The Specification Gap: Coordination Failure Under Partial Knowledge in Code Agents

Camilo Chacón Sartori[1*]

[1*]Catalan Institute of Nanoscience and Nanotechnology (ICN2), CSIC and BIST, Campus UAB, Bellaterra, 08193, Spain.

Corresponding author(s). E-mail(s): camilo.chacon@icn2.cat;

Abstract

LLM-based code agents are increasingly used to divide and independently implement parts of a software class—yet how specification completeness shapes their coordination under partial knowledge remains poorly understood from a software engineering perspective. We treat this as an integration problem: when two agents independently generate code for disjoint method groups, they must agree on shared internal data structures without runtime communication. We study this coordination problem across 51 class-generation tasks, varying specification completeness from full docstrings (L0) to bare signatures (L3) and introducing opposing structural biases to stress-test integration. Three findings emerge across three models from two families. First, a persistent *specification gap*: two-agent integration pass rate drops from 58% to 25% as specification detail is removed, leaving a 25–49pp coordination penalty that persists at every level. Second, an AST-based conflict detector flags structural mismatches at zero LLM cost, reaching 97% precision at L3—*diagnosing* where coordination fails but not improving merger-agent repair. Third, a $\mathbf{2} \times \mathbf{2}$ factorial recovery experiment shows that restoring the full specification to a merger agent recovers the single-agent ceiling (89%) in our experimental setting, while conflict reports are not statistically distinguishable from zero effect. A gap-decomposition experiment further isolates coordination cost (+16pp) from information asymmetry (+11pp), confirming the two effects are approximately additive. These results support a *specification-first* view: the specification functions as a coordination contract, and investing in richer specifications is more consequential than post-hoc conflict detection.

Keywords: Multi-agent code generation, Specification completeness, LLM coordination, Conflict detection, Software engineering

Corresponding author ORCID: 0000-0002-8543-9893. Code and data: https://github.com/camilochs/the_specification_gap.

1 Introduction

Large language models (LLMs) have demonstrated strong performance on single-agent code generation benchmarks (Chen et al. 2021; Li et al. 2023; Rozière et al. 2024). A natural next step is to move from one model to *multi-agent* systems, where several LLM-based agents collaborate to produce larger or more complex software artifacts (Hong et al. 2024; Qian et al. 2024). In this paper, we focus on *code agents*: LLM agents specialized for code generation that receive a specification and produce source code. The promise is straightforward: divide a class across agents, let each implement a subset of methods, and then merge the outputs. For empirical software engineering, this raises a concrete question about coordination under decomposition: when independently generated pieces of a software artifact must later be integrated, what information is necessary for them to remain compatible?

But division of labor introduces a coordination problem. When two code agents independently implement methods of the same class, they must agree on shared internal state—the data structures initialized in the class constructor (`__init__` in Python)—without communicating at runtime. If Agent A stores records as a list of tuples and Agent B expects a dictionary keyed by ID, their individually correct methods will fail when integrated. This is the problem of *partial knowledge*: each code agent sees only part of the specification and must infer the rest. Importantly, the errors that arise from this coordination failure are architecturally distinct from those produced by individual agents. Recent philosophical analysis has argued that AI code generation systems exhibit error profiles rooted in their stochastic architecture rather than in human-cognitive limitations (Sartori 2026); multi-agent decomposition introduces a further error dimension—structural incompatibility between independently generated outputs—that neither single-agent systems nor human teams typically face in the same form.

In traditional software engineering, this problem is solved by *specifications*. Parnas's information-hiding principle (Parnas 1972) and Meyer's Design by Contract (Meyer 1992) both argue that module interfaces must be precise enough to constrain implementations. But how precise is "precise enough" for code agents?

We treat multi-agent code generation as a software integration problem rather than as a model capability benchmark. This framing aligns with the tradition of integration testing (Engler et al. 2001), software merging (Mens 2002), and coordinated development under module boundaries (Parnas 1972): the question is not which model performs best in isolation, but what information is necessary for independently generated artifacts to remain compatible at integration time.

We answer this question empirically. We construct a controlled experiment with one independent variable—*specification completeness*—and measure its effect on multi-agent integration success. Our specification levels range from L0 (full docstrings with doctests and explicit data-structure references) to L3 (bare method signatures only). We assign two code agents *opposing* implementation biases: one prefers lists, the other

dictionaries. A single unbiased agent provides a ceiling at each level. The goal is not to compare specific proprietary models as a benchmark; it is to identify whether a coordination failure mode appears systematically once shared specifications become incomplete.

Figure 1 summarizes the central phenomenon studied in the paper: specification loss induces a large integration gap, and restoring the full specification is sufficient to recover from it in our experimental setting.

