

Calibration in Converge: What, Why, and How

This document explains how Converge adapts its thresholds and policy profiles to actual team behavior through data-driven calibration, origin-based overrides, and gradual rollout. It is written for an external audience — no prior knowledge of the Converge internals is assumed.

1. What Problem Does Calibration Solve?

Hard-coded thresholds don't fit every team. A team of 3 developers working on a small library produces fundamentally different changes than a team of 50 working on a monorepo. If entropy budgets, containment thresholds, and check requirements are static, they will be either too loose for one team (letting risky changes through) or too tight for another (blocking routine work).

Converge addresses this through three mechanisms:

Mechanism	What It Does
Percentile-based calibration	Adjusts entropy budgets from historical data
Policy profiles with JSON override	Lets teams customize all thresholds per risk level
Origin-based overrides	Different policies for humans, agents, and integrations
Gradual enforcement (rollout)	Safely introduces new enforcement rules without blocking everything at once

2. Percentile-Based Calibration

2.1 How It Works

The calibration function takes historical risk evaluation data and recalculates entropy budgets using percentile analysis:

```
def calibrate_profiles(historical_scores, base_profiles=None):
```

Given: sorted list of historical entropy_score values (from all recent intents)

N = number of historical data points

P75 = value at the 75th percentile ← "what 75% of changes achieve"

P90 = value at the 90th percentile ← "what 90% of changes achieve"

P95 = value at the 95th percentile ← "what 95% of changes achieve"

Low budget = max(P75 × 1.5, 10.0)

Medium budget = max(P75, 8.0)

High budget = max(P90, 5.0)

Critical budget = max(P95 × 0.8, 3.0)

Source: `policy.py`, function `calibrate_profiles`

2.2 Why Percentiles?

The core insight: The best threshold for “too much entropy” is derived from what the team actually produces. If 75% of a team’s changes have entropy below 12, then 12 is a reasonable baseline for medium-risk changes. If 90% are below 18, then 18 is a reasonable ceiling for high-risk changes.

Percentiles are:

- Robust to outliers: Unlike averages, a single extreme change doesn’t distort the calibration
- Self-adjusting: As the team’s behavior changes (smaller changes, better practices), budgets tighten naturally
- Interpretable: “P75 = 12” means “75% of our changes have entropy at or below 12”

2.3 Per-Level Rationale

Low Risk: **max(P75 × 1.5, 10.0)** The most lenient budget. The P75 is multiplied by 1.5 to give low-risk changes 50% more headroom than the typical change. Low-risk changes have limited blast radius, so being generous is safe.

Floor: 10.0 — even for teams with very low entropy, the system maintains a minimum threshold to catch genuinely problematic changes.

Medium Risk: **max(P75, 8.0)** The baseline budget equals the P75 exactly. This means the typical change (75th percentile) is exactly at the budget limit. Medium-risk changes should not exceed what 75% of changes achieve.

Floor: 8.0 — prevents the budget from dropping too low for teams with very clean histories.

High Risk: **max(P90, 5.0)** Uses the P90 — a tighter percentile. Only 10% of historical changes exceeded this entropy level. High-risk changes must be better than 90% of historical changes.

Floor: 5.0 — a minimum that ensures some enforcement even for teams with near-zero historical entropy.

Critical Risk: **max(P95 × 0.8, 3.0)** The strictest budget. Uses P95 multiplied by 0.8 — deliberately 20% stricter than the team’s historical 95th percentile. The reasoning: for critical changes (core infrastructure, production branches), “what we’ve gotten away with before” isn’t good enough. The 0.8 multiplier adds an extra safety margin.

Floor: 3.0 — the absolute minimum. Even the most relaxed team gets a tight budget for critical changes.

