The Dependency Graph in Converge: What, Why, and How

This document explains how Converge builds and uses the dependency graph — the data structure that underpins risk scoring, containment analysis, bomb detection, and propagation measurement. It is written for an external audience — no prior knowledge of the Converge internals is assumed.

---

1. What Problem Does the Dependency Graph Solve?

A flat list of "files changed" tells you the size of a change. But size alone doesn't capture risk. What matters is the structural relationships between what's being changed:

- Are the files in the same directory (co-located) or scattered across the project?
- Are there historical patterns of co-change between these files?
- Does the change depend on other pending changes?
- Does it target a critical branch?
- How many disconnected clusters of work are bundled in this single change?

Converge builds a directed graph (using NetworkX) for each change that models all of these relationships. This graph is the foundation for computing:

Metric	What It Uses from the Graph
Entropic load	Component count (fragmentation)
Contextual value	PageRank (file importance by centrality)
Complexity delta	Density, edge-to-node ratio, cross-directory edges
Path dependence	Cycles, longest DAG path
Containment score	Boundary crossings, components
Propagation score	Average out-degree, edge weights, unique targets
Bomb detection	PageRank + fanout (cascade), cycles (spiral), multi-indicator (thermal)

Without the graph, all of these would need separate, ad-hoc calculations. The graph provides a unified model that feeds every structural metric.

---

2. Graph Construction

2.1 The Build Process

The graph is built from two inputs: the intent (what the change proposes) and the simulation (what the merge actually looks like):

```
G = build_dependency_graph(intent, simulation, coupling_data)
```

The function performs four steps in sequence:

1. File and directory nodes: Add a node for each changed file and its parent directory, with containment edges

2. Proximity coupling: Connect files that share a directory
3. Scope edges: Connect scope hints to files
4. Intent and dependency edges: Add the intent node, dependency nodes, and merge target

An optional fifth step adds historical coupling from archaeology data.

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

2.2 Node Types

Node Type	What It Represents	Added By
`file`	A file modified by the change	Step 1
`directory`	A parent directory of a modified file	Step 1
`scope`	A semantic boundary (e.g., “auth”, “payments”)	Step 3
`dependency`	Another intent this change depends on	Step 4
`intent`	The change itself	Step 4
`branch`	The merge target branch (e.g., “main”)	Step 4

Every node carries a `kind` attribute identifying its type, enabling type-specific queries (e.g., “find all file nodes”).

2.3 Edge Types

Edge Type	Source → Target	Weight	Meaning
`contained_in`	file → directory	0.3	File lives in this directory
`co_located`	file ↔ file	0.2	Both files live in the same directory (bidirectional)
`scope_contains`	scope → file	0.5	File name matches the scope name
`scope_touches`	scope → file	0.2	File doesn’t match the scope but is part of the change
`depends_on`	intent → dependency	0.8	This change depends on another change
`merge_target`	intent → branch	1.0	This change targets this branch
`co_change`	file ↔ file	0.1–1.0	Historical co-change coupling (bidirectional)

Source: `risk/graph.py`, weight constants `_WEIGHT_*`

---

3. Edge Weights: Why These Values?

Edge weights serve two purposes: they influence PageRank computation (heavier edges transfer more importance) and they contribute to propagation scoring (heavier edges indicate stronger coupling).

3.1 Merge Target (1.0 — highest)

The link between the intent and its target branch is the strongest relationship in the graph. This ensures that the target branch receives significant PageRank — files targeting `main` are inherently part of a more important context than files targeting a feature branch.

3.2 Dependency (0.8 — very high)

Dependencies are strong, explicit relationships. When intent A depends on intent B, this is a hard ordering constraint that directly affects merge sequencing. The high weight ensures dependencies dominate the graph structure.

3.3 Scope Contains (0.5 — medium-high)

When a file name matches a scope hint (e.g., scope "auth" matches file `src/auth/login.py`), this is a strong semantic relationship — the file is part of the named scope. The 0.5 weight gives it significant influence without overwhelming the graph.

3.4 Containment (0.3 — medium)

A file "contained in" its directory is a universal, structural relationship. It's always true and always meaningful, but it's not as significant as a dependency or scope match. The moderate weight ensures directory structure influences the graph without dominating.

3.5 Co-located (0.2 — low-medium)

Files in the same directory are related by proximity, but this is a weaker signal than scope matching or containment. Two files in `src/utils/` might be completely unrelated functionally.

3.6 Scope Touches (0.2 — low-medium)

When a file doesn't match a scope by name but is still part of the change, the scope "touches" it. This is a weak association — the file is included in the change but may not be semantically related to the scope.

3.7 Co-change (0.1–1.0 — dynamic)

Historical co-change coupling comes from archaeology data: "files A and B were changed together N times in the past." The weight scales linearly with the number of co-changes (capped at 1.0):

```
weight = min(1.0, co_changes × 0.1)
```

One co-change = 0.1 (weak signal). Ten or more co-changes = 1.0 (very strong signal — these files are tightly coupled by history).