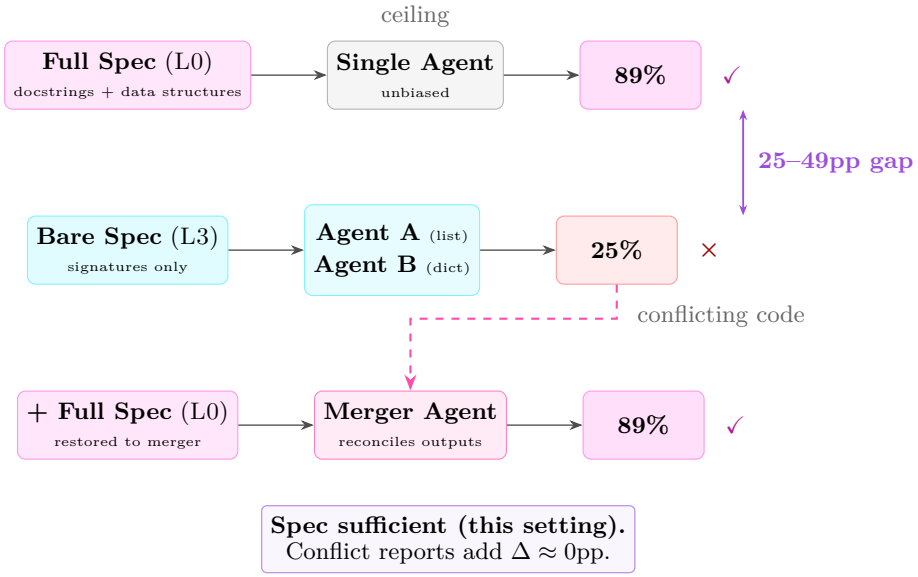


Fig. 1: The specification gap and its resolution. **Top:** A single agent with the full specification achieves 89% pass rate. **Middle:** Two biased agents with bare specifications produce conflicting code (25%). **Bottom:** Restoring the full specification to a merger agent recovers 89%—matching the single-agent ceiling—while conflict reports add nothing. The specification is both the cause of failure and a sufficient instrument of recovery in our setup.

Our key findings are:

1. **Specifications are coordination mechanisms.** Multi-agent integration success degrades monotonically from 58.2% (L0) to 24.6% (L3) as specification detail is removed. Explicit data-structure references in docstrings override agent biases.
2. **The coordination tax is persistent.** The gap between single-agent and multi-agent performance remains significant at every specification level (25–49pp across three models from two families), never narrowing to zero. Better specifications reduce the coordination cost but do not eliminate it in our setting—a substantial penalty persists even with full docstrings.
3. **Conflicts are diagnosable at zero cost.** An AST-based detector identifies structural conflicts between agent outputs before integration, with precision improving

from 43.5% at L0 to 96.7% at L3; however, our recovery experiment (RQ4) shows the detector is *diagnostic*—useful for explaining where coordination fails—but does not improve repair when a merger agent has the full specification.

4. **Specification is sufficient for recovery in our experimental setting.** A 2×2 factorial experiment ($n = 53$ tasks) reveals that providing the full specification to a merger agent restores 88.9% pass rate—matching the single-agent ceiling (88.3%)—while conflict reports alone produce a null effect (95% CI: $[-4.7, +4.7]$pp) and measurably decrease performance (-6.6pp, 95% CI: $[-13.6, -0.4]$pp) when the full specification is already available.

Together, these findings connect classical software engineering theory to modern AI practice. They suggest that code agents need specification-aware orchestration—not just task decomposition or post-hoc conflict detection—to coordinate effectively under partial knowledge. More broadly, they frame multi-agent code generation as an empirical software engineering problem about interfaces, integration, and coordination contracts rather than only as a model capability benchmark.

Beyond these empirical findings, we contribute a reusable experimental methodology for studying multi-agent coordination. Our design combines five elements: (i) a controlled benchmark with nested specification levels that isolate a single variable; (ii) a specification-ablation protocol that degrades information monotonically; (iii) a zero-cost AST conflict detector for pre-integration diagnostics; (iv) a factorial recovery experiment that separates specification quality from conflict information; and (v) a 2×2 init-visibility decomposition that disentangles coordination cost from information asymmetry. Each element can be adapted to other multi-agent settings, languages, or task types.

The paper unfolds as follows. Section 2 reviews prior work on multi-agent code generation, specification theory, and conflict detection. Section 3 details the experimental design, including the four specification levels, the agent configuration, and the 2×2 factorial recovery experiment. Section 4 reports the results for all four research questions. Section 5 examines the implications of the findings, cross-model replication, and threats to validity. Finally, Section 6 concludes the paper with practical recommendations for code agent systems.

2 Background and Related Work

2.1 Multi-Agent Code Generation

Recent systems divide code generation across multiple LLM agents. MetaGPT (Hong et al. 2024) assigns software engineering roles (architect, engineer, tester) to agents communicating through structured artifacts. ChatDev (Qian et al. 2024) simulates a software company with chat-based agent interaction. CodeAgent (Tang et al. 2024) integrates external tools for repository-level tasks. More recent work extends this line with adaptive planning for function-level generation (Zhu et al. 2025), simulation-driven planning and debugging loops (Islam et al. 2025), and decentralized collaboration over shared Git artifacts (Huang et al. 2025).

These systems demonstrate the potential of multi-agent code generation, but they do not isolate how specification quality shapes coordination. Our work complements them with a controlled experiment that manipulates specification completeness as a single independent variable.

2.2 Specification and Coordination in Software Engineering

Parnas (Parnas 1972) established that modular decomposition requires precise interface specifications to enable independent development. Meyer's Design by Contract (Meyer 1992) formalized this idea through preconditions, postconditions, and invariants. Together, these classical principles frame specifications as coordination contracts between independently developed modules.

Conway's Law (Conway 1968) predicts that system structure mirrors organizational communication structure. In multi-agent LLM systems, the "organization" is the set of agents, and the "communication" is the shared specification. Our experiment tests whether specification quality predicts the degree to which independent agents produce compatible code.

