Intents in Converge: What, Why, and How

This document explains what an intent is, why this abstraction exists, and how intents flow through the Converge system from creation to merge or rejection. It is written for an external audience — no prior knowledge of the Converge internals is assumed.

---

1. What Problem Do Intents Solve?

In traditional Git workflows, the unit of integration is a pull request or merge request — a request from a developer to merge their branch. PRs are informal: they carry a title, a description, maybe some labels, but they don't encode the structural properties of the change in a machine-readable way.

Converge introduces the intent: a semantic contract that represents a proposed change to a codebase. An intent is not a PR. It's a richer object that carries:

- The source and target branches
- Risk classification
- Priority ordering
- Semantic meaning (what problem it solves, what objective it pursues)
- Technical metadata (scope hints, dependency constraints)
- Dependencies on other intents
- Audit trail (who created it, when, under what trace)

Why does this matter? Because Converge needs to make automated decisions about whether a change can merge — and those decisions require structured, queryable data that PRs don't provide. An intent is the contract between the developer (or agent) and the integration engine.

---

2. The Intent Data Model

```
@dataclass
class Intent:
    id: str                         # unique identifier (12-char hex)
    source: str                     # source branch name
    target: str                     # target branch name
    status: Status                  # lifecycle state (see §3)
    created_at: str                 # ISO 8601 timestamp
    created_by: str                 # actor identity ("system", user, agent)
    risk_level: RiskLevel           # low | medium | high | critical
    priority: int                   # 1 (highest) to 5 (lowest), default 3
    semantic: dict                  # problem, objective, rationale
    technical: dict                 # scope_hint, affected_modules
    checks_required: list[str]      # explicit check requirements
    dependencies: list[str]         # IDs of intents that must merge first
    retries: int                    # number of times revalidation failed
    tenant_id: str | None           # multi-tenant isolation key
    plan_id: str | None              # groups related intents
    origin_type: str                 # "human" | "agent" | "integration"
```

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

2.1 Key Fields Explained

`id` — Identity A 12-character hex string generated from `uuid4()`. Short enough to be human-readable in logs and dashboards, long enough to be unique across millions of intents.

`source` and **`target`** — What Goes Where The source branch contains the changes; the target branch is where they will land. Converge simulates the merge of `source` into `target` in an isolated worktree, then evaluates whether the result is safe to integrate.

`risk_level` — How Dangerous Is This Change? Starts as `medium` by default. Can be set explicitly at creation time, or auto-reclassified by the risk evaluation engine based on computed scores:

Risk Score	Risk Level
0–24	Low
25–49	Medium
50–74	High
75–100	Critical

The risk level determines which policy profile is used — and therefore which checks are required, what entropy budget applies, how strict containment must be, and what security thresholds are enforced.

`priority` — Queue Ordering An integer from 1 (highest) to 5 (lowest). When the queue engine processes validated intents, higher-priority intents go first. Two intents at the same priority are processed in creation order.

`dependencies` — Ordering Constraints A list of intent IDs that must reach `MERGED` status before this intent can be processed. If intent A depends on B and C, the queue engine will skip A until both B and C are merged.

Why? In real codebases, changes have logical dependencies. Feature Y builds on Feature X. Converge makes these dependencies explicit and machine-enforceable, rather than relying on developers to coordinate merge order manually.

`semantic` — What and Why A free-form dictionary carrying human-readable context:

```
{
  "problem": "Login endpoint returns 500 on empty password",
  "objective": "Validate password field before auth attempt",
  "rationale": "Prevents crash and improves error messaging"
}
```

This information flows into dashboards and events for audit purposes. It doesn't affect automated decisions, but it gives humans (and agents) context about why a change exists.

`technical` — Machine-Readable Metadata A dictionary with structured metadata:

```
{
  "scope_hint": ["auth", "validation"],
  "affected_modules": ["src/auth/login.py", "tests/test_auth.py"]
}
```

`scope_hint` feeds directly into the dependency graph construction — scopes become nodes with edges to files, affecting containment and propagation scores.

`origin_type` — Who Created This? Three values: `human`, `agent`, `integration`. This field enables origin-based policy overrides — teams can apply different check requirements or thresholds depending on whether a change was made by a person, an AI agent, or a CI/CD integration.

`plan_id` — Grouping Related Intents When multiple intents are part of a coordinated plan (e.g., "migrate auth system in 5 steps"), they share a `plan_id`. The queue engine uses this for dependency resolution: if intent A's dependency B has the same `plan_id`, the relationship is visible in diagnostics and events.

