Verification in Converge: What, Why, and How

> This document explains how Converge uses the Verification Gate (Gate 1) to ensure that required CI checks pass before a change can be integrated. It is written for an external audience — no prior knowledge of the Converge internals is assumed.

---

1. What Problem Does Verification Solve?

Before merging any change into a shared codebase, you want machine evidence that it doesn't break things. "It works on my machine" is not sufficient — every change must pass a defined set of automated checks (linting, tests, security scans) that prove it meets a baseline quality bar.

The Verification Gate enforces this: a change cannot proceed through the merge pipeline unless every check required for its risk level has passed. This is the first gate evaluated, and it acts as a hard prerequisite — no amount of low entropy, good containment, or clean security findings can bypass a failed lint or test run.

---

2. Supported Checks

Converge defines a fixed vocabulary of five check types:

Check Type	Make Target	What It Proves
`lint`	`make lint`	Code style and static analysis rules are satisfied
`unit_tests`	`make test`	Unit-level logic works correctly in isolation
`integration_tests`	`make test-integration`	Components interact correctly end-to-end
`security_scan`	`make security-scan`	No known security issues detected by static tooling
`contract_tests`	`make test-contract`	API and interface contracts are not violated

Source: `engine.py`, constant `SUPPORTED_CHECKS`

2.1 Why Make Targets?

Converge delegates to `make` because it provides a universal interface regardless of the underlying toolchain. Whether a project uses pytest, jest, cargo, or a custom script, the check is invoked the same way: `make <target>`. This keeps Converge agnostic to the specific build system while giving projects full control over what each check actually runs.

Any check type not in `SUPPORTED_CHECKS` is silently skipped — the system will not execute arbitrary commands.

---

3. Check Execution

3.1 The Process

Each check runs as a subprocess with strict boundaries:

```
For each required check:
  1. Look up the command: check_commands[check_type] → ["make", "<target>"]
  2. Run as subprocess with:
     - capture_output=True (stdout + stderr captured)
     - text=True (string output, not bytes)
     - timeout=300 seconds
  3. Determine pass/fail:
     - returncode == 0 → passed, capture stdout (first 2000 chars)
     - returncode != 0 → failed, capture stderr (first 2000 chars)
  4. Record a CheckResult(check_type, passed, details)
  5. Emit a CHECK_COMPLETED event with the result
```

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

3.2 Timeout (300 seconds)

What: Every check has a hard timeout of 300 seconds (5 minutes).

Why this value: Long enough for a reasonable test suite or security scan to complete, but short enough to prevent runaway processes from blocking the pipeline. A test suite that takes more than 5 minutes on a change that is mid-pipeline is already a signal that something is wrong — either the suite is too large for a per-change gate, or the change is touching too much.

What happens on timeout: The check is recorded as failed with `details` set to the `TimeoutExpired` exception message. The change is blocked.

Source: `defaults.py`, constant `CHECK_TIMEOUT_SECONDS = 300`

3.3 Output Limit (2000 characters)

What: Only the first 2000 characters of check output are captured and stored.

Why: Check output is stored in events (persisted to SQLite). Without a cap, a verbose test suite could generate megabytes of output per run, bloating the event log and making diagnostics unreadable. 2000 characters is enough to see which tests failed or which lint rules were violated, without storing the entire CI log.

Source: `defaults.py`, constant `CHECK_OUTPUT_LIMIT = 2000`

3.4 The CheckResult Model

Every check execution produces a `CheckResult`:

```python
@dataclass
class CheckResult:
    check_type: str     # "lint", "unit_tests", etc.
    passed: bool        # True if returncode == 0
```

```
    details: str        # stdout (if passed) or stderr (if failed), truncated
    duration_ms: int    # execution time in milliseconds
```

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

---

4. Which Checks Are Required? Risk-Level Profiles

Not every change needs every check. Converge assigns required checks based on the change's risk level:

