to a point where the data is already available is often the single most effective way to flatten it.

The curve also exposes a second trade, with the failure modes below. Higher admission coverage means more of the platform's guarantees depend on the engine's availability, so reliability exposure rises with coverage as well. A platform far

up the curve has made itself more dependent on the engine and more exposed to its failure, and that cost appears nowhere in the latency numbers. It has to be counted separately.

VII. FAILURE MODES: FAIL-OPEN AND FAIL-CLOSED

When the engine is unavailable, the control plane must do something with the changes arriving at it. Both available answers are bad, and the useful move is to stop looking for a good one.

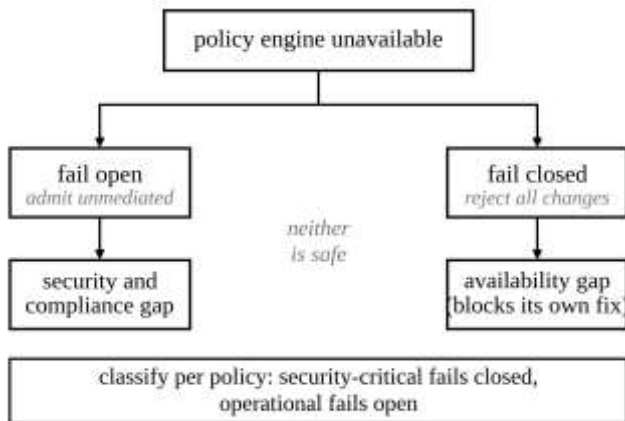


Fig. 4. The choice forced when the policy engine is unavailable. Failing open opens a security and compliance gap; failing closed opens an availability gap and can block its own remediation. The choice belongs to the policy, not to the platform.

Failing open admits changes without evaluating policy, at the cost of a security and compliance gap. For the duration of the outage, changes that policy would have rejected are admitted, and the guarantee is suspended exactly when the system is already degraded and more likely to be receiving hurried, risky changes. For a security-critical rule, that window is the one an attacker or an error would find.

Failing closed rejects changes while the engine is down, at the cost of an availability gap. Nothing deploys, including the fix for the incident in progress. If remediation requires a deployment, the engine's unavailability has blocked its own recovery, which is the deadlock of Section IV-D arriving by a different route.

Bainbridge's ironies of automation land squarely here [18]. The automated mechanism handles all the routine cases and hands the operator the hardest one, at the worst time, with the least practice. A fail-closed policy layer during an incident does precisely that: the deployment path is blocked, the people who need it are under time pressure, and the emergency bypass is a procedure nobody has exercised.

Neither choice is correct in general, and the resolution is to recognize that the choice is per-policy rather than per-platform. A security-critical rule whose suspension creates real exposure should fail closed and accept the availability cost, because admitting unmediated changes past it is worse than blocking deployment. An operational rule whose suspension is tolerable should fail open and accept the coverage gap, because blocking all deployment to enforce a naming convention is worse than admitting a few changes that skip it. A single uniform failure policy is wrong for one class or the other by construction. In Kubernetes terms this means the failure policy is a per-webhook decision that deserves the same review as the rule itself, and a webhook configuration in which every entry carries the same setting is a strong sign the question was never asked.

A. Reducing exposure to the choice

Better than choosing well is having to choose less often, which returns to the taxonomy. A policy enforced at admission and also in CI retains partial coverage when admission fails open, because the earlier check already rejected the obvious violations. That earlier enforcement is not redundant; it lowers the cost of the admission boundary's failure mode, and it is the main practical argument for deliberate duplication. Separately, an engine that does not share fate with the platform it guards does not fail when the platform is stressed, so it does not force the choice at the moment the choice is most expensive.

VIII. THE LATENCY ANATOMY OF A POLICY DECISION

To measure or reduce admission latency, decompose a single decision into its components. They differ by orders of magnitude and they respond to entirely different fixes, so an aggregate number tells you almost nothing about what to do next.

