Coherence in Converge: What, Why, and How

> This document explains how Converge uses the Coherence Gate (Gate 5) to verify that a change maintains system-level structural coherence. It is written for an external audience — no prior knowledge of the Converge internals is assumed.

---

1. What Problem Does Coherence Solve?

The other four policy gates (Verification, Containment, Entropy, Security) evaluate a change in terms of the change itself — does it pass checks? Is it contained? Is it too disordered? Does it introduce vulnerabilities?

Coherence asks a different question: does this change preserve the structural health of the overall system?

A change can pass all four gates and still cause problems if it: - Deletes test files (reducing coverage) - Introduces TODOs and FIXMEs without resolving existing ones - Adds large binary files to the source tree - Reduces the test-to-source ratio below acceptable levels

These are systemic properties — they're not about the change in isolation, but about what the codebase looks like after the change is applied. The Coherence Gate evaluates configurable, shell-based questions against measurable assertions and baselines to catch this class of degradation.

---

2. The Coherence Harness

2.1 What Is It?

The coherence harness is a configurable set of questions — each question is a shell command that produces a numeric result, paired with an assertion about what that result should be. The harness is defined in a JSON configuration file at `.converge/coherence_harness.json`.

2.2 The Question Model

Each question is a `CoherenceQuestion`:

```python
@dataclass
class CoherenceQuestion:
    id: str                    # unique identifier (e.g., "q-test-count")
    question: str              # human-readable question
    check: str                 # shell command to run
    assertion: str             # comparison expression
    severity: str = "high"     # critical | high | medium
    category: str = "structural"  # structural | semantic | health
```

Source: `models.py`, class `CoherenceQuestion`

2.3 Default Questions

Converge ships with a template of five questions (three enabled by default):

ID	Question	Check Command	Assertion	Severity	Enabled
`q-test-count`	Has the test file count decreased?	`find tests/ -name 'test_*.py' | wc -l`	`result >= baseline`	high	Yes
`q-no-fixme-growth`	Has the TODO/FIXME count increased?	`grep -r 'TODO|FIXME' src/ --include='*.py' | wc -l`	`result <= baseline`	medium	Yes
`q-no-large-files`	Were files larger than 1MB added to source?	`find src/ -type f -size +1M | wc -l`	`result == 0`	high	Yes
`q-src-file-count`	Is the source file count stable?	`find src/ -name '*.py' | wc -l`	`result >= baseline`	medium	No
`q-test-ratio`	Is the test-to-source ratio adequate?	`echo $(( test_count * 100 / src_count ))`	`result >= baseline`	medium	No

Source: `coherence.py`, constant `HARNESS_TEMPLATE`

---

3. Question Execution

3.1 Running a Single Question

Each question is executed as an independent subprocess:

```
1. Run check command:
   subprocess.run(q.check, shell=True, cwd=workdir,
                  capture_output=True, text=True, timeout=60)
2. If returncode != 0 → verdict = "fail" with error details
3. If timeout → verdict = "fail" with "Command timed out"
4. If success → parse numeric result from stdout (last line)
5. Evaluate assertion against result and baseline
6. Return CoherenceResult with verdict, value, baseline, and assertion
```

Source: `coherence.py`, function `run_question`

3.2 Timeout (60 seconds)

Each question has a 60-second timeout. This is shorter than the 300-second check timeout because coherence questions are meant to be simple, fast measurements — counting files, grepping for patterns, computing ratios. A question that takes more than a minute is doing too much.

Source: `coherence.py`, constant `QUESTION_TIMEOUT_SECONDS = 60`

3.3 Parsing Numeric Results

The check command's stdout is parsed as a float:

```
def _parse_numeric(raw: str) -> float:
    lines = raw.strip().splitlines()
    last = lines[-1].strip()
    return float(last)
```

Why the last line? Some commands produce multi-line output where the numeric result is on the last line (e.g., `wc -l` after a pipeline). Taking the last line handles this robustly.

---

4. The Assertion System

4.1 Supported Assertion Forms

Assertions are simple comparison expressions evaluated without `eval()`:

Form	Example	Meaning
`result >= baseline`	Test count hasn't decreased	Current value is at least as high as the stored baseline
`result <= baseline`	TODO count hasn't increased	Current value is at most the stored baseline
`result > baseline`	Improvement required	Current value must exceed baseline
`result < baseline`	Reduction required	Current value must be below baseline
`result == 0`	Zero tolerance	Current value must be exactly zero
`result == <number>`	Exact match	Current value must match a specific number
`result >= <number>`	Floor value	Current value must be at or above a number
`result <= <number>`	Ceiling value	Current value must be at or below a number

Compound assertions using `AND` / `OR` are also supported:

```
result >= 0 AND result <= 100
result == 0 OR baseline == 0
```

Source: `coherence.py`, functions `_evaluate_assertion` and `_safe_eval_assertion`

4.2 Safety: No eval()

The assertion evaluator uses explicit parsing — splitting on operators and resolving tokens — rather than Python's `eval()`. This prevents code injection through malicious assertion strings in the harness config.

