Entropy in Converge: What, Why, and How

This document explains how Converge uses the concept of entropy to measure, control, and predict the health of a codebase under continuous integration. It is written for an external audience — no prior knowledge of the Converge internals is assumed.

---

1. What Problem Does Entropy Solve?

When multiple developers (or agents) contribute changes to a shared codebase concurrently, integration risk doesn't grow linearly — it compounds. A ten-file change that touches three unrelated directories is riskier than a twenty-file change confined to a single module, even though the latter is "bigger." Traditional metrics (lines changed, number of files) miss this.

Entropy in Converge captures the disorder a change introduces: how scattered it is, how many subsystems it crosses, how many dependencies it drags in, and how fragmented the resulting dependency graph becomes. This is not Shannon entropy in the information-theory sense — it's a domain-specific metric inspired by the same idea: a measure of how much uncertainty and complexity a change injects into the system's structural state.

Converge uses entropy at two levels:

Level	Question it answers	Scope
Change-level (per intent)	"How much disorder does this change introduce?"	Single proposed merge
System-level (aggregate)	"Is the codebase trending toward chaos?"	All recent activity

---

2. Change-Level Entropy: Entropic Load

2.1 The Formula

Every proposed change (called an "intent" in Converge) is evaluated by the function `compute_entropic_load`. The formula is a weighted sum of five structural factors, clamped to a 0–100 scale:

```
entropic_load = clamp(
    files_count      ×  2.0  +
    conflict_count   × 15.0  +
    deps_count       ×  6.0  +
    dir_spread       ×  3.0  +
    (components − 1) ×  5.0
, 0, 100)
```

Source: `risk/signals.py`, function `compute_entropic_load`

2.2 What Each Factor Measures and Why

Files Changed (**`files_count × 2.0`**) What: The number of files modified by the change.

Why this matters: More files = more surface area for unexpected interactions. Each additional file is another moving part that could interact with other concurrent changes.

Why weight 2.0: Moderate weight — file count alone is a weak signal (a large but self-contained refactor is fine). This gives it baseline presence without dominating the score.

Example: A change touching 10 files contributes 20 points.

Merge Conflicts (**`conflict_count × 15.0`**) What: The number of merge conflicts detected when simulating the merge in an isolated worktree.

Why this matters: Conflicts are direct evidence that this change collides with the current target branch state. Each conflict means manual resolution, which introduces risk of semantic errors.

Why weight 15.0: The heaviest weight in the formula. Conflicts are the strongest signal of disorder — they prove that the change doesn't compose cleanly with the existing codebase. Even a single conflict is a significant event.

Example: 2 conflicts contribute 30 points — immediately pushing the score into the "high entropy" diagnostic range.

Dependencies on Other Intents (**`deps_count × 6.0`**) What: The number of other intents (pending changes) that this intent depends on.

Why this matters: Each dependency creates an ordering constraint. If intent A depends on intents B and C, then B and C must merge first, in the right order, successfully. Dependencies multiply the integration surface and create fragile chains — if any upstream intent fails, this one is blocked.

Why weight 6.0: Moderate-high weight. Dependencies aren't inherently bad, but each one adds a constraint that makes the merge ordering problem harder. At 3+ dependencies (18 points), the signal becomes meaningful.

Example: An intent with 4 dependencies contributes 24 points.

Directory Spread (**`dir_spread × 3.0`**) What: The count of unique directories touched by the changed files.

Why this matters: Directory spread is a proxy for architectural dispersion. A change that touches `src/auth/`, `src/payments/`, `tests/integration/`, and `docs/` is crossing four architectural boundaries. Changes with high directory spread are harder to review, harder to roll back, and more likely to produce unexpected interactions because they cut across module boundaries.

Why weight 3.0: Lower than conflicts or dependencies because touching multiple directories isn't inherently dangerous — it depends on context. But it consistently predicts review difficulty and merge risk.

Example: A change spanning 5 directories contributes 15 points.

Graph Fragmentation (**`(components − 1) × 5.0`**) What: The number of weakly connected components in the dependency graph, minus one. The dependency graph is built from the changed files, their directory containment, co-location relationships, semantic scopes, and explicit dependencies.