---

3. The Intent Lifecycle

An intent progresses through a strict state machine:

$$\begin{tikzcd}
\text{DRAFT} \arrow[r] & \text{READY} \arrow[r] \arrow[d] & \text{VALIDATED} \arrow[r] \arrow[d] & \text{QUEUED} \arrow[r] & \text{MERGED} \arrow[d] \\
 & \text{REJECTED} & \text{REJECTED} & & \text{REJECTED}
\end{tikzcd}$$

Status	Meaning	How It Gets Here
DRAFT	Created but not yet submitted for processing	Initial creation (optional — some flows skip to READY)
READY	Submitted and awaiting validation	Explicit transition or default on creation
VALIDATED	Passed simulation, checks, risk evaluation, and all policy gates	`validate_intent` succeeds
QUEUED	In the merge queue, awaiting execution	Queue engine picks it up
MERGED	Successfully integrated into target branch	`execute_merge` succeeds or `confirm_merge` called
REJECTED	Permanently blocked	Max retries exceeded, review rejected, or policy block

3.1 Why These Specific States?

Each state represents a distinct phase of evidence gathering:

- READY: “I want to merge this” — an intent of action, no evidence yet
- VALIDATED: “The system has confirmed this is safe” — simulation passed, checks passed, risk acceptable, policy gates cleared
- QUEUED: “This is waiting to be executed” — validated and ready for the actual merge
- MERGED: “This has been integrated” — the git merge was performed successfully

The separation between VALIDATED and QUEUED exists because validation is expensive (simulation, checks, risk evaluation) and merging is a state-changing operation. Separating them allows the queue engine to re-validate before merging (Invariant 2) without losing the original validation result.

---

4. The Three Invariants

The core engine enforces three invariants that govern intent processing. These are the fundamental contracts of the system:

Invariant 1: Mergeability

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

In plain language: An intent `i` targeting branch `t` is mergeable if and only if: 1. The merge simulation succeeds (no conflicts between the current state of `t` and the change `Δi`) 2. All required checks pass for the intent’s risk level

Both conditions must be true. This is enforced by `validate_intent`.

Invariant 2: Revalidation on State Change

```
If M(t) advances → revalidate before merging
```

In plain language: If the target branch `t` has changed since the last validation (because another intent merged into it), the intent must be revalidated before it can merge. This prevents “stale validations” — a change that was safe against the old state of `main` might conflict with the new state.

How it works: When the queue engine processes an intent, it calls `validate_intent` again — even though the intent is already in VALIDATED status. This re-simulation against the current `M(t)` ensures the validation is fresh.

Invariant 3: Bounded Retry

```
retries > MAX_RETRIES → REJECTED
```

In plain language: If an intent fails validation more than 3 times (default `MAX_RETRIES = 3`), it is permanently rejected. No infinite retry loops.

Why 3? A single failure might be a flaky test. Two failures might be a transient infrastructure issue. Three failures mean the change has a real, persistent problem. Without this bound, a broken intent could cycle indefinitely, consuming pipeline resources and blocking the queue.

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

---

5. The Validation Pipeline

When an intent is validated, it passes through a multi-step pipeline. Each step can block (reject the intent) or pass (continue to the next step):

```
Step 1: Simulation
   Run merge simulation in isolated git worktree
   → Block if merge conflicts detected
        |
Step 2: Verification (Checks)
   Run required checks (lint, unit_tests, etc.) per risk level
   → Block if any required check fails
        |
Step 3: Risk Evaluation
   Build dependency graph → compute 4 risk signals → composite scores
   Optional: auto-reclassify risk level from computed scores
   (Never blocks — informational)
        |
Step 3.5: Coherence Evaluation
   Run coherence harness questions against working tree
   Cross-validate against risk metrics
   → Block if coherence verdict is "fail"
        |
Step 4: Policy Gates
   Evaluate all 5 gates against risk-level-appropriate thresholds:
     Gate 1: Verification (checks)
     Gate 2: Containment (score >= min)
     Gate 3: Entropy (delta <= budget)
     Gate 4: Security (findings within limits)
     Gate 5: Coherence (score >= threshold)
   → Block if any gate fails
        |
Step 5: Risk Gate
   Composite risk/damage/propagation scores within limits
   Gradual enforcement with deterministic rollout
   → Block if enforced and scores exceed limits
        |
Step 6: Finalize
   Status → VALIDATED
   Emit INTENT_VALIDATED event
   Return decision with full risk/policy/coherence data
```

If any step blocks, the pipeline short-circuits: a `INTENT_BLOCKED` event is emitted with the reason, and the remaining steps are not executed.