2.3 LLM Code Generation Benchmarks

HumanEval (Chen et al. 2021) and MBPP (Austin et al. 2021) evaluate single-function generation. ClassEval (Du et al. 2024) extends this to class-level generation with inter-method dependencies, making it suitable for studying multi-agent coordination. We build our benchmark (AmbigClass) on ClassEval tasks, generating specification variants at four completeness levels.

2.4 Conflict Detection in Software

Static analysis for type conflicts (Engler et al. 2001), merge-conflict detection (Mens 2002), and program analysis for integration errors (Binkley et al. 1995) are well-studied in traditional software. We adapt these ideas to LLM-generated code, using AST-based analysis to detect type mismatches and state-access conflicts between agent outputs before integration.

3 Experimental Design

Our experiment isolates the effect of specification completeness on multi-agent code integration. We assign two biased LLM agents disjoint subsets of methods from the same class, vary the amount of structural information each agent receives, and measure whether their independently generated code can be merged into a passing implementation. A single-agent baseline, which always sees the full class body, provides the performance ceiling. This design lets us separate specification-driven coordination failures from intrinsic task difficulty, and test whether post-hoc conflict detection or richer specifications better enable recovery.

3.1 Research Questions

Four research questions progressively move from characterizing the specification gap to understanding its causes and potential remedies:

RQ1 Does specification completeness predict multi-agent integration success?
RQ2 How does the multi-agent coordination penalty vary across specification levels?
RQ3 Can AST-based conflict detection provide an observable signal of specification inadequacy?
RQ4 What enables recovery from multi-agent integration failures: specification quality, conflict information, or their combination?

3.2 Independent Variable: Specification Level

We define four nested specification levels, each providing strictly less information than the previous (Table 1):

Table 1: Information content at each specification level.

Information	L0	L1	L2	L3
Method signatures	✓	✓	✓	✓
Full docstrings	✓	✓	simplified	—
Doctest examples	✓	—	—	—
Edge-case behavior	✓	✓	—	—
Data-structure references	✓	✓	—	—

The critical transition is between L1 and L2: L0/L1 contain explicit references to data structures (*e.g.*, "add to the `job_listings` list"), while L2/L3 use abstract language ("Publish positions").

3.3 Conditions

For each task t and specification level ℓ, we evaluate three conditions:

Single(ℓ) One unbiased agent receives the skeleton at level ℓ *with* the `__init__` body intact. This provides the ceiling: the agent sees the data structures in code.
Split(ℓ) Two biased agents each receive the skeleton at level ℓ *without* the `__init__` body (replaced with `pass`). Agent A is instructed to prefer lists; Agent B prefers dictionaries. Each implements a disjoint subset of methods. Their outputs are merged via textual concatenation of method bodies.
Conflicts(ℓ) The AST-based conflict detector analyzes the two agent outputs before integration, reporting type, state, and protocol conflicts.

The key design choice: *stripping `__init__` for split agents only.* This forces split agents to infer data structures from the specification (or default to their bias), while single agents always know the ground truth.

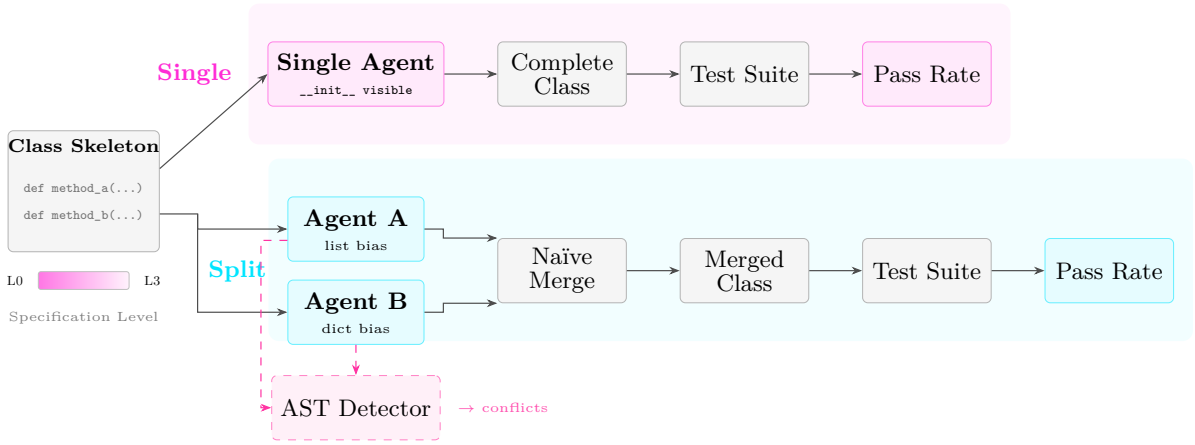


Fig. 2: Experimental overview. Each task's class skeleton is processed at specification level $\ell \in \{\text{L0–L3}\}$. The *single* condition (top) gives one unbiased agent the full skeleton including `__init__`. The *split* condition (bottom) hides `__init__` and assigns methods to two biased agents whose outputs are merged. An AST-based detector (dashed) checks for structural conflicts before integration.

Note on asymmetry.