2.4 Calibration Constants

Constant	Value	Purpose
CALIB_P75	0.75	Percentile index for medium/low budgets
CALIB_P90	0.90	Percentile index for high budget

Constant	Value	Purpose
CALIB_P95	0.95	Percentile index for critical budget
CALIB_LOW_MULT	1.5	Headroom multiplier for low-risk budget
CALIB_CRITICAL_MULT	0.8	Tightening multiplier for critical budget
CALIB_FLOOR_LOW	10.0	Minimum entropy budget for low risk
CALIB_FLOOR_MEDIUM	8.0	Minimum entropy budget for medium risk
CALIB_FLOOR_HIGH	5.0	Minimum entropy budget for high risk
CALIB_FLOOR_CRITICAL	3.0	Minimum entropy budget for critical risk

Source: defaults.py

3. Policy Profiles

3.1 Default Profiles

Each risk level has a default profile that controls all policy thresholds:

```
DEFAULT_PROFILES = {
    "low":      {
        "entropy_budget": 25.0,
        "containment_min": 0.3,
        "blast_limit": 50.0,
        "checks": ["lint"],
        "coherence_pass": 75,
        "coherence_warn": 60,
    },
    "medium":   {
        "entropy_budget": 18.0,
        "containment_min": 0.5,
        "blast_limit": 35.0,
        "checks": ["lint"],
        "coherence_pass": 75,
        "coherence_warn": 60,
    },
    "high":     {
        "entropy_budget": 12.0,
        "containment_min": 0.7,
        "blast_limit": 20.0,
        "checks": ["lint", "unit_tests"],
    }
}
```

```

        "coherence_pass": 80,
        "coherence_warn": 65,
    },
    "critical": {
        "entropy_budget": 6.0,
        "containment_min": 0.85,
        "blast_limit": 10.0,
        "checks": ["lint", "unit_tests"],
        "coherence_pass": 85,
        "coherence_warn": 70,
    },
}

```

Source: defaults.py

3.2 The Design Pattern: Tighter With Risk

Every threshold follows the same pattern — stricter as risk increases:

Element	Low → Critical	Direction
Entropy budget	25.0 → 6.0	Decreasing (less disorder allowed)
Containment min	0.30 → 0.85	Increasing (more isolation required)
Blast limit	50.0 → 10.0	Decreasing (smaller blast radius allowed)
Coherence pass	75 → 85	Increasing (higher coherence required)
Coherence warn	60 → 70	Increasing (narrower warn zone)
Required checks	lint → lint + tests	More checks

This is the fundamental invariant of the policy system: higher risk = tighter constraints across all dimensions.

3.3 JSON Override

Teams customize profiles by placing a file at .converge/policy.json, policy.json, or policy.default.json:

```
{
  "profiles": {
    "medium": {
      "entropy_budget": 20.0,
      "checks": ["lint", "unit_tests"]
    },
    "high": {
      "checks": ["lint", "unit_tests", "integration_tests"]
    }
  }
}
```

```

        },
    },
    "queue": {
        "max_retries": 5,
        "default_target": "develop"
    },
    "risk": {
        "max_risk_score": 70.0
    }
}

```

The override merges with defaults — only specified fields are replaced. A team that only wants to change the medium entropy budget doesn't need to re-specify all other thresholds.

Source: `policy.py`, function `load_config`

3.4 Config Loading Priority

The config loader tries paths in order, using the first one found:

1. Explicit path (if provided via `--config` flag)
2. `.converge/policy.json`
3. `policy.json`
4. `policy.default.json`

If none exist, defaults are used unchanged.

4. Origin-Based Overrides

4.1 The Problem

Changes come from three sources:

Origin	Typical Source	Trust Profile
human	Developer writing code	Standard trust — humans understand context
agent	AI coding agent	Lower trust — agents may not understand all implications
integration	CI/CD automation	Variable — depends on the integration

A team might want:

- Agents to pass `unit_tests` even for medium-risk changes (humans only need lint)
- Integrations to have higher containment requirements
- Agents to have stricter entropy budgets

4.2 How It Works

Origin overrides are specified in the policy config:

```
{
  "origin_overrides": {
    "agent": {
      "medium": {
        "checks": ["lint", "unit_tests"],
        "containment_min": 0.6
      },
      "high": {
        "checks": ["lint", "unit_tests", "integration_tests"]
      },
      "_default": {
        "entropy_budget": 15.0
      }
    }
  }
}
```