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

---

6. Queue Processing

The queue engine processes validated intents in priority order:

```
1. Acquire queue lock (file-based, prevents concurrent execution)
2. List all intents with status=VALIDATED, up to limit
3. For each intent (sorted by priority):
   a. Check dependencies:
      → Skip if any dependency is not MERGED
   b. Check retries:
      → REJECT if retries >= MAX_RETRIES (Invariant 3)
   c. Check pending reviews:
      → Skip if reviews are still pending/assigned
   d. Check rejected reviews:
      → Block if a review was rejected
   e. Revalidate (Invariant 2):
      → Run full validate_intent against current M(t)
      → If blocked: increment retries, set READY or REJECTED
   f. If validated:
      → Status → QUEUED
      → If auto_confirm: execute the actual merge
4. Emit QUEUE_PROCESSED event
5. Release queue lock
```

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

6.1 The Queue Lock

The queue uses a global file lock (`.converge/queue.lock`) with a 300-second TTL. Only one queue process can run at a time. This prevents race conditions where two processes might try to merge the same intent simultaneously.

If a previous queue run crashes without releasing the lock, the TTL ensures the lock expires and doesn't permanently block the system.

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

6.2 Dependency Resolution

Dependencies are checked before each intent is processed:

```python
for dep_id in intent.dependencies:
    dep = event_log.get_intent(dep_id)
    if dep is None or dep.status != Status.MERGED:
        unmet.append(dep_id)
```

If any dependency is not MERGED, the intent is skipped (not rejected) — it stays in VALIDATED status and will be retried in the next queue run. This allows dependencies to resolve naturally as other intents merge.

6.3 Retry Logic

When validation fails during queue processing:

```
new_retries = intent.retries + 1

If new_retries >= MAX_RETRIES:
    Status → REJECTED
    Emit INTENT_REJECTED event
Else:
    Status → READY (back in queue for next run)
    Emit INTENT_REQUEUED event
```

The READY status means the intent will be re-fetched and revalidated in the next queue run. Each failure increments the retry counter. After 3 failures, the intent is permanently rejected.

---

7. Merge Execution

When an intent passes all validation steps and auto_confirm is enabled:

```
1. Call scm.execute_merge_safe(source, target, cwd)
   → Performs the actual git merge
   → Returns the merge commit SHA
2. On success:
   Status → MERGED
   Emit INTENT_MERGED event with commit SHA
3. On failure:
   Increment retries
   Status → READY (or REJECTED if max retries)
   Emit INTENT_MERGE_FAILED event
```

The merge is performed using the SCM adapter, which wraps git operations. This is the only point in the entire pipeline where the target branch is actually modified.

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

7.1 Post-Merge Confirmation

For workflows where the merge is performed externally (e.g., via GitHub PR merge), the `confirm_merge` function transitions a QUEUED or VALIDATED intent to MERGED:

```
def confirm_merge(intent_id, merged_commit=None):
    # Verify intent exists and is in QUEUED or VALIDATED status
    # Update status to MERGED
    # Emit INTENT_MERGED event
```

---

8. Events and Audit Trail

Every state transition and significant action emits an Event to the append-only event log:

Event Type	When	Payload
`INTENT_CREATED`	Intent is created	Full intent data
`SIMULATION_COMPLETED`	Merge simulation finishes	mergeable, conflicts, files_changed
`CHECK_COMPLETED`	A check finishes running	check_type, passed, details
`RISK_EVALUATED`	Risk evaluation completes	All 4 signals, composite scores, bombs
`RISK_LEVEL_RECLASSIFIED`	Risk level changes from auto-classification	old level, new level, risk_score
`COHERENCE_EVALUATED`	Coherence harness completes	score, verdict, results
`POLICY_EVALUATED`	Policy gates evaluated	verdict, gates (pass/fail each)
`INTENT_VALIDATED`	All steps pass	source, target, trace_id
`INTENT_BLOCKED`	Any step blocks	reason, trace_id
`INTENT_REQUEUED`	Failed but retries remain	reason, retries
`INTENT_REJECTED`	Max retries or permanent block	reason, retries
`INTENT_MERGED`	Successfully merged	merged_commit, source, target
`INTENT_MERGE_FAILED`	Merge execution fails	error, retries
`INTENT_DEPENDENCY_BLOCKED`	Dependencies unmet	unmet list, plan_id
`QUEUE_PROCESSED`	Queue run completes	processed count

Every event carries a `trace_id` that links all events from a single validation run. This enables full traceability: given an intent ID and trace ID, you can reconstruct the exact sequence of steps, their inputs, and their outputs.