The single condition sees the full skeleton including the `__init__` body, while the split condition does not (it is replaced with `pass`). This asymmetry is *intentional in the main experiment*: it lets coordination cost (two agents disagreeing on data structures) and information asymmetry (split agents inferring data structures from scratch) appear together, as they would in realistic multi-agent settings where agents do not share all internal state. We later isolate the two factors via a 2×2 init-visibility experiment (Section 5.3), which shows that the coordination penalty persists (+15.7pp) even when both biased agents see and are told to preserve the identical `__init__` body.

Figure 2 summarizes the full experimental pipeline, from class skeleton to agent generation, conflict analysis, and downstream evaluation.

3.4 Tasks and Benchmark

We use 51 tasks from ClassEval (Du et al. 2024), each with $\geq$3 methods (from 88 eligible tasks in the full ClassEval set of 100). For each task, we generate four specification variants (L0–L3) using progressive information removal. The resulting benchmark, **AmbigClass**, contains 204 task-specification pairs. The experiment requires 612 LLM calls (3 per task-level: 1 single + 2 split agents) and 408 test suite executions.

3.5 Agent Configuration

Each configuration choice is motivated by the need to create a controlled setting in which specification-driven coordination failures arise reliably while remaining representative of real multi-agent workflows.

- **Model**: Claude Sonnet 4.[1] We use the same model for all agents to ensure that performance differences reflect specification and coordination effects rather than capability differences between models.
- **Temperature**: 0.7 for biased agents, 0.0 for single agent. The elevated temperature for split agents increases output diversity, making it more likely that latent specification ambiguities surface as observable conflicts. The single agent uses greedy decoding to provide a stable, reproducible ceiling.
- **Biases**: Agent A's system prompt includes "always store data as LISTS"; Agent B includes "always store data as DICTS." This deliberate bias serves as a controlled proxy for the implicit stylistic divergences that emerge when different developers—or agents—make independent design choices without a shared data-structure contract. The list-vs-dictionary opposition was chosen because it produces structurally incompatible code that is easy to detect via AST analysis yet representative of real-world type conflicts.
- **Task split**: Methods are divided into two disjoint groups by alternating index. This simple, deterministic scheme guarantees that both agents contribute to every task and that inter-method dependencies are spread across agents rather than concentrated within one, maximizing the coordination surface.

3.6 Evaluation

Each condition's output is evaluated against the original ClassEval test suite. We use three metrics with explicit units:

- **Test-level pass rate** (continuous, 0–100%): proportion of tests passed within a single task. Per-task test counts vary across the 51 ClassEval tasks (median ≈6, range 3–31).
- **Task-level success** (binary): a task is "successful" if its test-level pass rate ≥80%. We use this only for descriptive task counts; statistical inference uses the continuous test-level pass rate.
- **Statistical unit**: each (task, condition) pair contributes one observation. Wilcoxon signed-rank tests pair Single and Split observations within the same task. Friedman tests treat condition as a within-subject factor and task as the unit. We report **macro-averaged pass rate** (mean of per-task percentages) so that tasks with more tests do not dominate the aggregate.

3.7 Recovery Experiment (RQ4)

To isolate the contributions of specification quality and conflict information to integration recovery, we conduct a 2×2 factorial experiment on 53 tasks. Two biased agents always generate code under L3 specifications (bare signatures). Their outputs are then given to a *merger agent* that attempts to combine them into a correct class. The factorial manipulates what the merger receives:

Specification level Either L3 (the same minimal spec the agents used) or L0 (the full specification with docstrings and data-structure references).

[1] API model key: `claude-sonnet-4-20250514`

Conflict report Either absent or present (the AST detector's report of type, state, and protocol conflicts between the two agents' outputs).

This yields four merge conditions—*Blind* (L3, no report), *Guided* (L3, with report), *Spec-Only* (L0, no report), and *Resolve* (L0, with report)—plus two baselines: *Single* (one unbiased agent with L0 and `__init__` visible) and *Naïve* (direct concatenation without any merger). Total cost: $3.54.

In total, the experimental design produces 204 task-specification pairs for the main experiment and 318 merger runs for the recovery experiment, enabling a systematic comparison across specification levels, agent conditions, and recovery strategies. We now turn to the results.

4 Results

We present the results organized by research question. First, we show that specification completeness is the dominant predictor of multi-agent integration success (RQ1). We then quantify the coordination penalty that persists even under full specifications (RQ2), demonstrate that AST-based conflict detection reliably signals specification inadequacy (RQ3), and finally evaluate which factors—specification quality, conflict information, or their combination—enable recovery from integration failures (RQ4).

4.1 RQ1: Specification Completeness Predicts Integration Success

RQ1 asks whether the amount of structural information in the specification predicts whether independently generated agent outputs can be successfully integrated. We compare Single and Split conditions across all four specification levels, measuring test pass rates and the number of AST-detected conflicts.

Table 2: Test pass rates by specification level and condition ($n = 51$ tasks, Sonnet). Conflicts = mean AST-detected conflicts per task. Gap 95% CI from paired bootstrap ($B = 10{,}000$).