Risk Level	Required Checks	Rationale
Low	`lint`	Low-risk changes only need to pass basic style/static analysis
Medium	`lint`	Same as low — medium risk doesn't yet warrant running the full test suite
High	`lint`, `unit_tests`	High-risk changes must prove they don't break core logic
Critical	`lint`, `unit_tests`	Critical changes get the same as high (additional gates like security and coherence provide further scrutiny)

Source: `defaults.py`, constant `DEFAULT_PROFILES`

4.1 Why Not Require All Checks for Everything?

Running `integration_tests` and `contract_tests` on every low-risk change would be wasteful. A one-file documentation fix doesn't need an integration test suite. The tiered approach ensures:

- Speed: Low-risk changes move fast through the pipeline
- Focus: High-risk changes get more scrutiny where it matters
- Customizability: Teams can override profiles in `policy.json` to add checks at any level

4.2 How Risk Level Is Determined

The risk level comes from the risk evaluation step that runs before the policy gates. It is computed from four independent signals (entropic load, contextual value, complexity delta, path dependence) and classified into a level:

Risk Score	Risk Level
0–24	Low

Risk Score	Risk Level
25–49	Medium
50–74	High
75–100	Critical

This means: the more structurally risky a change is, the more checks it must pass. A focused one-file change in a non-critical directory → low risk → lint only. A cross-cutting change to core infrastructure → high risk → lint + tests.

---

5. The Gate Evaluation

5.1 The Logic

The Verification Gate (Gate 1) is the simplest gate in the policy engine:

```
required_checks = profile[risk_level].checks
missing = [c for c in required_checks if c not in checks_passed]

Gate passes if: missing is empty (all required checks passed)
Gate blocks if: missing is not empty
```

Source: `policy.py`, function `evaluate` (Gate 1 section)

5.2 The GateResult

The gate produces a `GateResult` with machine-readable fields:

Field	Value
`gate`	`GateName.VERIFICATION`
`passed`	`True` if no missing checks, `False` otherwise
`reason`	`"All required checks passed"` or `"Missing checks: ['unit_tests']"`
`value`	Number of checks that passed
`threshold`	Number of checks required

Example — passing:

```
Gate: verification
Passed: true
Reason: "All required checks passed"
Value: 2 (lint, unit_tests both passed)
Threshold: 2 (2 checks required for high risk)
```

Example — failing:

```
Gate: verification
Passed: false
Reason: "Missing checks: ['unit_tests']"
Value: 1 (only lint passed)
Threshold: 2 (2 checks required for high risk)
```

---

6. Event Recording

Every check execution emits a `CHECK_COMPLETED` event to the append-only event log:

```
Event(
    event_type = "CHECK_COMPLETED",
    trace_id   = <current trace>,
    intent_id  = <the intent being validated>,
    tenant_id  = <tenant context>,
    payload    = {
        "check_type": "lint",
        "passed": true,
        "details": "... first 2000 chars of output ..."
    },
    evidence   = {
        "check_type": "lint",
        "passed": true
    }
)
```

Why record events? Three reasons:

1. Audit trail: Every decision is traceable. You can answer "why was this change blocked?" by querying events.
2. Diagnostics: The `details` field captures the actual output — which tests failed, which lint rules were violated.
3. Analytics: Historical check data feeds calibration and health scoring. If 30% of intents fail `unit_tests`, that's a signal worth tracking.

---

7. Retry Behavior

When a change fails verification, it doesn't immediately die. The queue engine implements bounded retry (Invariant 3):

```
On verification failure:
  1. Increment intent.retries
  2. If retries >= MAX_RETRIES (3):
       → Status = REJECTED
       → Emit INTENT_REJECTED event
  3. Else:
       → Status = READY (back to queue)
```

```
    → Emit INTENT_REQUEUED event
```

Why 3 retries? A flaky test might fail once. A transient infrastructure issue might fail twice. But if a change fails verification three times, it has a real problem that needs human attention. Three retries balances tolerance for flakiness against the cost of repeated pipeline runs.