---

4. Graph Metrics

After construction, the graph is analyzed for key metrics:

```
metrics = graph_metrics(G)
```

Metric	How It's Computed	What It Tells You
`nodes`	`len(G)`	Total elements in the change model
`edges`	`G.number_of_edges()`	Total relationships
`pagerank_max`	Highest PageRank value	How important the most central node is
`pagerank_top`	Top 5 nodes by PageRank	Which nodes are most central
`critical_files`	Top 5 file nodes by PageRank	Which files are most important
`components`	Weakly connected components	How fragmented the change is
`density`	`nx.density(G)`	How interconnected the graph is (0–1)

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

4.1 PageRank

Converge uses NetworkX's `nx.pagerank(G, weight="weight")` — the standard PageRank algorithm with edge weights. Key properties:

- Weighted: Heavier edges transfer more importance. A file connected to `main` via `merge_target` (weight 1.0) receives more importance than a file connected via `co_located` (weight 0.2).
- Transitive: Importance propagates through the graph. A file connected to an important file is itself more important, even without a direct high-weight edge.
- Normalized: PageRank values sum to 1.0 across all nodes. The `importance_ratio` in contextual value compares actual PageRank to the expected per-node average (1/N).

4.2 Weakly Connected Components

The graph is analyzed as undirected (ignoring edge direction) to find weakly connected components. This answers: "Are all parts of this change structurally related?"

Components	Meaning
1	Fully connected — all files/scopes/deps are reachable from each other
2	Two disconnected clusters — the change bundles two unrelated modifications
3+	Highly fragmented — the change should probably be split

Components feed into entropic load (penalty per extra component) and containment (penalty per extra component).

4.3 Density

Graph density measures how interconnected the nodes are:

```
density = actual_edges / possible_edges
```

A density of 0 means no edges (isolated nodes). A density of 1 means every node is connected to every other node. For typical changes:

Density	Meaning
< 0.1	Sparse — files are loosely related
0.1–0.3	Moderate — normal connectivity
0.3–0.5	Dense — many cross-connections
> 0.5	Very dense — tightly coupled cluster

Density feeds into complexity delta (weight 40 — the highest factor).

---

5. Impact Edges

In addition to the full graph, Converge builds a flat list of impact edges — a simplified representation used for containment scoring and backward compatibility:

```
edges = build_impact_edges(intent, simulation)
```

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

File limit: Only the first 20 files are included as `modifies_file` edges. This prevents a large change from generating an unwieldy edge list while the full graph retains all files.

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

5.1 Why Both a Graph and Flat Edges?

The graph (NetworkX DiGraph) is powerful but heavy — you need graph algorithms to query it. Impact edges are a lightweight, serializable representation that's easy to store in events, display in dashboards, and use for simple metric calculations (containment, propagation). They complement each other:

- Graph: Used for PageRank, components, density, cycles, longest path
- Impact edges: Used for containment (boundary crossings), propagation (weight sums), and API responses

---

6. Propagation Score

The propagation score measures how far damage could spread if something goes wrong:

```python
def propagation_score(G, edges) -> float:
```

It combines two components:

6.1 Graph Component

```
file_nodes = [nodes where kind == "file"]
avg_out = average out-degree of file nodes
graph_component = min(50, avg_out × 10)
```

What this measures: Average fan-out of changed files. A file with out-degree 5 connects to 5 other nodes — damage here could reach 5 places. Higher average fan-out = wider potential blast radius.

Cap at 50: Prevents the graph component from dominating the total score.

6.2 Edge Component

```
total_weight = sum of all edge weights
unique_targets = count of distinct edge targets
edge_component = min(50, total_weight × 3 + unique_targets × 2)
```

What this measures: Total coupling strength (weight sum) and breadth (unique targets). A change with many heavy edges to many distinct targets has high propagation potential.

Cap at 50: Same rationale — ensures balance between graph and edge components.

6.3 Total

```
propagation_score = min(100, graph_component + edge_component)
```

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

---

7. Containment Score

The containment score measures how isolated the change is from the rest of the system:

```
def containment_score(intent, G, edges) -> float:

boundary_tokens = set(edge targets + dependencies + scope hints)
crossings = len(boundary_tokens)

n_components = weakly_connected_components(G)
component_penalty = (n_components - 1) × 0.03

containment = max(0.0, 1.0 - crossings × 0.05 - component_penalty)
```

See the containment document for full explanation.

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

---

8. Historical Coupling (Archaeology)

When archaeology data is available, the graph gains co-change edges that capture historical relationships:

```
# For each pair (file_a, file_b) with co_changes count:
weight = min(1.0, co_changes × 0.1)
G.add_edge(file_a, file_b, rel="co_change", weight=weight)
G.add_edge(file_b, file_a, rel="co_change", weight=weight)
```

What this adds: If two files have been changed together 8 times in the past (weight 0.8), they're tightly coupled even if they're in different directories and different scopes. This coupling affects:

- PageRank: Co-changed files share importance
- Density: More edges increase density
- Components: Co-change edges may connect otherwise disconnected components
- Complexity delta: Cross-directory co-change edges count as architectural boundary violations

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

---

9. How the Graph Feeds Other Systems

9.1 Risk Signals

Signal	Graph Usage
Entropic load	`nx.number_weakly_connected_components(G)` → component count
Contextual value	`nx.pagerank(G, weight="weight")` → file importance ratios

