# Risk Scoring in Converge: What, Why, and How

> This document explains how Converge computes risk scores for proposed changes using four independent signals, how those signals compose into aggregate scores, and how risk classification drives the entire policy engine. It is written for an external audience — no prior knowledge of the Converge internals is assumed.

---

## 1. What Problem Does Risk Scoring Solve?

Not all code changes carry the same risk. A one-line typo fix in a README is fundamentally different from a 50-file refactoring of the authentication system. But "how risky is this change?" is a multi-dimensional question:

- A change can be structurally disordered (many files, scattered across directories) without touching anything important
- A change can be surgically focused but target the single most critical file in the codebase
- A change can be simple by itself but create a complex web of dependencies with other pending changes
- A change can increase system complexity by adding coupling between previously independent modules

No single metric captures all of these. Converge uses four independent signals, each measuring a different dimension of risk, then combines them into composite scores that drive all downstream decisions.

---

## 2. The Four Independent Signals

### 2.1 Signal 1: Entropic Load (Disorder)

Question: "How much structural disorder does this change introduce?"

```
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)
```

| Factor | Weight | What It Measures |
|---|---|---|
| Files changed | 2.0 | Surface area of the change |
| Merge conflicts | 15.0 | Direct evidence of collision with current state |
| Dependencies on other intents | 6.0 | Ordering constraints and fragile chains |
| Directory spread | 3.0 | Architectural dispersion |
| Graph components - 1 | 5.0 | Fragmentation (bundled unrelated changes) |

Why these weights? Conflicts are weighted highest (15×) because they're the strongest empirical signal — a conflict proves the change doesn't compose cleanly. Files are weighted lowest (2×) because file count alone is a weak signal. See the entropy document for full rationale.

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

---

2.2 Signal 2: Contextual Value (Importance)

Question: "How important are the files being changed?"

A large refactoring of test fixtures has low contextual value. A one-line change to the core authentication function has high contextual value. This signal measures the criticality of what's being touched, not the size or complexity of the change itself.

```
contextual_value = clamp(
    min(importance_ratio × 30, 60)  +    ← PageRank importance (capped)
    core_ratio × 20                 +    ← fraction of files in core paths
    target_bonus                   +    ← 10 if targeting core branch, else 0
    risk_bonus                           ← 0/5/15/30 by current risk level
, 0, 100)
```

| Factor | How It's Computed | Maximum Contribution |
|---|---|---|
| Importance ratio | Sum of PageRank scores of changed files, divided by expected average PageRank per file. Measures how much more "central" the changed files are than average. | 60 (capped) |
| Core path ratio | Fraction of changed files that live in core paths (`src/`, `lib/`, `core/`, `pkg/`, `internal/`, `app/`). | 20 |
| Target branch bonus | 10 points if the target is a core branch (`main`, `master`, `release`, `production`, `prod`). | 10 |
| Risk level bonus | A static bonus based on the intent's current risk level: low=0, medium=5, high=15, critical=30. This creates a positive feedback loop — critical changes get more scrutiny. | 30 |

Why PageRank? PageRank measures node importance in a graph — nodes that are linked to by many other important nodes get high scores. In the dependency graph, a file that many other files depend on (directly or transitively) has high PageRank. Changes to high-PageRank files have outsized impact because many other parts of the system depend on them.

Source: `risk/signals.py`, function `compute_contextual_value`; `risk/_constants.py` for `_CORE_PATHS`, `_CORE_TARGETS`, `_RISK_BONUS`

---

### 2.3 Signal 3: Complexity Delta (Net Complexity Change)

Question: "Does this change make the system more or less complex?"

```
complexity_delta = clamp(
    density × 40                          +    ← graph density
    min(edge_node_ratio × 10, 30)         +    ← coupling density (capped)
    cross_dir_edges × 3                   +    ← cross-directory connections
    scope_count × 5                            ← semantic scopes affected
, 0, 100)
```

| Factor | Weight | What It Measures |
|---|---|---|
| Graph density | 40 | How interconnected the dependency graph is. A dense graph means many relationships between nodes — the change creates a tightly coupled cluster. |
| Edge-to-node ratio | 10 (cap 30) | Another coupling measure. Ratio > 2 means each node has, on average, more than 2 connections — a sign of high internal complexity. |
| Cross-directory edges | 3 | File-to-file edges that cross directory boundaries. Each one represents an architectural boundary violation. |
| Scope count | 5 | Number of semantic scopes the change touches. More scopes = wider architectural footprint. |

Why is this different from entropy? Entropy measures disorder (how scattered the change is). Complexity delta measures coupling (how interconnected the change makes the system). A change can be low-entropy (few files, one directory) but high-complexity (creates dense cross-dependencies). They're correlated but capture different failure modes.