When loading a profile for a given risk level and origin type:

```
def profile_for(self, risk_level, origin_type=None):
    base = dict(self.profiles[risk_level])  # start with base profile
    if origin_type and self.origin_overrides:
        origin_rules = self.origin_overrides.get(origin_type, {})
        # Look for risk-level-specific override, fall back to _default
        overrides = origin_rules.get(risk_level, origin_rules.get("_default", {}))
        if overrides:
            base.update(overrides)  # merge overrides
    return base
```

Source: policy.py, method PolicyConfig.profile_for

4.3 The `_default` Key

The `_default` key applies to all risk levels that don't have a specific override. In the example above, `"entropy_budget": 15.0` applies to all agent changes regardless of risk level, unless a specific level overrides it further.

5. Gradual Enforcement (Canary Rollout)

5.1 The Problem

Introducing a new enforcement rule (like a risk gate) to a live system is risky. If the threshold is wrong, it could block every change in the pipeline. Gradual rollout lets you test enforcement on a fraction of changes before full deployment.

5.2 How It Works

The risk gate supports three modes:

Mode	enforce_ratio	Effect
shadow	—	Logs <code>would_block</code> but never actually blocks. Safe for observation.
enforce	0.0	Never blocks (equivalent to shadow)
enforce	0.5	Blocks 50% of changes that would be blocked
enforce	1.0	Blocks all changes that would be blocked (full enforcement)

5.3 Deterministic Bucketing

The enrollment decision uses a deterministic hash of the intent ID:

```
def _rollout_bucket(intent_id: str) -> float:
    h = hashlib.sha256(intent_id.encode()).hexdigest()[:8]
    return int(h, 16) / 0xFFFFFFFF      # float in [0, 1]

enforced = (mode == "enforce"
            AND would_block
            AND bucket < enforce_ratio)
```

Why deterministic? If intent abc123 has a bucket of 0.37 and the `enforce_ratio` is 0.5, it's in the enforcement group. On a retry, the same intent still has bucket 0.37 — the decision is consistent. Random bucketing would mean the same intent might be enforced on one attempt and not on the next, which is confusing and hard to debug.

Why SHA-256? Provides uniform distribution across the [0, 1) range with no correlation between similar intent IDs.

Source: `policy.py`, functions `_rollout_bucket` and `evaluate_risk_gate`

5.4 Rollout Strategy

A typical rollout of a new risk gate:

Week 1: mode="shadow"
 → Observe: how many changes would be blocked?
 → Tune: adjust thresholds if too many false positives

Week 2: mode="enforce", `enforce_ratio=0.1`
 → 10% of would-be-blocked changes are actually blocked
 → Monitor: are the blocked changes truly risky?

Week 3: mode="enforce", `enforce_ratio=0.5`
 → 50% enforcement
 → Continue monitoring

Week 4: mode="enforce", enforce_ratio=1.0
→ Full enforcement

6. Risk Gate Thresholds

The risk gate evaluates three composite scores against configurable limits:

Metric	Default Threshold	What It Means
risk_score	65.0	Overall risk above 65 is excessive
damage_score	60.0	Potential damage above 60 is excessive
propagation_score	55.0	Blast radius above 55 is excessive

These are checked in the RISK_GATE_CHECKS table:

```
RISK_GATE_CHECKS = [  
    ("risk_score", "max_risk_score", 65.0),  
    ("damage_score", "max_damage_score", 60.0),  
    ("propagation_score", "max_propagation_score", 55.0),  
]
```

A breach occurs when any metric exceeds its limit. Multiple breaches are recorded but the gate triggers on the first one.

Source: defaults.py

7. Health-Based Calibration

Beyond entropy budget calibration, the health projection system also adapts behavior based on system-level metrics:

7.1 Intake Throttling

When repo health drops, the system throttles or pauses new intents:

Condition	Effect
Health < 60	Throttle intake to 50% (INTAKE_THROTTLE_RATIO)
Health < 30	Pause intake entirely

Source: defaults.py, constants INTAKE_PAUSE_BELOW_HEALTH, INTAKE_THROTTLE_BELOW_HEALTH, INTAKE_THROTTLE_RATIO

7.2 Predictive Health Gate

The health prediction system can recommend blocking new intents before the system reaches a critical state:

```
If projected_status == "red" AND current_status != "red":  
    → Signal: "predict.approaching_red" (severity: critical)  
    → Recommendation: "Consider pausing new intents"  
    → should_gate = True
```

This is proactive calibration — the system adjusts behavior not just based on current state but on the trajectory of that state.