Level	Single (%)	Split (%)	Gap (pp)	Gap 95% CI (pp)	Conflicts (mean)
L0	88.6	58.2	+30.4	[+18.6, +42.3]	1.0
L1	77.8	43.0	+34.8	[+24.5, +44.9]	1.4
L2	66.0	36.5	+29.5	[+22.3, +37.1]	1.5
L3	55.8	24.6	+31.3	[+23.0, +40.0]	1.6
Mean	72.1	40.6	+31.5	—	1.4

Both conditions degrade monotonically as specification detail is removed (Table 2 and fig. 3), confirming that specification completeness is a strong predictor of integration success. At L0, where docstrings explicitly name data structures and include usage examples, the single agent achieves 88.6% and the split condition reaches 58.2%. By L3, where only bare method signatures remain, these figures drop to 55.8% and

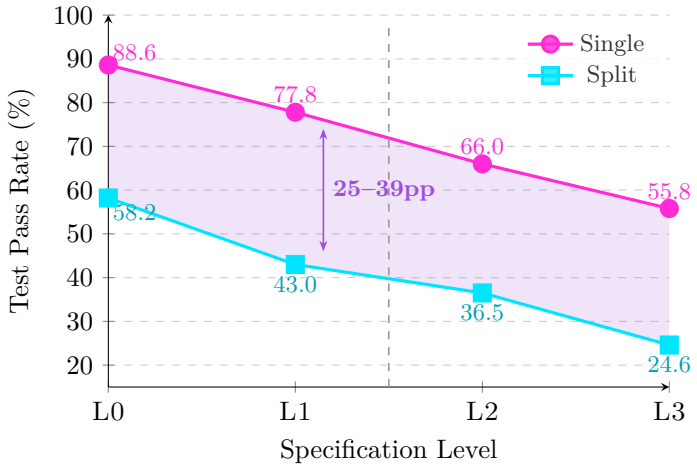


Fig. 3: Test pass rates degrade monotonically as specification detail is removed. The shaded band highlights the persistent coordination gap (25–39pp across Claude variants; broader 25–49pp range with GPT-5.4-mini, see Section 5.4) between single-agent and split-agent conditions. The dashed line marks the transition where explicit data-structure references are stripped (L1→L2).

24.6%, respectively. The overall magnitude of degradation is similar for both conditions (∼33pp), yet the split condition starts from a substantially lower baseline because coordination errors compound information loss: even at L0, agents that cannot see the shared `__init__` body must independently infer compatible data structures from the specification alone. The mean number of AST-detected conflicts also rises with specification degradation (1.0 at L0 to 1.6 at L3), providing an independent structural signal that corroborates the test-based results.

4.2 RQ2: The Coordination Tax

RQ2 investigates whether the performance gap between single-agent and multi-agent conditions can be closed by improving specification quality, or whether multi-agent coordination imposes an irreducible cost. If specifications were the only bottleneck, we would expect the gap to vanish at L0.

The gap between Single and Split is significant at every level, ranging from 29.5 to 34.8pp for Sonnet and 25.5 to 38.8pp for Haiku (Table 6). This "coordination tax" is *not* eliminated by better specifications—even at L0, where docstrings explicitly name data structures, the two-agent condition loses 25–30 percentage points.

Wilcoxon signed-rank tests confirm the gap is significant at every level (all $p < 0.001$; Cohen's $d = 0.71$–1.08). A Friedman test finds no significant variation in gap magnitude across levels ($\chi^2 = 6.93$, $p = 0.074$), indicating that the coordination tax neither increases nor decreases with specification degradation. The 25–39pp gap range is not driven by sampling noise: 95% bootstrap CIs at every level exclude zero by a wide margin (e.g., Sonnet L0: [+18.6, +42.3]pp; L3: [+23.0, +40.0]pp; see Table 2 and

Table 6 for full CI columns). This finding has a direct practical implication: even when teams invest in comprehensive specifications, multi-agent architectures that divide code generation across agents will still incur a measurable performance penalty relative to a single agent that sees the full context. The penalty stems not from ambiguous specifications but from the inherent difficulty of producing structurally compatible code without shared state.

4.3 RQ3: Conflict Detection as a Specification Signal

RQ3 examines whether AST-based conflict detection can serve as an automated diagnostic for specification inadequacy. If structural conflicts between agent outputs correlate with specification degradation and predict integration failure, the detector would provide an actionable signal—without requiring any additional LLM calls—that current specifications are insufficient for reliable multi-agent coordination.

Conflicts increase monotonically with specification degradation (mean 1.0 to 1.6 per task; see Figure 9 for the full taxonomy). Type conflicts dominate at every level (60.6% of all 284 detections), growing fastest as data-structure references are removed.

The detector operates at zero inference cost: it analyzes only the generated ASTs and requires no additional LLM calls. Table 3 and fig. 4 show its performance. Overall recall is 62.9% and precision is 75.0%, but these averages hide the main pattern. Precision rises from 43.5% at L0 to **96.7%** at L3, where weak specifications produce structural conflicts rather than subtler semantic incompatibilities.

Table 3: AST conflict detector performance by specification level. A task is "failed" if Split pass rate <50%; "detected" if ≥1 conflict found.

Level	n	TP	FN	FP	Recall	Precision
L0	51	10	8	13	55.6%	43.5%
L1	51	16	12	7	57.1%	69.6%
L2	51	23	12	5	65.7%	82.1%
L3	51	29	14	1	67.4%	**96.7%**
All	204	78	46	26	62.9%	75.0%

A breakdown by conflict type (Figure 9 in Appendix A) confirms that type conflicts dominate at all levels, growing from 26 at L0 to 53 at L3; state conflicts are second, while protocol and return conflicts remain rare.