Signal	Graph Usage
Complexity delta	`nx.density(G)`, edge-to-node ratio, cross-directory edge count
Path dependence	`nx.is_directed_acyclic_graph(G)`, `nx.simple_cycles(G)`, `nx.dag_longest_path_length(G)`

9.2 Bomb Detection

Bomb Type	Graph Usage
Cascade	PageRank to find high-centrality files, `nx.descendants(G, f)` for blast radius
Spiral	`nx.simple_cycles(G)` to find circular dependencies
Thermal death	Components count, edge density (edges > nodes × 2)

9.3 Metrics and Diagnostics

Output	Graph Usage
`graph_metrics`	Nodes, edges, density, PageRank top, components
`diag.low_containment`	Containment score (from boundary crossings + components)
`diag.high_propagation`	Propagation score (from out-degree + edge weights)

---

10. Graph Construction: Step by Step

Here's exactly what happens when the graph is built for a change that modifies 3 files in 2 directories, has 1 dependency, 1 scope hint, and targets `main`:

```
Intent: id="abc123", source="feature/x", target="main"
        dependencies=["dep001"], scope_hint=["auth"]
Simulation: files_changed=["src/auth/login.py", "src/auth/utils.py", "tests/test_auth.py
            conflicts=[]

Step 1: File and directory nodes
  Nodes added:
    "src/auth/login.py"  (kind=file)
    "src/auth/utils.py"  (kind=file)
    "tests/test_auth.py" (kind=file)
    "src/auth"           (kind=directory)
    "tests"              (kind=directory)
```

```
  Edges added:
    "src/auth/login.py"  → "src/auth"  (contained_in, 0.3)
    "src/auth/utils.py"  → "src/auth"  (contained_in, 0.3)
    "tests/test_auth.py" → "tests"     (contained_in, 0.3)

Step 2: Proximity coupling
  Edges added:
    "src/auth/login.py" ↔ "src/auth/utils.py" (co_located, 0.2 each way)
  (test_auth.py is alone in "tests/" — no co-location partner)

Step 3: Scope edges
  Nodes added:
    "auth" (kind=scope)
  Edges added:
    "auth" → "src/auth/login.py"  (scope_contains, 0.5)  ← "auth" in path
    "auth" → "src/auth/utils.py"  (scope_contains, 0.5)  ← "auth" in path
    "auth" → "tests/test_auth.py" (scope_contains, 0.5)  ← "auth" in path

Step 4: Intent and dependency edges
  Nodes added:
    "dep001"  (kind=dependency)
    "abc123"  (kind=intent)
    "main"    (kind=branch)
  Edges added:
    "abc123" → "dep001" (depends_on, 0.8)
    "abc123" → "main"   (merge_target, 1.0)

Result: 8 nodes, 10+ edges, 1 weakly connected component
```

---

11. Design Rationale Summary

Design Choice	Rationale
Directed graph (DiGraph)	Relationships have direction: a file is contained in a directory, not the other way around. Direction matters for PageRank propagation and reachability analysis.
Weighted edges	Not all relationships are equally strong. A dependency (0.8) is a harder constraint than co-location (0.2). Weights let PageRank and propagation distinguish strong from weak coupling.

Design Choice	Rationale
Multiple node types	A pure file graph would miss dependencies, scopes, and branch context. Multi-type nodes create a richer model that captures the full context of a change.
Co-located edges are bidirectional	If A and B are in the same directory, the relationship is symmetric. Both files share equal proximity.
Scope matching by name containment	`scope.lower() in file.lower()` is a simple heuristic that works for common naming conventions (scope "auth" matches file "src/auth/login.py"). No configuration required.
Historical coupling as optional enrichment	Not all teams have archaeology data. The graph works without it (structural edges only) and improves with it (coupling edges added).
Co-change weight scaling	`min(1.0, co_changes × 0.1)` creates a smooth curve: 1 co-change = weak signal, 10+ = strong. The cap at 1.0 prevents outlier pairs from dominating PageRank.
NetworkX	Mature, well-tested graph library with built-in PageRank, cycle detection, component counting, and path algorithms. No need to reimplement graph algorithms.
Impact edges as flat list	Lightweight serialization for events and API responses. Not every consumer needs full graph algorithms — many just need "what does this change touch?"
File limit (20) on impact edges	Prevents serialization bloat for large changes while the full graph retains all files internally.
Weakly connected for components	Ignoring edge direction for component counting is correct: "are these parts of the change related?" doesn't depend on which way the containment arrow points.

---

12. File Reference

File	Role
`risk/graph.py`	Graph construction, metrics, impact edges, propagation score, containment score
`risk/signals.py`	Four risk signals that consume graph metrics
`risk/eval.py`	Full evaluation pipeline that builds the graph and feeds it to signals/bombs
`risk/bombs.py`	Bomb detection using PageRank, descendants, cycles from the graph
`risk/_constants.py`	Core paths and targets used in contextual value computation
`models.py`	`RiskEval` stores `graph_metrics` and `impact_edges`