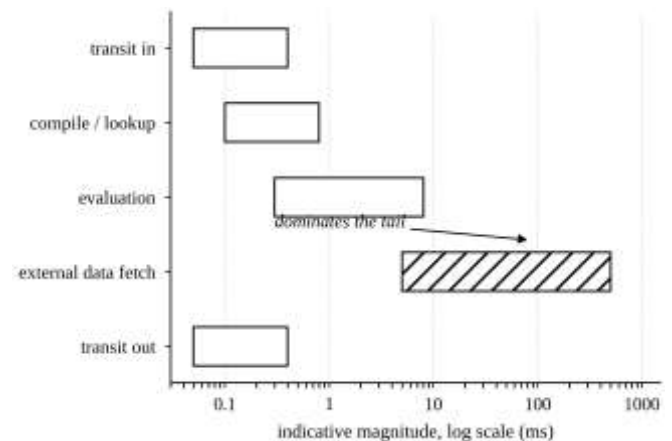


Fig. 5. Components of a single admission decision, on a log scale. Magnitudes are indicative orders of magnitude consistent with published datacenter latency figures [19], not measurements taken in this work; the point is the separation between components, which spans roughly four orders of magnitude.

Transit in is the request from control plane to engine, network plus serialization. It is small but not zero, and it grows if the engine is remote. Compile or lookup is the fixed per-call cost of preparing the policy for evaluation, and it is reducible by caching compiled policies rather than recompiling per request, a common and entirely avoidable overhead. Evaluation is the policy computation applied to the change, and its cost depends on language expressiveness and rule complexity: a field comparison is cheap, iterating a large input or matching a complex condition is not. This is where policy authors have the most direct influence, and where a poor choice compounds across every change the rule mediates.

External data fetch is consulting an outside source to reach a decision, and it is frequently the dominant term and almost always the dominant contributor to the tail. It introduces a second synchronous dependency with its own latency and availability, converting a fast local computation into a network round trip whose tail is set by the slowest source. The scale gap is what makes this decisive: an in-cluster call is a sub-millisecond affair, while a call leaving the datacenter is several orders of magnitude more expensive, and the standard datacenter latency figures are the right ones to reason with [19]. Transit out mirrors transit in and is similarly small.

The components differ not only in magnitude but in variability. Transit and compile are small and stable, evaluation is moderate and policy-dependent, and external

fetches are large, variable, and tail-dominating. The practical consequence is that admission tail latency is usually a data-dependency problem rather than a computation problem, and the most effective reductions come from removing fetches from the admission path rather than from optimizing rules. Measure the anatomy, not the aggregate, or you will optimize the component you can see instead of the one that is costing you.

IX. BENCHMARKING METHODOLOGY

The model and the curve are only useful to a team that can locate itself on them, and that measurement is specific enough to need a defined method. Three quantities are needed.

- **Admission latency**, as a distribution reported at the tail, measured under realistic concurrency rather than at rest, and decomposed along the anatomy of Section VIII.
- **Reliability cost**, observed from the engine's own availability history and from induced unavailability, to establish what the platform actually does rather than what its configuration suggests.
- **Coverage**, the fraction of the rules that matter which are enforced under complete mediation, measured against an enumerated list of required rules rather than against the set of policies that happen to exist.

Load matters more than it might appear. Concurrency is where admission layers fail, because a queue forms: by Little's law the number of in-flight evaluations rises with arrival rate and service time together [20], so a service-time increase that looks tolerable in a quiet cluster becomes a queue during a large rollout, and the tail degrades non-linearly rather than proportionally. A benchmark taken at rest measures the case that was never in question.

Inducing unavailability is a fault injection and needs the usual care around blast radius and abort conditions [21], but there is no substitute for it. Teams that believe they fail closed frequently discover a namespace exclusion added during an incident two years ago and never removed. Teams that believe they fail open frequently discover one webhook set to fail closed that nobody remembers registering. The configuration is a claim; the injection is the test.

Two pitfalls are worth naming because both are easy to fall into and both understate the cost. The first is benchmarking evaluation in isolation from its data dependencies, which measures the cheap component and omits the expensive one, since the fetch is normally the dominant tail term and is exactly what a static-input benchmark leaves out. The second is reporting the mean rather than the tail, which understates what the control plane actually experiences, since the control plane waits for the slow evaluation and not the average one.

