Containment in Converge: What, Why, and How

This document explains how Converge uses the Containment Gate (Gate 2) to measure and enforce how structurally isolated a change is. It is written for an external audience — no prior knowledge of the Converge internals is assumed.

---

1. What Problem Does Containment Solve?

When a change touches code in a single module, the blast radius is limited — if something goes wrong, only that module is affected. When a change reaches across multiple modules, scopes, and dependencies, the blast radius grows: a bug in one place can cascade to many others.

Containment measures how isolated a change is from the rest of the system. A perfectly contained change modifies files in one directory with no cross-boundary dependencies. A poorly contained change touches files across many boundaries, depends on other pending changes, and scatters across semantic scopes.

The Containment Gate blocks changes that are too spread out for their risk level. Higher-risk changes must be more contained — because when a high-risk change goes wrong, structural containment is the first line of defense against cascading damage.

---

2. The Containment Score Formula

2.1 The Formula

The containment score is a number between 0.0 (maximum spread) and 1.0 (perfectly contained):

```
containment_score = max(0.0,
    1.0
    - crossings × 0.05
    - (components - 1) × 0.03
)
```

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

2.2 What Each Term Means

Starting Point: 1.0 (Perfect Containment) Every change starts with a score of 1.0 — the assumption is that it's perfectly contained until evidence shows otherwise. Each penalty term then subtracts from this ideal.

Boundary Crossings (**`crossings × 0.05`**) What "crossings" means: A boundary crossing is any distinct entity outside the change itself that the change interacts with. Crossings are counted from three sources:

1. Impact edge targets: Every file, scope, dependency, or branch that the change reaches via its impact graph. These include:
   - Files modified (each is a target in the impact edge list)
   - The merge target branch
   - Dependent intents
   - Semantic scopes declared in the intent
2. Dependencies: Each intent this change depends on is a boundary crossing — the change can't merge independently.
3. Scope hints: Each semantic scope the change touches (e.g., "auth", "payments") is a boundary crossing.

All three sources are deduplicated into a set of unique boundary tokens before counting.

Why weight 0.05: Each crossing subtracts 5% from containment. A change with 10 crossings has 50% containment (1.0 - 0.50 = 0.50). This creates a smooth degradation: small changes with few crossings stay near 1.0; large cross-cutting changes quickly drop toward 0.

Example: A change that modifies 6 files, targets `main`, depends on 2 other intents, and touches 2 scopes has approximately 10+ boundary tokens → containment drops by at least 0.50.

Graph Fragmentation (**`(components - 1) × 0.03`**) What "components" means: The number of weakly connected components in the dependency graph built from the change. A connected graph (1 component) means all parts of the change are structurally related. Multiple components mean the change bundles unrelated modifications.

Why weight 0.03: A 3% penalty per extra component. Fragmentation is a secondary signal — it's less impactful than boundary crossings but still meaningful. A change with 5 disconnected components loses 12% (0.12) from fragmentation alone, which is enough to flag it for attention.

Why subtract 1: A single component (the expected case) contributes zero penalty. Only additional components beyond the first are penalized — having a connected graph is the baseline, not a bonus.

2.3 Interpretation Scale

Score	Interpretation	Typical change
1.0	Perfect containment	Empty change or zero crossings
0.8–1.0	Well contained	Small change in one module, no external deps
0.5–0.8	Moderate	Multi-directory change with a few dependencies
0.3–0.5	Poorly contained	Cross-cutting change across many boundaries

Score	Interpretation	Typical change
0.0–0.3	Very spread	Large change with many crossings and a fragmented graph

2.4 Edge Cases

- Empty graph and no edges: Returns 1.0 (no evidence of spread → assume contained)
- No crossings: Returns 1.0 minus only the component penalty (if any)
- Score floors at 0.0: The `max(0.0, ...)` prevents negative scores

---

3. The Dependency Graph

Containment is computed in the context of a dependency graph that models the structural relationships of the change. Understanding this graph is essential.