Source: `defaults.py`, constant `MAX_RETRIES = 3`; `engine.py`, function `_handle_blocked_intent`

---

8. Configuration Override

The default profiles can be overridden via a JSON file at `.converge/policy.json` (or `policy.json` or `policy.default.json`):

```json
{
  "profiles": {
    "low": {
      "checks": ["lint"]
    },
    "medium": {
      "checks": ["lint", "unit_tests"]
    },
    "high": {
      "checks": ["lint", "unit_tests", "integration_tests"]
    },
    "critical": {
      "checks": ["lint", "unit_tests", "integration_tests", "contract_tests"]
    }
  }
}
```

This allows teams to customize the check requirements per risk level without modifying Converge source code. The override merges with defaults — only specified fields are replaced.

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

8.1 Origin-Based Overrides

Changes can also have different requirements based on their origin type (human, agent, or integration). For example, a team might require stricter checks for agent-generated changes:

```json
{
  "origin_overrides": {
    "agent": {
      "medium": { "checks": ["lint", "unit_tests"] },
      "high": { "checks": ["lint", "unit_tests", "integration_tests", "security_scan"]
    }
  }
}
```

Why? Agent-generated changes may have different quality characteristics than human-written code. Some teams want to apply stricter verification to automated changes as a safety measure.

---

9. Integration in the Validation Flow

Verification is Step 2 in the full validation pipeline:

```
1. Simulation     — Can the change merge cleanly?
                    |
2. Verification   — Do required checks pass?     ← THIS GATE
                    |
3. Risk Evaluation — Compute risk scores and signals
                    |
4. Coherence      — Does the change maintain system coherence?
                    |
5. Policy Gates   — All 5 gates evaluated together:
                      Gate 1: Verification (checks passed)
                      Gate 2: Containment (change is contained)
                      Gate 3: Entropy (entropy within budget)
                      Gate 4: Security (no critical findings)
                      Gate 5: Coherence (coherence score above threshold)
                    |
6. Risk Gate      — Composite risk score within limits
                    |
7. Verdict        — ALLOW or BLOCK
```

If verification fails (Step 2), the pipeline short-circuits — the change is immediately blocked without evaluating risk, coherence, or the other policy gates. This is intentional: if basic checks fail, there's no point computing entropy or containment.

Source: `engine.py`, function `validate_intent` → `_run_validation_checks`

---

10. The Invariant

Verification enforces Invariant 1 (part 2) of Converge's core engine:

`mergeable(i, t) = can_merge(M(t), Δi) ∧ checks_pass`

In plain language: An intent `i` is mergeable into target `t` if and only if: - The merge simulation succeeds (no conflicts), AND - All required checks pass for the intent's risk level

Both conditions must be true. This is the fundamental contract of the system — no change can bypass simulation or verification.

Source: `engine.py`, docstring and `validate_intent`

---

11. Design Rationale Summary

Design choice	Rationale
Make targets as interface	Universal abstraction over build systems. Projects control what each check actually runs; Converge controls which checks are required.
Risk-tiered requirements	Low-risk changes shouldn't bear the cost of full test suites. High-risk changes need more evidence. Proportional enforcement.
Hard 300s timeout	Prevents runaway processes from blocking the pipeline. Forces projects to keep per-change checks fast.
2000-char output cap	Stores enough diagnostic information for debugging without bloating the event log.
Bounded retry (3)	Tolerates transient failures (flaky tests, infra issues) without letting genuinely broken changes cycle indefinitely.
Gate evaluates missing checks, not failed	A check not in `checks_passed` is missing — it either wasn't run or failed. The gate doesn't distinguish, which simplifies the logic.
Short-circuit on failure	No point evaluating entropy/containment if basic checks fail. Saves compute and provides clear, fast feedback.
Event recording per check	Creates a complete audit trail. Every check execution is traceable by intent, trace, and timestamp.

---

12. File Reference

File	Role
`engine.py`	Check execution (`run_checks`), supported checks constant, validation flow
`policy.py`	Gate 1 evaluation logic, profile loading
`defaults.py`	`CHECK_TIMEOUT_SECONDS`, `CHECK_OUTPUT_LIMIT`, `DEFAULT_PROFILES` with per-level checks
`models.py`	`CheckResult` dataclass, `GateName.VERIFICATION`, `GateResult`