At L0, false positives arise because structural differences do not always cause test failures—agents may use different internal representations that happen to satisfy the same interface contract. False negatives reflect the complementary limitation: semantically incorrect but structurally compatible code that AST analysis cannot catch, such as two agents that both use dictionaries but disagree on key names.

Together, these results establish the AST detector as a useful—though imperfect—specification adequacy signal. Its high precision at weak specification levels means that

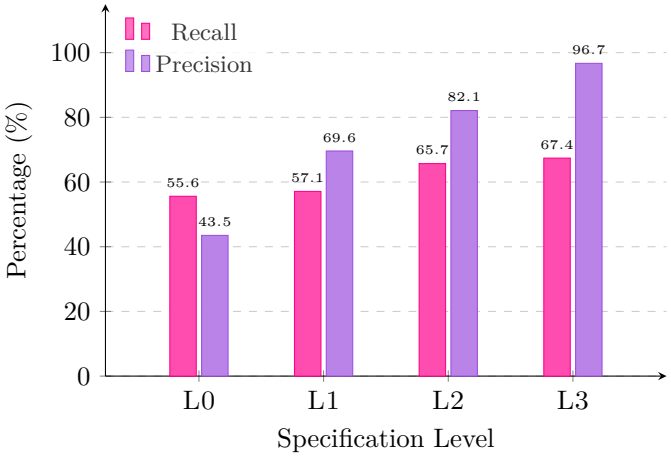


Fig. 4: AST conflict detector recall and precision by specification level. Precision improves as specifications degrade, reaching 96.7% at L3—the detector is most precise at the weakest specification level, where structural conflicts are most frequent.

when it flags a conflict, the specification almost certainly lacks the structural detail needed for reliable coordination. The detector can *diagnose* integration failures—but can conflict information help a merger agent *fix* them?

4.4 RQ4: Recovery Requires Specification, Not Conflict Reports

RQ4 asks what enables a merger agent to recover from integration failures produced by biased split agents: richer specifications, conflict reports, or their combination. This question has direct practical significance—it determines whether teams should invest in building better conflict-detection infrastructure or in improving the specifications that agents receive. The 2×2 factorial design (Section 3) independently varies specification level (L3 vs. L0) and conflict report availability (absent vs. present), yielding four merge conditions tested on 53 tasks.

Table 4 and figs. 5, 5a and 5b make the result unambiguous. The specification effect dominates (+36.2pp, 95% CI: $[+27.6, +45.0]$pp). Conflict information produces no detectable effect at L3 ($\Delta = 0.0$pp, 95% CI: $[-4.7, +4.7]$pp) and measurably decreases performance at L0 (-6.6pp, 95% CI: $[-13.6, -0.4]$pp; the CI excludes zero).

The critical finding is the Spec-Only condition. At 88.9%, it matches—and even marginally exceeds—the single-agent ceiling (88.3%), showing that in our experimental setting, the full specification alone was sufficient to recover from integration failure. This means that a merger agent equipped with a complete specification can reconstruct a correct implementation from two structurally incompatible agent outputs, effectively undoing the coordination damage caused by partial knowledge.

Equally striking is the comparison between Blind and Guided (both 52.7%): adding conflict reports to a minimal specification produces no statistically distinguishable

Table 4: Recovery experiment: 2×2 factorial ($n = 53$ tasks). 120 AST-detected conflicts (79 type, 37 state, 3 protocol, 1 return).

Condition	Spec	Conflicts	Pass rate (%)
Single (ceiling)	L0	—	88.3
Naïve (floor)	—	—	0.0
Blind	L3	No	52.7
Guided	L3	Yes	52.7
Spec-Only	L0	No	88.9
Resolve	L0	Yes	82.3
Spec effect: +36.2pp [95% CI: +27.6, +45.0]			Conflict effect: ≈ 0pp [CI: −4.7, +4.7] / −6.6pp [CI: −13.6, −0.4]

improvement (95% bootstrap CI: $[-4.7, +4.7]$pp). Without the structural context that the full specification provides, the merger agent cannot act on the detector's diagnoses—knowing *that* two agents disagree on a data structure is useless without knowing *what* the correct structure should be.

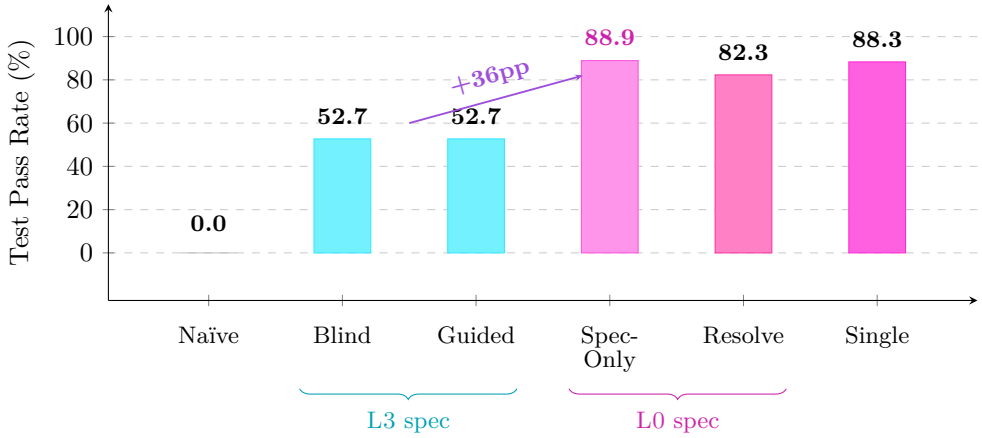
The negative interaction (−6.6pp in the Resolve condition) suggests that conflict reports introduce noise when the specification already resolves the structural mismatch. One plausible mechanism is that the detailed conflict report biases the merger toward local, symptom-level fixes rather than the holistic redesign that the full specification enables, paralleling findings where redundant information degrades human decision-making (Engler et al. 2001).