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

---

### 2.4 Signal 4: Path Dependence (Ordering Sensitivity)

Question: "Does the merge order matter for this change?"

```
path_dependence = clamp(
    conflict_count × 20   +    ← direct ordering failure
    core_touches × 4      +    ← files in high-contention areas
    deps_count × 8        +    ← explicit ordering constraints
    cycle_count × 5       +    ← circular dependencies (capped at 20)
    longest_path × 2           ← dependency chain length
, 0, 100)
```

| Factor | Weight | What It Measures |
|---|---|---|
| Conflicts | 20 | The strongest signal — a conflict means merge order has already failed for at least one pairing. |
| Core path touches | 4 | Files in core paths (`src/`, `lib/`, etc.) are high-contention — other changes are likely to touch them too. |
| Dependency count | 8 | Each dependency creates an ordering constraint. More deps = more sensitive to execution order. |
| Cycle count | 5 | Circular dependencies in the graph. Cycles make ordering impossible to satisfy perfectly — any linearization breaks at least one cycle edge. Capped at 20 cycles to bound computation. |
| Longest DAG path | 2 | The longest path in the dependency DAG measures the maximum chain length. Longer chains are more fragile — any link failing blocks the entire chain. |

Why does path dependence matter? In a concurrent integration system, multiple changes merge in sequence. If change A must merge before change B, but the system processes B first, B's validation is wasted (it'll need re-validation after A merges, per Invariant 2). High path dependence means the system is spending compute on validations that are likely to be invalidated.

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

---

## 3. Why Four Signals?

A single "risk number" would conflate different kinds of risk. Consider:

| Change | Entropic Load | Contextual Value | Complexity Delta | Path Dependence |
|---|---|---|---|---|
| Large test refactor (50 files, one dir) | High | Low | Low | Low |
| One-line auth fix targeting main | Low | High | Low | Low |
| Cross-module depen-dency rewiring | Medium | Medium | High | Medium |

| Change | Entropic Load | Contextual Value | Complexity Delta | Path Dependence |
|---|---|---|---|---|
| Conflicting change with dep chain | Medium | Low | Low | High |

Each of these changes is "risky" for a different reason. The four signals are designed to be orthogonal — each captures a different dimension:

- 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?" (coupling)
- Path Dependence: "Does the merge order matter?" (ordering sensitivity)

---

## 4. Composite Scores

### 4.1 Risk Score

The overall risk score is a weighted average of the four signals:

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

Clamped to [0, 100].

| Weight | Signal | Rationale |
|---|---|---|
| 0.30 | Entropic Load | Disorder is the strongest predictor of integration problems. Gets the highest weight. |
| 0.25 | Contextual Value | Importance determines blast radius if something goes wrong. |
| 0.25 | Path Dependence | Ordering sensitivity causes wasted work and pipeline thrashing. |
| 0.20 | Complexity Delta | Coupling is important but partially captured by entropy and containment. Gets the lowest weight. |

Source: `risk/eval.py`, constants `_RISK_W_ENTROPIC`, `_RISK_W_CONTEXTUAL`, `_RISK_W_COMPLEXITY`, `_RISK_W_PATH_DEP`

### 4.2 Damage Score

A separate composite that answers: "If something goes wrong, how bad would it be?"

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

Key difference: Damage score weights contextual value at 50% (vs 25% in risk score). Why? When damage occurs, the importance of the affected files determines impact. A bug in a test fixture causes low damage; the same bug in core authentication causes high damage. Complexity delta is excluded because it measures potential (coupling could cause problems) rather than actual impact.

Source: `risk/eval.py`, constants `_DMG_W_CONTEXTUAL`, `_DMG_W_ENTROPIC`, `_DMG_W_PATH_DEP`

### 4.3 Entropy Score

```
entropy_score = entropic_load
```

The entropy score is simply the entropic load signal passed through directly. It feeds the Entropy Gate (Gate 3) in the policy engine.