Why this matters: A connected graph means all parts of the change are related to each other — there's a structural reason they're being modified together. Multiple disconnected components mean the change bundles unrelated modifications. This is the classic "drive-by change" problem: someone fixes a bug and also reformats an unrelated file, creating a logically fragmented commit.

Why weight 5.0: Fragmentation is a strong signal of poor change hygiene. Each additional disconnected component means the change is actually N independent changes bundled into one, which complicates review, blame, and rollback.

Example: A dependency graph with 3 components contributes 10 points: `(3−1) × 5.0`.

2.3 Interpretation Scale

Score	Interpretation	Typical trigger
0–10	Low entropy	Small, focused change in one directory, no conflicts
10–20	Moderate	Multi-directory change with a few dependencies
20–40	High	Cross-cutting change with dependencies or conflicts
40+	Very high	Large change with conflicts, many dependencies, fragmented graph
80+	Extreme	Near-maximum disorder — many factors elevated simultaneously

2.4 Diagnostic Thresholds

The system generates actionable diagnostics when entropy exceeds thresholds:

Threshold	Severity	Diagnostic code	Recommendation
`entropy_score > 20`	Medium	`diag.high_entropy`	"Reduce file count or dependencies before merging"

Threshold	Severity	Diagnostic code	Recommendation
`entropy_score > 40`	High (escalated)	`diag.high_entropy`	Same, but higher priority
`entropic_load > 50`	High	`diag.high_entropic_load`	“Reduce the number of files, directories, or dependencies touched”

Source: `risk/eval.py`, `_THRESHOLD_DIAGS` table

---

3. Entropy in the Risk Scoring System

Entropic load is one of four independent signals that compose the overall risk score for a change:

Signal	What it measures	Weight in risk_score	Weight in damage_score
Entropic Load	Disorder introduced by the change	30%	30%
Contextual Value	Importance of the files being modified (PageRank centrality)	25%	50%
Complexity Delta	Net change in system structural complexity	20%	—
Path Dependence	Sensitivity to merge ordering	25%	20%

3.1 Why Four Signals?

A single-metric risk score would conflate different kinds of risk. A change can be high-entropy but low-importance (touching many test fixtures), or low-entropy but high-importance (one-line change to a core authentication function). The four signals are designed to be orthogonal — each captures a different dimension of risk:

- Entropic Load: “How messy is this change?” (structural disorder)
- Contextual Value: “How important are the files it touches?” (criticality)
- Complexity Delta: “Does this make the system more complex?” (net complexity change)
- Path Dependence: “Does the merge order matter?” (ordering sensitivity)

3.2 Composite Formulas

Risk Score (overall risk, 0–100):

```
risk_score = entropic_load × 0.30 + contextual_value × 0.25 +
             complexity_delta × 0.20 + path_dependence × 0.25
```

Damage Score (potential harm if something goes wrong, 0–100):

```
damage_score = contextual_value × 0.50 + entropic_load × 0.30 +
               path_dependence × 0.20
```

Note that damage_score weights contextual value more heavily — because when damage occurs, it's the importance of the affected files that determines impact, not just the disorder.

3.3 Risk Level Classification

The composite risk score maps to a risk level:

Risk Level	Score Range
Low	0–24
Medium	25–49
High	50–74
Critical	75–100

Source: `defaults.py`, `RISK_CLASSIFICATION_THRESHOLDS`

---

4. The Entropy Gate: Policy Enforcement

Converge enforces five policy gates that a change must pass before it can merge. The Entropy Gate (Gate 3) is a hard blocker:

```
Gate passes if: entropy_delta ≤ entropy_budget
Gate blocks if: entropy_delta > entropy_budget
```

Each risk level has its own entropy budget — stricter for riskier changes:

Risk Level	Entropy Budget	Rationale
Low	25.0	Lenient — low-risk changes have high tolerance for disorder
Medium	18.0	Standard baseline
High	12.0	Restricted — high-risk changes must be focused
Critical	6.0	Very tight — critical changes must be surgically precise

Why different budgets? A low-risk change touching many files across directories is acceptable — the blast radius if something goes wrong is limited. But a critical-risk change (touching core infrastructure, targeting production branches) with high entropy signals a change that is both dangerous and unfocused. The budget ensures that as risk increases, entropy tolerance decreases.