5 Discussion

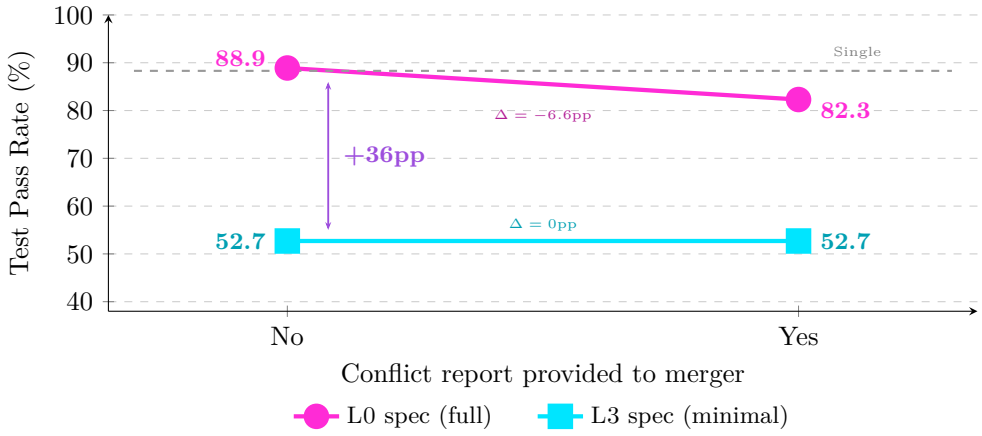
The results in Section 4 establish three core findings: specification completeness predicts integration success, a persistent coordination tax survives even under full specifications, and recovery from integration failure is driven almost entirely by specification quality rather than conflict diagnostics. In this section, we interpret these findings through the lens of classical software engineering principles, examine the sources of task-level variability, disentangle coordination cost from information asymmetry through an additional factorial experiment, and address the limitations of our experimental design. Together, these analyses clarify both the practical implications of the specification gap and the boundaries of our claims.

5.1 Specifications as Coordination and Recovery

Our results lend empirical support to an intuition from classical software engineering: specifications are not merely documentation; they also function as *coordination mechanisms*. When two code agents share a specification that says "add to the `job_listings` list," both produce list-based implementations regardless of their individual biases. When the specification says only "Publish positions," each agent falls back to its prior and integration fails. In this sense, our findings align with Meyer's Design by Contract (Meyer 1992): for code agents operating under partial knowledge,



(a) All six conditions, ordered by merger input.



(b) 2×2 factorial interaction.

Fig. 5: Recovery experiment results ($n = 53$ tasks). (a) Six conditions showing Spec-Only (88.9%) matches Single (88.3%), while Blind and Guided are identical (52.7%)—conflict information has no detectable effect without the full specification. (b) Interaction plot: the L0 specification accounts for +36pp recovery (95% CI: $[+27.6, +45.0]$pp); conflict reports produce no detectable effect at L3 (95% CI: $[-4.7, +4.7]$pp) and −6.6pp at L0. Dashed line: single-agent baseline.

"precise enough" appears to require data-structure information, not only behavioral descriptions.

The factorial recovery experiment (Table 4) qualifies this interpretation. In our setting, the AST conflict detector is valuable as a *diagnostic* tool—it helps explain *why* integration fails—but it does not improve recovery once a merger agent is involved. By contrast, the full specification was sufficient to restore performance to the single-agent

ceiling in this controlled setting. One plausible interpretation is that the specification already resolves the structural ambiguities that the conflict reports expose, leaving little additional value for the reports themselves.

The practical implication is correspondingly cautious: when code agents fail to integrate under conditions similar to ours—shared-class generation with opposing structural biases—improving the specification is likely to be more effective than adding conflict-detection infrastructure alone.

A natural concern is whether the prompt-injected biases in our design overstate the coordination problem. We argue that the injection models a realistic operational risk: in practice, agents are configured with different system prompts, reuse prompts inherited from prior projects, or are trained on codebases with divergent stylistic norms. The list-vs-dictionary opposition was chosen because it produces structurally detectable conflicts, but the underlying mechanism—independent agents defaulting to incompatible priors under ambiguous specifications—does not require this level of explicitness to operate. The init-visibility experiment (Section 5.3) supports this interpretation: even when both agents share the identical constructor body and are instructed to preserve it, the coordination tax persists at +15.7pp, indicating that the gap is not an artifact of the bias injection but a property of independent generation itself.