4.3 Baseline Handling

When an assertion references `baseline` but no baseline exists yet:

```
If "baseline" in assertion and baseline is None → pass (True)
```

Why? On the first run, there are no baselines. A question like "result >= baseline" would always fail if baselines were required. By returning True when baselines are missing, the system is permissive on first use and only starts enforcing after baselines are established.

---

5. Baselines

5.1 What Baselines Are

A baseline is the last known good value for a coherence question. When `q-test-count` last ran and produced 42, the baseline for `q-test-count` is 42. Next time, the assertion `result >= baseline` will check that the test count is still at least 42.

5.2 Baseline Storage

Baselines are stored as events in the event log:

```python
def update_baselines(results: list[CoherenceResult]) -> dict[str, float]:
    baselines = {r.question_id: r.value for r in results if r.error is None}
    event_log.append(Event(
        event_type = "COHERENCE_BASELINE_UPDATED",
        payload = {"baselines": baselines}
    ))
    return baselines
```

Loading: The most recent `COHERENCE_BASELINE_UPDATED` event is queried. Its payload contains the full baseline map.

Source: `coherence.py`, functions `load_baselines` and `update_baselines`

5.3 Why Event-Based Storage?

Using events for baselines means: - Audit trail: You can see when baselines changed and what they changed from - No separate storage: Baselines live in the same event log as everything else - Time travel: You can reconstruct the baseline state at any point in history

---

6. Scoring

6.1 The Scoring Formula

```
score = 100 − total_penalty
```

```
Where total_penalty = sum of weights for all failed questions
```

Clamped to [0, 100].

6.2 Severity Weights

Each failed question subtracts points based on its severity:

Severity	Weight (points lost)	Rationale
Critical	30	A critical failure significantly impacts the score — one critical failure drops from 100 to 70
High	20	A high failure is serious — two high failures drop from 100 to 60
Medium	10	A medium failure is notable — even three medium failures only drop from 100 to 70

Source: `coherence.py`, constant `SEVERITY_WEIGHTS`

6.3 Why This Weighting?

The weights create a clear severity gradient:

- One critical failure (30 points) has more impact than two medium failures (20 points)
- Two high failures (40 points) has more impact than one critical + one medium (40 points) — they're equal, which is intentional
- A mix of failures degrades the score proportionally — you can have several medium failures and still pass if there are no critical or high failures

The total possible penalty is bounded by the number and severity of configured questions. With the default 3 enabled questions (1 medium + 2 high), the maximum penalty is 50 (10 + 20 + 20), so the minimum possible score with defaults is 50.

6.4 Examples

Perfect score (all pass):

```
q-test-count: pass (0 points lost)
q-no-fixme-growth: pass (0 points lost)
q-no-large-files: pass (0 points lost)
Score: 100 - 0 = 100
```

One high failure:

```
q-test-count: fail (20 points lost — test count decreased)
q-no-fixme-growth: pass (0 points lost)
q-no-large-files: pass (0 points lost)
Score: 100 - 20 = 80
```

All fail:

```
q-test-count: fail (20 points lost)
q-no-fixme-growth: fail (10 points lost)
q-no-large-files: fail (20 points lost)
Score: 100 - 50 = 50
```

---

7. Verdict Determination

7.1 Three-Tier Verdict

The coherence score maps to one of three verdicts:

Verdict	Condition	Meaning
PASS	`score >= pass_threshold`	System coherence is maintained
WARN	`score >= warn_threshold AND < pass_threshold`	Coherence is degrading — flagged but not blocked
FAIL	`score < warn_threshold`	Coherence is compromised — gate blocks

7.2 Thresholds by Risk Level

Risk Level	Pass Threshold	Warn Threshold	Pass Zone	Warn Zone	Fail Zone
Low	75	60	75–100	60–74	0–59
Medium	75	60	75–100	60–74	0–59
High	80	65	80–100	65–79	0–64
Critical	85	70	85–100	70–84	0–69

Source: `defaults.py`, `DEFAULT_PROFILES` (coherence_pass, coherence_warn) and `COHERENCE_PASS_THRESHOLD`, `COHERENCE_WARN_THRESHOLD`

7.3 The Gate and the Verdict

In the policy gate, the coherence gate passes if the score is at or above the warn threshold (not the pass threshold). The warn zone is a "soft pass":

```
# Gate passes if score >= warn threshold
passed = coherence_score >= warn_threshold
```

Why? The engine handles escalation for warn verdicts separately — it creates a review request when the verdict is "warn." This allows the change to proceed through the pipeline while flagging it for human attention. Only a "fail" verdict (below the warn threshold) actually blocks.