4.1 Other Policy Gates (for context)

Gate	What it checks
1. Verification	Required CI checks (lint, tests) passed for this risk level
2. Containment	Change is structurally contained (containment_score ≥ threshold)
3. Entropy	Entropy within budget
4. Security	No critical/high security findings above threshold
5. Coherence	Coherence harness score above threshold

All five gates must pass for the policy verdict to be `ALLOW`. Any single gate blocking results in `BLOCK`.

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

---

5. Automatic Calibration

Entropy budgets are not static — they adapt to the team's actual patterns through data-driven calibration using historical data:

```
Given: sorted list of historical entropy_score values

P75 = value at the 75th percentile
P90 = value at the 90th percentile
P95 = value at the 95th percentile

low.budget     = max(P75 × 1.5,  10.0)   ← generous, based on typical behavior
medium.budget  = max(P75,        8.0)   ← baseline = what 75% of changes achieve
high.budget   = max(P90,        5.0)   ← tighter = only 10% of changes exceed this
critical.budget = max(P95 × 0.8,  3.0)   ← strictest = 95th percentile, tightened 20%
```

5.1 Why This Design?

Percentile-based: Instead of arbitrary thresholds, the budgets reflect what the team actually produces. A team that routinely makes small, focused changes will get tighter budgets. A team with a different workflow will get budgets that match their reality.

Minimum floors: The `max(..., floor)` ensures budgets never drop below safe minimums — even if a team's historical data shows very low entropy, the system maintains a minimum threshold to catch genuinely problematic changes.

Asymmetric tightening for critical: The 0.8 multiplier on P95 for critical-risk changes means the system is deliberately stricter than the team's historical worst-case. For critical changes, "what we've gotten away with before" isn't good enough.

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

---

6. System-Level Entropy

Beyond evaluating individual changes, Converge tracks entropy across the entire codebase over time to detect systemic trends.

6.1 Repo Health Score

The repo health score (0–100) incorporates entropy as one of three factors:

```
health_score = 100.0
health_score -= conflict_rate × 30        ← conflict impact
health_score -= min(avg_entropy, 50) × 0.5 ← entropy impact (capped at −25)
health_score -= min(rejected_count, 20) × 1.5  ← rejection impact (capped at −
30)
```

Why cap entropy at 50 before weighting? Without the cap, a single extreme-entropy change could dominate the health score. The cap ensures entropy can reduce health by at most 25 points (50 × 0.5), giving space for other signals to contribute.

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

6.2 Change Health Score

For individual changes, entropy contributes to a per-change health assessment:

```
change_health = 100.0
             − risk_score × 0.5       ← risk impact
             − entropy × 0.3          ← entropy impact
             − (30 if not mergeable)  ← conflict penalty
```

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

---

7. Trend Detection and Predictions

7.1 Entropy Spike Detection

Converge compares entropy between two 24-hour windows to detect sudden increases:

```
avg_now  = average entropy of last 24h
avg_prev = average entropy of previous 24h (24h–48h ago)

Spike detected if:
  avg_now > avg_prev × 1.2   AND   ← 20% increase
  avg_now > 15                AND   ← absolute floor
  sample_count > 3                  ← statistical significance
```

Why both relative AND absolute? The relative check (20% increase) catches sudden changes. The absolute floor (15) prevents false alarms — going from entropy 2 to 3 is a 50% increase but not concerning. Both conditions must be true.