The output of the exercise should be three artifacts: a tail latency for admission decomposed by component, a measured reliability cost including the platform's observed behavior under engine unavailability, and a coverage figure against an enumerated set of required rules. Together those place the platform on the curve with real numbers, which is what makes it possible to choose an operating point rather than inherit one. The numbers are specific to the policies, data dependencies, and topology they were taken from, so they must be taken locally rather than borrowed from a published benchmark of a different layer.

X. DESIGN GUIDANCE

The taxonomy, the cost model, the curve, and the anatomy converge on a handful of choices.

- **Enforce each policy at the earliest point that guarantees it.** This is the end-to-end argument applied to policy placement. A rule that CI can check completely belongs in CI, and it belongs at admission only where admission is the earliest point that provides the guarantee. Asking that question once per policy is the single most effective control on cost.
- **Keep external data out of the admission path.** Fetches dominate the tail and add a dependency. A policy that decides from the change alone is both cheaper and more reliable. Where a rule genuinely needs external state, bring the data to the path in advance by caching or replication rather than fetching it synchronously during evaluation.
- **Classify each policy by its failure mode.** Security-critical rules fail closed, operational rules fail open, and a uniform setting across a webhook configuration is wrong for one class by construction.
- **Engineer the engine as a control-plane dependency.** Its availability gates deployment, so give it the redundancy, spread, and operational care a control-plane component receives, and keep its recovery path independent of its own availability.
- **Use defense in depth across enforcement points.** A rule enforced both at admission and earlier retains partial coverage when admission fails open, which lowers the cost of that failure mode rather than merely duplicating work.
- **Choose the operating point deliberately, and measure to know where it is.** The efficient point is where required guarantees are met before the curve steepens, and the steepening is platform-specific. A platform that has never measured is almost certainly further up the curve than it intended to be.

Read together, these amount to giving the policy layer an explicit budget. A latency budget for the admission path and an availability target for the engine, both stated in advance and both tracked, turn the accumulation of policy from something that happens into something that is decided [22]. Every rule proposed for admission then has a visible price, which is the condition under which the marginal question in Section VI can actually be asked.

XI. OPEN PROBLEMS

Several problems limit how completely the cost of policy enforcement can be managed.

- **Incremental evaluation.** A large share of admission cost re-evaluates policies whose relevant inputs have not materially changed. Evaluating only the affected part, or precomputing partial results, would flatten the curve, but incremental evaluation of arbitrary policy with correctness guarantees remains unsolved.
- **The external data dependency.** A policy needing external state inherits that source's latency and availability. Caching and replication help, but deciding quickly and reliably against genuinely changing external state, with a characterized freshness-versus-cost trade, is still hard.
- **Coverage measurement.** Without a complete enumeration of the rules that ought to be enforced, coverage can only be measured against the policies that happen to exist, which measures enforcement rather than coverage. A method for

establishing the full policy surface would put the curve on much firmer ground.

- **Policy composition.** As policies multiply, their interactions, conflicts, and redundancies grow faster than their number, and reasoning about the combined behavior and combined cost of a large set, including contradiction and subsumption, is weakly supported by current tooling. A platform with several hundred policies has an aggregate behavior that nobody fully understands.

XII. CONCLUSION

Policy-as-code at the admission boundary is the strongest form of automated policy, because admission is the point of complete mediation. Every change passes the check and none can bypass it. The same property makes it a synchronous dependency on the critical path of every deployment, whose latency is added to every change and whose availability gates every change, and that reliability cost goes unmeasured largely because the security framing obscures it.

Treating the cost as a design parameter means seeing three things at once: that policy can be enforced at several points which trade coverage against feedback latency and critical-path cost; that admission offers the strongest coverage at the highest coupling; and that the cost of admission enforcement is a curve rather than a constant, so that marginal coverage becomes more expensive and less valuable as it approaches completeness. The efficient operating point is where the required guarantees are met, before the curve steepens. The failure modes the coupling forces, open into a coverage gap or closed into an availability gap, are policy-specific and must be classified per rule. And the latency cost is dominated by external data dependencies rather than by evaluation, which is what tells a team where to optimize.

None of this is exotic, and none of the building blocks are new. The reference monitor is fifty years old, the end-to-end argument is forty, and the enforcement-point taxonomy will be familiar to anyone who has built a deployment pipeline. What distinguishes a platform that can rely on its guardrails from one that is surprised by them is simply that it has measured the latency and reliability cost of its policy layer instead of assuming that cost negligible. That assumption is tested every time the engine is slow or unavailable, and the test arrives, as it always does, when the platform can least afford to fail it.