---

## 5. Risk Level Classification

The composite risk score maps to a risk level that drives all policy decisions:

| Risk Level | Score Range | What It Means |
|---|---|---|
| Low | 0–24 | Routine change with minimal risk |
| Medium | 25–49 | Standard change requiring basic verification |
| High | 50–74 | Significant change requiring thorough verification |
| Critical | 75–100 | Major change requiring maximum scrutiny |

Source: `defaults.py`, `RISK_CLASSIFICATION_THRESHOLDS`

### 5.1 Auto-Reclassification

When the `risk_auto_classify` feature flag is enabled, the risk level is automatically updated based on computed scores:

```python
new_level = classify_risk_level(risk_eval.risk_score)
if new_level != intent.risk_level:
    intent.risk_level = new_level
    # Emit RISK_LEVEL_RECLASSIFIED event
```

This means an intent that was created as `medium` but computes to a risk score of 55 will be reclassified to `high` — and will then face the stricter policy profile for high-risk changes.

Source: `engine.py`, function `_evaluate_risk_step`

### 5.2 What Risk Level Controls

The risk level selects the policy profile that determines:

| Policy Element | Low | Medium | High | Critical |
|---|---|---|---|---|
| Required checks | lint | lint | lint, unit_tests | lint, unit_tests |
| Entropy budget | 25.0 | 18.0 | 12.0 | 6.0 |
| Containment min | 0.30 | 0.50 | 0.70 | 0.85 |
| Security max_high | 5 | 2 | 0 | 0 |
| Coherence pass | 75 | 75 | 80 | 85 |
| Coherence warn | 60 | 60 | 65 | 70 |

Source: `defaults.py`, `DEFAULT_PROFILES`

---

## 6. The Risk Gate

Beyond the five policy gates, there is a separate risk gate that evaluates the composite scores directly:

```
Breached if ANY of:
  risk_score       > max_risk_score       (default 65.0)
  damage_score     > max_damage_score     (default 60.0)
  propagation_score > max_propagation_score (default 55.0)
```

### 6.1 Gradual Enforcement (Canary Rollout)

The risk gate supports gradual enforcement to avoid blocking everything at once:

```
mode = "shadow"     # "shadow" or "enforce"
enforce_ratio = 1.0 # 0.0 to 1.0

bucket = sha256(intent_id)[:8] / 0xFFFFFFFF  # deterministic [0, 1)

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

| Mode | enforce_ratio | Behavior |
|---|---|---|
| `shadow` | any | Logs `would_block` but never enforces. Safe for observation. |
| `enforce` | 0.0 | Never enforces (equivalent to shadow) |
| `enforce` | 0.5 | Enforces for 50% of intents (deterministic by ID — same intent always in same bucket) |
| `enforce` | 1.0 | Enforces for all intents |

| Mode | enforce_ratio | Behavior |
|---|---|---|

Why deterministic bucketing? Using `sha256(intent_id)` ensures the same intent always lands in the same bucket. This prevents inconsistent behavior across retries — if intent X is in the enforcement group, it stays there on every attempt.

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

---

## 7. Diagnostics

The risk evaluation generates actionable diagnostics when scores exceed thresholds:

| Condition | Diagnostic Code | Base Severity | Escalation | Recommendation |
|---|---|---|---|---|
| `risk_score > 60` | `diag.high_risk` | high | critical at > 80 | “Split this change into smaller, independent intents” |
| `entropy_score > 20` | `diag.high_entropy` | medium | high at > 40 | “Reduce file count or dependencies before merging” |
| `propagation_score > 40` | `diag.high_propagation` | high | — | “Review impact graph and consider narrowing scope” |
| `containment_score < 0.4` | `diag.low_containment` | medium | — | “Add scope hints or reduce cross-boundary dependencies” |
| `entropic_load > 50` | `diag.high_entropic_load` | high | — | “Reduce the number of files, directories, or dependencies touched” |
| `contextual_value > 60` | `diag.high_contextual_value` | high | — | “Ensure thorough review — these files have high centrality” |
| `path_dependence > 40` | `diag.path_dependence` | medium | — | “Coordinate merge timing with related intents” |

Diagnostics are sorted by severity (critical first) and attached to the risk evaluation result.

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

---

## 8. Findings

Beyond diagnostics (which are threshold-based), the system generates findings from structural analysis:

| Condition | Finding Code | Severity |
|---|---|---|
| Files changed > 15 | `semantic.large_change` | high |
| Dependencies > 3 | `semantic.dependency_spread` | medium |
| Target is core branch | `semantic.core_target` | high |
| Merge conflicts > 0 | `semantic.merge_conflict` | critical |

Findings are less about computed scores and more about specific structural properties that humans should be aware of.

Source: `risk/eval.py`, function `analyze_findings`

---

## 9. The Full Risk Evaluation Pipeline