3.1 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
`scope`	A semantic boundary declared in the intent
`dependency`	Another intent 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)

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

3.2 Why a Graph for Containment?

A flat file list can count boundary crossings but can't detect fragmentation. The graph enables:

- Component counting: Are all parts of the change structurally related, or are there disconnected clusters?
- Boundary detection: Which directories, scopes, and dependencies does the change interact with?
- Coupling analysis: Historical co-change data reveals hidden relationships between files

---

4. The Containment Gate

4.1 Gate Logic

The Containment Gate (Gate 2) is a threshold comparison:

```
containment_min = profile[risk_level].containment_min

Gate passes if: containment_score >= containment_min
Gate blocks if: containment_score < containment_min
```

Source: `policy.py`, function `evaluate` (Gate 2 section)

4.2 Thresholds by Risk Level

Each risk level has its own minimum containment requirement:

Risk Level	containment_min	Rationale
Low	0.30	Very lenient — low-risk changes can be spread out
Medium	0.50	Moderate — at least half-contained
High	0.70	Strict — most boundary crossings must be avoided
Critical	0.85	Very strict — nearly perfect containment required

Source: `defaults.py`, `DEFAULT_PROFILES`

4.3 Why Different Thresholds?

The containment requirement scales with risk because blast radius matters more for high-risk changes:

- A low-risk change that modifies test fixtures across 5 directories is fine — even if it's spread out, the blast radius of a failure is limited (test files don't affect production).
- A critical-risk change to core authentication that also touches payment and logging boundaries is dangerous — if something goes wrong, the spread means multiple subsystems are affected simultaneously.

The gate ensures that as risk increases, the change must be more surgically focused.

4.4 Concrete Examples

Passing (high risk):

```
Change: 3 files in src/auth/, 1 dependency, 1 scope ("auth")
Boundary tokens: ~5 crossings
Components: 1 (connected)
Containment: 1.0 - (5 × 0.05) - (0 × 0.03) = 0.75
Threshold: 0.70 (high risk)
Result: PASS (0.75 >= 0.70)
```

Failing (high risk):

```
Change: 8 files across 4 directories, 3 dependencies, 2 scopes
Boundary tokens: ~13 crossings
Components: 2 (fragmented graph)
Containment: 1.0 - (13 × 0.05) - (1 × 0.03) = 0.32
Threshold: 0.70 (high risk)
Result: BLOCK (0.32 < 0.70)
```

---

5. Diagnostic Generation

When containment is low, Converge generates actionable diagnostics:

Threshold	Severity	Diagnostic Code	Recommendation
`containment_score < 0.4`	Medium	`diag.low_containment`	"Add scope hints or reduce cross-boundary dependencies"

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

This diagnostic fires independently of the gate — a change might pass the containment gate (e.g., low risk with 0.35 containment ≥ 0.30 threshold) but still get flagged diagnostically for being poorly contained. This provides early warning before the score drops further.

---

6. Impact Edges

The containment score depends on impact edges — a flat list of directed relationships that the change creates:

Edge Type	Source	Target	Weight
`merge_target`	source branch	target branch	1.0
`depends_on`	intent ID	dependency intent ID	0.8
`touches_scope`	intent ID	scope name	0.5
`modifies_file`	intent ID	file path	0.3

Impact edges serve two purposes:

1. Containment scoring: The unique targets of all impact edges form the set of boundary crossings
2. Propagation scoring: The total weight and target count feed into the propagation score (a related but separate metric)

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

6.1 File Limit

Only the first 20 files are included as `modifies_file` edges. This prevents a large refactoring change (touching 200 files) from generating an unwieldy edge list. The graph itself still contains all files — the limit only affects the flat impact edge list.