REFERENCES

- [1] J. P. Anderson, "Computer security technology planning study," U.S. Air Force Electronic Systems Division, Bedford, MA, USA, Tech. Rep. ESD-TR-73-51, 1972.
- [2] J. H. Saltzer and M. D. Schroeder, "The protection of information in computer systems," *Proc. IEEE*, vol. 63, no. 9, pp. 1278-1308, Sep. 1975.
- [3] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, Omega, and Kubernetes," *ACM Queue*, vol. 14, no. 1, pp. 70-93, 2016.
- [4] J. Dean and L. A. Barroso, "The tail at scale," *Commun. ACM*, vol. 56, no. 2, pp. 74-80, Feb. 2013.
- [5] B. W. Lampson, "Protection," *ACM SIGOPS Oper. Syst. Rev.*, vol. 8, no. 1, pp. 18-24, Jan. 1974.
- [6] D. F. Ferraiolo and D. R. Kuhn, "Role-based access control," in *Proc. 15th Nat. Comput. Security Conf.*, 1992, pp. 554-563.
- [7] V. C. Hu et al., "Guide to attribute based access control (ABAC) definition and considerations," NIST, Gaithersburg, MD, USA, Special Publication 800-162, 2014.
- [8] J. DeTreville, "Binder, a logic-based security language," in *Proc. IEEE Symp. Security and Privacy*, 2002, pp. 105-113.
- [9] M. Y. Becker, C. Fournet, and A. D. Gordon, "Design and semantics of a decentralized authorization language," in *Proc. 20th IEEE Comput. Security Found. Symp. (CSF)*, 2007, pp. 3-15.
- [10] OASIS, "eXtensible access control markup language (XACML) version 3.0," OASIS Standard, Jan. 2013.
- [11] J. H. Saltzer, D. P. Reed, and D. D. Clark, "End-to-end arguments in system design," *ACM Trans. Comput. Syst.*, vol. 2, no. 4, pp. 277-288, Nov. 1984.

- [12] J. Gray, "Why do computers stop and what can be done about it?," in *Proc. 5th Symp. Reliability in Distributed Software and Database Systems*, 1986, pp. 3-12.
- [13] M. T. Nygard, *Release It! Design and Deploy Production-Ready Software*, 2nd ed. Raleigh, NC, USA: Pragmatic Bookshelf, 2018.
- [14] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at Google with Borg," in *Proc. 10th Eur. Conf. Comput. Syst. (EuroSys)*, 2015, pp. 1-17.
- [15] B. Beyer, C. Jones, J. Petoff, and N. R. Murphy, Eds., *Site Reliability Engineering: How Google Runs Production Systems*. Sebastopol, CA, USA: O'Reilly Media, 2016.
- [16] G. Candea and A. Fox, "Crash-only software," in *Proc. 9th Workshop Hot Topics in Operating Systems (HotOS)*, 2003, pp. 67-72.
- [17] H. S. Gunawi et al., "Why does the cloud stop computing? Lessons from hundreds of service outages," in *Proc. 7th ACM Symp. Cloud Computing (SoCC)*, 2016, pp. 1-16.
- [18] L. Bainbridge, "Ironies of automation," *Automatica*, vol. 19, no. 6, pp. 775-779, 1983.
- [19] L. A. Barroso, U. Hölzle, and P. Ranganathan, *The Datacenter as a Computer: Designing Warehouse-Scale Machines*, 3rd ed. San Rafael, CA, USA: Morgan and Claypool, 2018.
- [20] J. D. C. Little, "A proof for the queuing formula: $L = \lambda W$," *Oper. Res.*, vol. 9, no. 3, pp. 383-387, 1961.
- [21] A. Basiri et al., "Chaos engineering," *IEEE Softw.*, vol. 33, no. 3, pp. 35-41, May 2016.
- [22] B. Beyer, N. R. Murphy, D. K. Rensin, K. Kawahara, and S. Thorne, Eds., *The Site Reliability Workbook: Practical Ways to Implement SRE*. Sebastopol, CA, USA: O'Reilly Media, 2018.