Source: `projections/health.py`, function `_detect_health_signals`

8. Learning System

When calibration alone isn't enough, Converge generates structured learning — actionable recommendations derived from metrics:

Condition	Lesson Code	Action
Mergeable rate < 85%	<code>learn.low_mergeable</code>	"Reduce average change size and enforce pre-merge checks"
Average entropy > 15	<code>learn.high_entropy</code>	"Split large intents into smaller focused changes"
Rejected count > 3	<code>learn.frequent_rejections</code>	"Review policy thresholds and source branch preparation workflows"
Health below 70	<code>learn.health_below_target</code>	"Prioritize resolving conflicts, reducing entropy, and clearing the queue"

Each lesson includes: - Code: Machine-readable identifier - Title: Human-readable summary - Why: Explanation of why this metric matters - Action: Specific next step - Priority: 0 (critical) to 2 (medium) - Metric: Observed value vs target value (for agents to act on programmatically)

Source: `projections/learning.py`

9. End-to-End Calibration Flow

1. Historical risk evaluations accumulate in event log

2. calibrate_profiles() called:
 - Extract entropy_score values from historical data
 - Sort and compute P75, P90, P95
 - Apply multipliers and floors
 - Update profile entropy budgets
 3. Team customizes .converge/policy.json:
 - Override any threshold per risk level
 - Add origin-specific overrides
 4. Intent arrives with origin_type:
 - Load base profile for risk level
 - Apply origin overrides (if any)
 - Final profile used for gate evaluation
 5. Risk gate evaluates with rollout:
 - Check risk/damage/propagation vs limits
 - Compute deterministic bucket from intent ID
 - Apply enforcement based on mode + ratio
 6. Health system monitors system-wide metrics:
 - If health declining → throttle intake
 - If trajectory toward red → recommend gate
 - Generate learning lessons for action
 7. Cycle repeats: new evaluations feed future calibrations
-

10. Design Rationale Summary

Design Choice	Rationale
Percentile-based calibration	Adapts to actual team behavior. Hard-coded thresholds either fit one team or fit none.
Minimum floors	Safety net — even teams with very clean histories maintain minimum enforcement levels.
P95 × 0.8 for critical	“What we’ve gotten away with before” is not enough for critical changes. The 20% tightening provides a safety margin.
JSON override (merge, not replace)	Teams only specify what they want to change. No need to re-specify all defaults for one adjustment.

Design Choice	Rationale
Origin-based overrides	Different trust profiles for different change sources. Agents may need stricter enforcement than humans.
<code>_default</code> key for origins	Simplifies configuration. One line applies to all risk levels for an origin type.
Deterministic rollout bucketing	Consistent enforcement across retries. Same intent always in same bucket. No random variance.
Shadow mode	Safe observation before enforcement. Collects data on would-be-blocks without disrupting workflow.
Gradual enforcement ratio	Risk reduction during rollout. Problems surface at 10% affecting a fraction of changes, not 100%.
Health-based intake control	When the system is degrading, accepting more changes makes it worse. Throttling is a stabilization mechanism.
Structured learning	Machines can read <code>metric.observed</code> vs <code>metric.target</code> . Humans can read <code>action</code> . Both get value.

11. File Reference

File	Role
<code>policy.py</code>	<code>calibrate_profiles</code> , <code>load_config</code> , <code>PolicyConfig.profile_for</code> , <code>evaluate_risk_gate</code> , <code>_rollout_bucket</code>
<code>defaults.py</code>	All calibration constants, default profiles, risk thresholds, intake thresholds
<code>projections/health.py</code>	Repo health, change health, predictive health gate, velocity detection
<code>projections/learning.py</code>	Structured learning derivation from health and change metrics
<code>models.py</code>	<code>RiskLevel</code> enum, <code>PolicyConfig</code> profile structure