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

---

8. Cross-Validation (Consistency Checks)

8.1 What Is Cross-Validation?

Even if the coherence harness passes, there may be inconsistencies between what the harness says and what the objective risk metrics show. Cross-validation detects these mismatches:

Check	Condition	Signal
Score mismatch	Coherence > 75 BUT risk_score > 50	Harness says "fine" but risk metrics say "elevated"
Bomb undetected	All questions pass BUT structural bombs detected	Degradation patterns exist that the harness doesn't cover
Missing scope validation	Propagation > 40 BUT no `q-scope-*` questions	High blast radius without scope-level checks

Source: `coherence.py`, function `check_consistency`

8.2 Why Cross-Validation Matters

The coherence harness is configurable — teams define their own questions. If a team's questions don't cover an important dimension, the harness can pass while the system is actually degrading. Cross-validation catches this gap by comparing the harness results against the objective risk metrics that Converge always computes.

8.3 Verdict Downgrade

When inconsistencies are detected, the verdict is downgraded:

Original Verdict	With Inconsistencies	Effect
Pass	Downgraded to Warn	Flagged for review but not blocked
Warn	Downgraded to Fail	Now blocks the change
Fail	Stays Fail	Already blocked

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

Why? Inconsistencies mean the harness is giving a misleading signal. If the harness says "pass" but risk metrics say "elevated," the truth is probably somewhere in between — hence "warn." If the harness already says "warn" and there are additional inconsistencies, the confidence in the change drops enough to justify blocking — hence "fail."

8.4 Events and Reviews

When inconsistencies are detected:

1. A `COHERENCE_INCONSISTENCY` event is emitted with details
2. A `COHERENCE_EVALUATED` event records the final verdict
3. If the verdict is "warn" or there are inconsistencies, a review request is automatically created via `reviews.request_review(intent_id, trigger="coherence")`

This ensures human attention for ambiguous cases without blocking the developer outright.

---

9. No Harness Configured

If no coherence harness file exists at `.converge/coherence_harness.json`:

```
if not questions:
    return CoherenceEvaluation(
        coherence_score=100.0,
        verdict="pass",
        results=[],
        harness_version="none",
    )
```

Effect: Score = 100, verdict = pass, gate passes. The system is backward compatible — teams that haven't configured a harness are not blocked by the coherence gate.

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

---

10. Initializing the Harness

Teams can bootstrap the coherence harness with:

```
coherence.init_harness()
```

This creates `.converge/coherence_harness.json` with the default template (5 questions, 3 enabled). If the file already exists, it's not overwritten.

The harness is versioned (currently `1.1.0`). The version is recorded in every `CoherenceEvaluation` for traceability — you can track which version of the harness was active when a particular change was evaluated.

---

11. The CoherenceResult and CoherenceEvaluation Models

11.1 Per-Question Result

```python
@dataclass
class CoherenceResult:
    question_id: str       # "q-test-count"
    question: str          # "Has the test file count decreased?"
    verdict: str           # "pass" | "warn" | "fail"
    value: float           # current measured value
    baseline: float | None # stored baseline (None if first run)
    assertion: str         # "result >= baseline"
    error: str | None      # error message if command failed
```

11.2 Aggregate Evaluation

```python
@dataclass
class CoherenceEvaluation:
    coherence_score: float                    # 0-100
    verdict: str                              # "pass" | "warn" | "fail"
    results: list[CoherenceResult]            # per-question results
    harness_version: str                      # "1.1.0"
    inconsistencies: list[dict[str, Any]]     # cross-validation findings
```