Signal emitted: `entropy_spike` (severity: medium)

Source: `projections/predictions.py`, function `_detect_entropy_spike`

7.2 Entropy Velocity Projection

The health prediction system computes entropy velocity — the rate of change over time:

```
Split recent snapshots into older half and recent half

entropy_velocity = avg(recent_half) - avg(older_half)
projected_entropy = avg(recent_half) + entropy_velocity
```

Velocity	Severity	Signal
> 3.0	Medium	`predict.entropy_rising`
> 5.0	High	`predict.entropy_rising`

Why velocity matters: A single high-entropy change is a point event. But rising entropy across the team's changes signals a systemic shift — perhaps increased scope creep, loosening review standards, or growing codebase complexity that makes even small changes touch more files.

Source: `projections/health.py`, functions `_compute_velocities` and `_detect_health_signals`

7.3 Thermal Death Detection

The most severe entropy signal combines three system-level metrics:

```
Thermal death triggered if ALL are true:
  avg_entropy > 20         ← changes are consistently disordered
  conflict_rate > 20%      ← one in five changes has conflicts
  avg_propagation > 30     ← changes have wide blast radius
```

Why "thermal death"? The name comes from thermodynamics — the heat death of the universe is the state of maximum entropy where no useful work can be done. In a codebase, thermal death means the system has become so disordered that nearly every change conflicts with something, has a wide blast radius, and introduces more chaos. At this point, the system needs a coordinated halt: stop accepting new changes and focus on stabilizing.

Signal emitted: `bomb.thermal_death` (severity: critical) Recommendation: "Halt new intents — system entropy is approaching critical levels"

Source: `projections/predictions.py`, function `_detect_bomb_thermal`

---

8. Bomb Detection (Per-Change)

At the individual change level, Converge detects three structural degradation patterns ("bombs") that use entropy-related signals:

8.1 Cascade Bomb

Trigger: The change modifies files with high PageRank centrality (importance) that also fan out to many other nodes in the dependency graph, and the total blast radius exceeds 1.5× the number of files changed.

What this means: The change touches “load-bearing” files that many other files depend on. A bug here cascades.

8.2 Spiral Bomb

Trigger: The dependency graph contains 2+ cycles of length ≥ 2.

What this means: Circular dependencies — A depends on B depends on A. These create fragile feedback loops where a change to any node can propagate unpredictably through the cycle.

8.3 Thermal Death Bomb (Per-Change)

Trigger: 3+ out of 5 entropy indicators are elevated simultaneously:

Indicator	Threshold
Files changed	> 10
Merge conflicts	> 0
Dependencies	> 3
Graph components	> 3
Edge density	edges > nodes × 2

What this means: The change is “hot” across multiple dimensions — it’s large, has conflicts, depends on other changes, is fragmented, and has a dense dependency graph. This is the per-change version of the system-level thermal death signal.

Source: `risk/bombs.py`

---

9. The Dependency Graph

Entropy is not measured in isolation — it’s computed in the context of a dependency graph that models the structural relationships of the change. Understanding this graph is key to understanding why the entropy formula works.

9.1 Graph Construction

The graph is built from the intent and its merge simulation:

Node type	What it represents
`file`	A file modified by the change
`directory`	A directory containing modified files

Node type	What it represents
`scope`	A semantic boundary (e.g., “auth”, “payments”) declared in the intent
`dependency`	Another intent that this one depends on
`intent`	The change itself
`branch`	The merge target branch

Edge type	Weight	Meaning
`contained_in`	0.3	File → parent directory
`co_located`	0.2	Files in the same directory (bidirectional)
`scope_contains`	0.5	Scope → file that matches the scope name
`scope_touches`	0.2	Scope → file that doesn’t match
`depends_on`	0.8	Intent → dependency intent
`merge_target`	1.0	Intent → target branch
`co_change`	0.1–1.0	Historical co-change coupling (from archaeology data)

9.2 Why a Graph?

File lists are flat. A graph captures relationships: which files are co-located, which are in different architectural boundaries, which depend on each other through explicit or historical coupling. The graph enables:

- Component counting for entropy (fragmentation signal)
- PageRank for contextual value (file importance)
- Cycle detection for path dependence and spiral bombs
- Density for complexity delta

Source: `risk/graph.py`, function `build_dependency_graph`

---

10. Learning System

When entropy is high, Converge doesn’t just block — it generates structured learning with specific actions:

Condition	Lesson Code	Action
Repo avg entropy > 15	`learn.high_entropy`	“Split large intents into smaller focused changes”
Repo avg entropy > 30	`learn.high_entropy`	Same, priority elevated to P1

Condition	Lesson Code	Action
Change entropy > 20	`learn.change_entropy`	“Reduce scope or break into incremental, independently-mergeable changes”
Change entropy > 40	`learn.change_entropy`	Same, priority elevated to P1

Each lesson includes:

- Observed vs target metrics (machine-readable for agents)
- Why the metric matters
- What to do (specific next action)
- Priority (0 = critical, 1 = high, 2 = medium)

Source: `projections/learning.py`

---

11. End-to-End Flow

Here’s how entropy flows through the system from a developer’s push to a merge decision:

```
1. Developer pushes to feature branch
                    |
2. Intent created (source → target)
                    |
3. Merge simulation in isolated worktree
   → files_changed, conflicts detected
                    |
4. Dependency graph built
   → nodes, edges, components, density
                    |
5. Four risk signals computed:
   → entropic_load = f(files, conflicts, deps, dirs, components)
   → contextual_value = f(PageRank, core paths, target branch)
   → complexity_delta = f(density, edge ratio, cross-dir edges)
   → path_dependence = f(conflicts, core touches, cycles, longest path)
                    |
6. Composite scores:
   → risk_score = weighted combination of 4 signals
   → damage_score = weighted combination (importance-heavy)
   → entropy_score = entropic_load (direct)
                    |
7. Risk level classified (low/medium/high/critical)
                    |
8. Policy gates evaluated:
   → Gate 3 (Entropy): entropy_score ≤ budget[risk_level]?
   → Other gates: verification, containment, security, coherence
                         │
9. Verdict: ALLOW or BLOCK
                         │
10. Events recorded:
    → risk.evaluated (entropy_score stored in payload)
    → policy.evaluated (gate results stored)
                         │
11. System-level aggregation:
    → Repo health score (includes avg entropy)
    → Entropy trend (time series)
    → Spike/velocity/thermal death detection
    → Calibration of budgets from historical data
```

---

12. Design Rationale Summary

Design choice	Rationale
Weighted sum, not Shannon entropy	We need a metric that's interpretable, tunable, and directly connected to structural properties of a code change. Shannon entropy measures information content; we need disorder of a change.
Conflicts weighted 15× (highest)	Conflicts are the strongest empirical signal of disorder — they prove incompatibility with the current state.
Different budgets per risk level	High-risk changes to core infrastructure must be surgically focused. Low-risk changes can afford more scatter.
Graph-based component counting	A flat list of files misses fragmentation. The graph captures whether the change is one coherent unit or several unrelated modifications bundled together.
Calibration from percentiles	Hard-coded thresholds don't fit every team. Percentile-based calibration adapts to actual team behavior while maintaining safety floors.
System-level tracking	Individual change entropy is necessary but not sufficient. The system needs to detect when the trend is problematic, even if each individual change is within budget.
Capped contributions to health	Without caps, one extreme value dominates the health score. Caps ensure the health metric reflects a balanced view of multiple factors.

Design choice	Rationale
Velocity projections	A snapshot shows current state. Velocity shows direction. A system at entropy 18 and falling is healthy; a system at entropy 12 and rising fast may need attention soon.

---

13. File Reference

File	Role
`risk/signals.py`	Entropic load formula and the other three risk signals
`risk/eval.py`	Composite risk scoring, diagnostic generation
`risk/graph.py`	Dependency graph construction, propagation and containment scores
`risk/bombs.py`	Structural degradation pattern detection
`risk/_constants.py`	Core paths, target branches, severity ordering
`policy.py`	Policy gate evaluation (including entropy gate) and calibration
`defaults.py`	All default thresholds, entropy budgets, calibration constants
`projections/health.py`	Repo health score, change health, predictive health gate
`projections/predictions.py`	Entropy spike, thermal death, and other trend detectors
`projections/trends.py`	Entropy and risk time series endpoints
`projections/learning.py`	Structured learning with entropy-based lessons