Source: `risk/graph.py`, constant `_IMPACT_FILES_LIMIT = 20`

---

7. Relationship to Other Metrics

Containment is one of several structural metrics Converge computes. Here's how it relates to the others:

Metric	What It Measures	Relationship to Containment
Entropic Load	Disorder of the change (files, conflicts, deps, dirs, components)	Shares the components signal but measures disorder, not isolation
Propagation Score	How far damage would spread if something goes wrong	Inverse correlation — low containment often means high propagation
Contextual Value	Importance of the files being changed	Independent — a well-contained change can touch critical files
Path Dependence	Sensitivity to merge ordering	Partially correlated — changes with many dependencies have both high path dependence and low containment

7.1 Containment in the Risk Score

Unlike the four primary risk signals (entropic load, contextual value, complexity delta, path dependence), containment is not a component of the composite risk score. It is computed independently and evaluated only through its own policy gate.

Why? Containment is a structural property, not a risk signal. A change can be poorly contained but low-risk (e.g., a cross-cutting test refactor). Including it in the risk score would conflate structural isolation with risk severity. Instead, it has its own gate with risk-level-appropriate thresholds.

---

8. Configuration Override

Containment thresholds can be customized per risk level in `.converge/policy.json`:

```json
{
  "profiles": {
    "low": { "containment_min": 0.2 },
    "medium": { "containment_min": 0.4 },
    "high": { "containment_min": 0.6 },
    "critical": { "containment_min": 0.9 }
  }
}
```

Teams working in monorepos where cross-directory changes are normal might lower thresholds. Teams with strict module boundaries might raise them.

---

9. End-to-End Flow

Here's how containment flows through the system:

```
1. Intent created (source → target, with dependencies and scope hints)
                         │
2. Merge simulation in isolated worktree
   → files_changed detected
                         │
3. Dependency graph built:
   → File nodes, directory nodes, scope nodes, dependency nodes
   → Edges: containment, co-location, scope, dependency, coupling
                         │
4. Impact edges built:
   → merge_target, depends_on, touches_scope, modifies_file
                         │
5. Containment score computed:
   → Count unique boundary tokens from edges + deps + scopes
   → Count weakly connected components in graph
   → Score = 1.0 − crossings×0.05 − (components−1)×0.03
                         │
```

```
6. Policy gate evaluated:
   → containment_score >= containment_min[risk_level]?
                     |
7. Result:
   → PASS: change is sufficiently contained
   → BLOCK: change is too spread out for its risk level
                     |
8. Diagnostic generated if score < 0.4 (independent of gate)
```

---

10. Design Rationale Summary

Design Choice	Rationale
Score 0–1 (not 0–100)	Containment is a ratio — “what fraction of perfect containment does this change achieve?” A 0–1 scale maps naturally to this interpretation.
Crossings weighted 0.05 (5% each)	Creates a smooth degradation curve. 20 crossings → 0.0 containment. This matches intuition: a change that reaches 20 distinct boundaries is maximally spread.
Components weighted 0.03 (3% each)	A secondary penalty — fragmentation is less important than total spread but still matters. A change with 10 extra components loses 30%, which is significant but doesn’t dominate.
Different thresholds per risk level	Blast radius matters more for high-risk changes. A low-risk change can tolerate more spread because the consequences of failure are limited.
Independent from risk score	Containment is a structural property, not a risk dimension. Mixing it into the composite risk score would conflate isolation with severity.
Boundary tokens are deduplicated	Prevents double-counting when the same entity appears as both a dependency and a scope hint.
Floor at 0.0	Negative containment is meaningless. The floor ensures the score stays interpretable.
Default 1.0 for empty changes	No evidence of spread → assume contained. This is the conservative default (don’t block something you can’t analyze).

11. File Reference

File	Role
`risk/graph.py`	Containment score formula, graph construction, impact edges, propagation score
`policy.py`	Gate 2 evaluation logic, threshold lookup from profiles
`defaults.py`	`DEFAULT_PROFILES` with `containment_min` per risk level
`risk/eval.py`	Diagnostic generation when containment < 0.4
`models.py`	`GateName.CONTAINMENT`, `GateResult`, `RiskEval.containment_score`