Source: `models.py`

---

12. End-to-End Flow

```
1. Intent enters validation pipeline
                    |
2. After risk evaluation completes (risk_eval available)
                    |
3. Load coherence questions from .converge/coherence_harness.json
   → If no file exists → score=100, verdict=pass, skip rest
                    |
4. Load baselines from most recent COHERENCE_BASELINE_UPDATED event
                    |
5. For each enabled question:
```

```
   a. Execute shell command (60s timeout)
   b. Parse numeric result from stdout
   c. Evaluate assertion against result and baseline
   d. Record CoherenceResult (pass/fail, value, baseline)
                    |
6. Calculate score:
   score = 100 - sum(severity_weight for each failed question)
                    |
7. Determine verdict:
   score >= pass_threshold → PASS
   score >= warn_threshold → WARN
   score < warn_threshold  → FAIL
                    |
8. Cross-validate against risk metrics:
   → score_mismatch? → inconsistency
   → bomb_undetected? → inconsistency
   → missing_scope_validation? → inconsistency
                    |
9. Apply verdict downgrades if inconsistencies found:
   pass → warn, warn → fail
                    |
10. Emit events:
    → COHERENCE_EVALUATED (always)
    → COHERENCE_INCONSISTENCY (if inconsistencies)
                    |
11. If verdict == "warn" or inconsistencies:
    → Auto-create review request
                    |
12. If verdict == "fail":
    → Block the change
                    |
13. Pass coherence_score to policy gate evaluation:
    → Gate 5: coherence_score >= warn_threshold?
```

---

13. Configuration

13.1 Harness Configuration

The harness file is fully customizable. Teams can add, remove, or modify questions:

```json
{
  "version": "1.2.0",
  "questions": [
    {
      "id": "q-test-count",
      "question": "Has the test file count decreased?",
      "check": "find tests/ -name 'test_*.py' | wc -l",
```

```
      "assertion": "result >= baseline",
      "severity": "high",
      "category": "structural",
      "enabled": true
    },
    {
      "id": "q-scope-api-coverage",
      "question": "Are API endpoints covered by contract tests?",
      "check": "find tests/contract/ -name '*.py' | wc -l",
      "assertion": "result >= 5",
      "severity": "critical",
      "category": "semantic",
      "enabled": true
    }
  ]
}
```

Questions with `"enabled": false` are skipped. This allows teams to experiment with new questions without immediately enforcing them.

13.2 Policy Thresholds

Pass and warn thresholds can be overridden per risk level in `.converge/policy.json`:

```
{
  "profiles": {
    "low": { "coherence_pass": 70, "coherence_warn": 50 },
    "high": { "coherence_pass": 90, "coherence_warn": 75 },
    "critical": { "coherence_pass": 95, "coherence_warn": 80 }
  }
}
```

---

14. Design Rationale Summary

Design Choice	Rationale
Shell-based checks	Universal — any measurable property of the codebase can be a coherence question. No need for Converge to understand every possible metric; just run a command and read a number.
Configurable questions	Different projects have different invariants. A web app cares about API coverage; a library cares about backwards compatibility. The harness adapts to the project.

Design Choice	Rationale
Severity-weighted scoring	Not all questions are equally important. Losing test coverage (high) matters more than FIXME growth (medium). Weights ensure the score reflects actual impact.
Three-tier verdict (pass/warn/fail)	Binary pass/fail is too rigid. The warn zone creates a space for “this is concerning but not blocking” — paired with automatic review creation for human judgment.
Cross-validation against risk metrics	The harness is user-configured and may have gaps. Cross-validation catches cases where the harness gives a misleading “pass” by comparing against objective metrics.
Verdict downgrade on inconsistencies	Inconsistencies mean the harness is not telling the full story. Downgrading the verdict adds appropriate caution without completely overriding the harness.
Auto-review on warn	Warn verdicts and inconsistencies get human attention without blocking the pipeline. This balances automation with human oversight.
Baseline-based assertions	Absolute thresholds (e.g., “must have > 50 tests”) are fragile across projects. Relative assertions (“don’t decrease from baseline”) adapt naturally to any codebase.
Graceful absence (score=100 if no harness)	Teams that haven’t configured a harness aren’t penalized. The gate is opt-in for enforcement, ensuring backward compatibility.
Event-based baselines	Baselines are auditable, time-travellable, and don’t require separate storage. They fit naturally into the event log architecture.
60-second question timeout	Coherence questions should be fast, simple measurements. A long-running question signals overengineering.
No eval()	The assertion evaluator parses comparisons explicitly, preventing code injection through malicious harness configurations.

---

15. File Reference

File	Role
`coherence.py`	Question loading, execution, scoring, baselines, cross-validation, harness init
`policy.py`	Gate 5 evaluation logic (`_evaluate_coherence_gate`)
`engine.py`	Coherence step in validation flow (`_evaluate_coherence_step`), verdict downgrades, review creation
`defaults.py`	`COHERENCE_PASS_THRESHOLD`, `COHERENCE_WARN_THRESHOLD`, per-level thresholds in `DEFAULT_PROFILES`
`models.py`	`CoherenceQuestion`, `CoherenceResult`, `CoherenceEvaluation`, `CoherenceVerdict`, `GateName.COHERENCE`