```
1. Intent + Simulation enter risk evaluation
                    |
2. Build dependency graph (NetworkX DiGraph)
   → Nodes: files, directories, scopes, dependencies, intent, branch
   → Edges: containment, co-location, scope, dependency, coupling
                    |
3. Compute graph metrics
   → PageRank (file importance)
   → Components (fragmentation)
   → Density (coupling)
                    |
4. Compute 4 independent signals:
   → entropic_load   = f(files, conflicts, deps, dirs, components)
   → contextual_value = f(PageRank, core paths, target, risk level)
   → complexity_delta = f(density, edge ratio, cross-dir, scopes)
  → path_dependence  = f(conflicts, core touches, deps, cycles, longest path)
                    |
5. Compute legacy scores (backwards compat):
   → propagation_score = f(graph out-degree, edge weights, unique targets)
   → containment_score = f(boundary crossings, components)
                    |
6. Detect structural bombs:
   → cascade (high-PageRank files with high fanout)
   → spiral (circular dependencies)
   → thermal_death (multiple indicators elevated)
                    |
7. Compute composite scores:
   → risk_score   = 0.30×EL + 0.25×CV + 0.20×CD + 0.25×PD
```

```
   → entropy_score = EL (direct)
   → damage_score  = 0.50×CV + 0.30×EL + 0.20×PD
                     |
8. Generate findings and diagnostics
                     |
9. Return RiskEval with all signals, scores, graph metrics, bombs,
   findings, impact edges — the complete risk assessment
```

Source: `risk/eval.py`, function `evaluate_risk`

---

## 10. Design Rationale Summary

| Design Choice | Rationale |
|---|---|
| Four orthogonal signals | A single risk number conflates different failure modes. Four signals let the system explain why something is risky, not just how much. |
| Weighted composition | Simple, interpretable, and tunable. More sophisticated models (ML, bayesian) would be harder to explain to users and harder to debug. |
| Entropic load at 30% (highest weight) | Disorder is the strongest empirical predictor of integration problems. It captures the most common failure mode: "the change is too big and too scattered." |
| Damage score separate from risk score | Risk = "how likely is a problem?" Damage = "how bad if a problem occurs?" These are different questions with different weight distributions. |
| PageRank for contextual value | PageRank is a well-understood, deterministic algorithm for graph centrality. It captures both direct and transitive importance without requiring manual file classification. |
| Core paths and targets | Some files and branches are categorically more important. Hard-coded core paths (`src/`, `lib/`) and targets (`main`, `production`) provide a reliable baseline that PageRank alone might miss for small graphs. |
| Risk level from score, not from labels | Human-assigned risk labels are subjective and inconsistent. Computing risk from objective signals and auto-reclassifying ensures the policy profile matches the actual risk. |

| Design Choice | Rationale |
|---|---|
| Gradual rollout for risk gate | New enforcement policies need safe rollout. Shadow mode collects data; gradual enforcement catches problems in a controlled fraction before full enforcement. |
| Deterministic bucketing | Same intent always in same bucket → consistent behavior across retries and re-evaluations. No randomness in the enforcement decision. |
| Diagnostic escalation | A medium-severity diagnostic at score 25 is informational. The same diagnostic at score 50 escalates to high — severity should match magnitude. |
| Clamped scores [0, 100] | Bounded, comparable, interpretable. Every score has the same scale, making dashboard displays and threshold comparisons straightforward. |

---

## 11. File Reference

| File | Role |
|---|---|
| `risk/signals.py` | Four independent signal functions and their constants |
| `risk/eval.py` | Composite scoring, risk classification, diagnostics, findings, full evaluation pipeline |
| `risk/graph.py` | Dependency graph construction, PageRank, propagation and containment scores |
| `risk/bombs.py` | Structural degradation pattern detection (cascade, spiral, thermal death) |
| `risk/_constants.py` | Core paths, core targets, risk bonus table, severity ordering |
| `policy.py` | Risk gate evaluation, rollout bucketing, policy profiles |
| `defaults.py` | Risk classification thresholds, risk gate defaults, policy profiles |
| `models.py` | `RiskEval` dataclass, `RiskLevel` enum |
| `engine.py` | Risk evaluation step in validation, auto-reclassification |