Source: `engine.py` (throughout), `event_types.py`

---

9. How Intents Differ from Pull Requests

Aspect	Pull Request	Converge Intent
Data model	Title, description, labels (unstructured)	Typed fields: risk_level, priority, dependencies, scope_hint (structured)
Risk assessment	Human judgment + optional CI	Automated: 4 risk signals, composite scoring, risk level classification
Policy enforcement	Manual review rules, CODEOWNERS	5 machine-evaluated gates with risk-adaptive thresholds
Dependencies	Informal ("merge X first")	Explicit: `dependencies: ["intent-id-1"]`, machine-enforced ordering
Retries	Manual re-run of CI	Automatic with bounded retries (MAX_RETRIES=3)
Ordering	First-come-first-served (usually)	Priority-based with dependency resolution
Merge simulation	Run at PR creation (may go stale)	Re-simulated before every merge attempt (Invariant 2)
Multi-tenant	Not supported	`tenant_id` isolates intents by tenant
Origin tracking	Author attribution	`origin_type` distinguishes human/agent/integration with policy overrides
Audit trail	CI logs, review comments	Structured events with trace_id linking all steps

---

10. Intent Origins and Policy Overrides

The `origin_type` field enables different treatment for different sources of changes:

Origin Type	Typical Source	Example Policy Override
`human`	Developer typing code	Standard profile
`agent`	AI coding agent	Stricter checks, lower risk threshold
`integration`	CI/CD pipeline, automated tooling	May skip certain checks, faster processing

Policy overrides are configured in `policy.json`:

```
{
  "origin_overrides": {
    "agent": {
      "medium": { "checks": ["lint", "unit_tests"] },
      "_default": { "containment_min": 0.6 }
    }
  }
}
```

When a policy profile is loaded for a given risk level, origin overrides are merged on top. This allows fine-grained control: “agent-generated medium-risk changes must pass unit_tests too” or “all agent changes need higher containment.”

Source: `policy.py`, `PolicyConfig.profile_for`

---

11. Design Rationale Summary

Design Choice	Rationale
Intent as a typed contract	Machine decisions require structured data. Unstructured PR descriptions can’t feed automated risk scoring or policy gates.
Explicit lifecycle states	Each state represents a distinct evidence level. The transition rules are the invariants of the system.
Revalidation before merge (Inv. 2)	Target branches move. A validation against stale state is not a validation. Re-simulation prevents “it was safe yesterday” failures.
Bounded retry (Inv. 3)	Prevents broken intents from cycling indefinitely. Three attempts is enough to distinguish flakiness from genuine failure.
Priority ordering	Not all changes are equal. A security fix should merge before a formatting change. Priority makes this explicit and machine-enforceable.
Explicit dependencies	Implicit ordering (merge in branch creation order) breaks when changes have logical relationships. Dependencies make ordering constraints visible and enforceable.
Origin-type tracking	Different sources of changes have different trust profiles. Agents may need stricter validation; integrations may need faster processing.
Plan grouping	Coordinated multi-step migrations need to be visible as a unit. Plan IDs enable dependency visualization and coordinated processing.
Event-sourced audit trail	Every decision is traceable. Regulatory compliance, post-incident analysis, and debugging all require knowing exactly what happened and why.

Design Choice	Rationale
Short hex IDs	12-char hex IDs are human-readable in logs and dashboards while remaining effectively unique across practical volumes.
Semantic + technical separation	Semantic fields are for humans (what/why). Technical fields are for machines (scope_hint, affected_modules). Separating them keeps both clean.
File-based queue lock	Simple, portable, no external dependencies. TTL prevents deadlock. Sufficient for the single-process queue model.

---

12. File Reference

File	Role
`models.py`	`Intent` dataclass, `Status` enum, `RiskLevel` enum
`engine.py`	Three invariants, validation pipeline, queue processing, merge execution
`policy.py`	Policy gate evaluation, profile loading, origin overrides
`defaults.py`	`MAX_RETRIES`, `DEFAULT_TARGET_BRANCH`, `QUEUE_LOCK_TTL_SECONDS`, `DEFAULT_PROFILES`
`event_log.py`	Intent CRUD, event append, queue lock, status transitions
`event_types.py`	`EventType` enum with all intent-related event types