5.2 The Coordination Tax and Task-Level Variability

Even with full specifications (L0), multi-agent integration loses 25–30pp relative to a single agent. This gap persists at every specification level (25–39pp across Claude variants; up to 49pp with GPT-5.4-mini, see Section 5.4), suggesting a coordination cost that specification quality alone does not eliminate in our setting. Part of the gap does stem from a design asymmetry: single agents see the `__init__` body, while split agents must infer data structures. The init-visibility experiment (Section 5.3) separates these effects and indicates that coordination is the *larger* component (+16–20pp), while information asymmetry is secondary (+11–15pp); the two are approximately additive.

The aggregate gap conceals substantial task-level variation (std 27–43pp). Method count does not predict the coordination gap ($r \approx 0$, $p > 0.5$ for both models). Instead, task difficulty depends on whether agents' biases happen to be compatible. Cross-model category agreement is 72% (36/50 common tasks), with "easy" tasks sharing a common trait: simple, unambiguous state where list-vs-dictionary bias is irrelevant.

In a minority of cases, split agents actually outperform the single agent (13.7% of tasks at L0, 5.9% at L3). This occurs when the biased agents' forced data structure accidentally matches the expected interface better than the single agent's unconstrained choice.

5.3 Disentangling Coordination from Information Asymmetry

The preceding gap conflates two factors: *coordination cost* (two biased agents must agree on data structures) and *information asymmetry* (split agents do not see the `__init__` body that single agents receive). To separate these, we run a 2×2 factorial

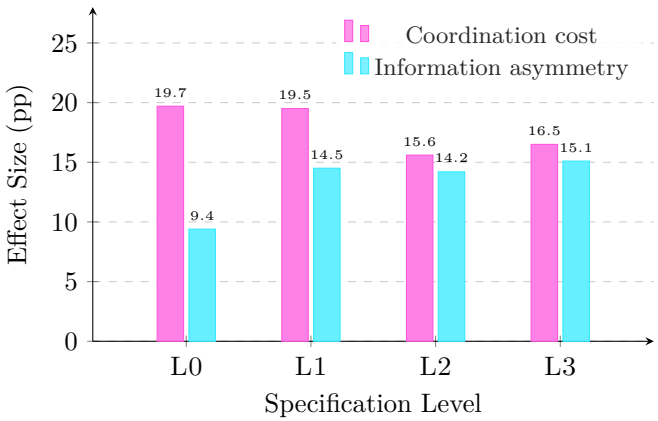


Fig. 6: Gap decomposition by specification level. The coordination effect (single vs. split, averaging over visibility) consistently exceeds the information effect (visible vs. hidden, averaging over agent mode) at every level. At L0, coordination accounts for twice the gap; by L3, the two components converge.

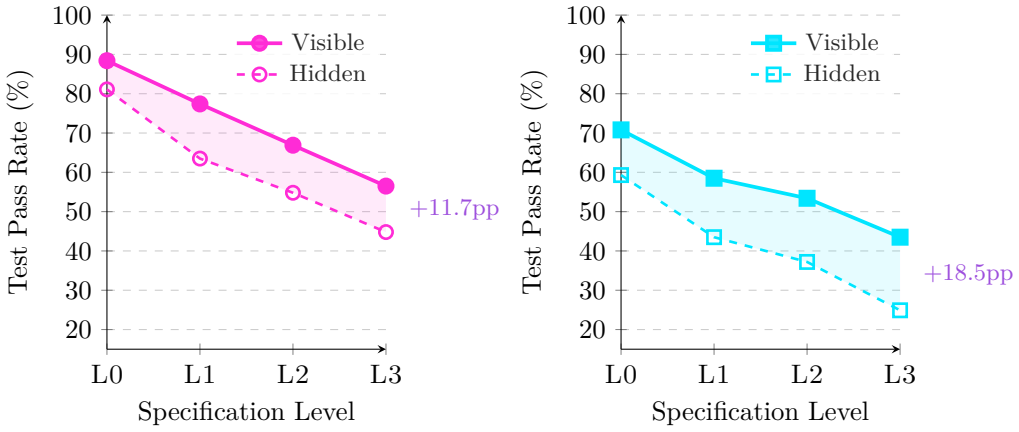
experiment ($n = 50$ tasks) crossing agent mode (single vs. split) with `__init__` visibility (visible vs. hidden), creating four conditions (see Appendix C for full prompts). Table 5 reports the results.

Table 5: Init visibility experiment: 2×2 factorial ($n = 50$ tasks). Top: mean pass rates (%) across all spec levels. Bottom: per-level detail.

	Init Visible (%)	Init Hidden (%)	Info Effect (pp)
Single	72.3	61.1	+11.2
Split	56.6	41.2	+15.3
Coord Effect (pp)	+15.7	+19.8	Interaction: −4.1

	Single (%)		Split (%)		Info Effect		Coord Effect	
Level	Vis	Hid	Vis	Hid	Single	Split	Vis	Hid
L0	88.4	81.1	70.8	59.3	+7.3	+11.5	+17.6	+21.8
L1	77.4	63.5	58.5	43.5	+13.9	+15.0	+18.9	+20.0
L2	66.9	54.8	53.4	37.2	+12.1	+16.3	+13.5	+17.6
L3	56.5	44.8	43.5	24.9	+11.7	+18.5	+13.0	+19.9

Coordination is the larger factor. Even when both agents see the identical `__init__` body and are told to preserve it, the list/dict bias still causes a +15.7pp penalty. This suggests that the specification gap is not merely an information confound.



(a) Single agent: constructor visible vs. hidden.

(b) Split agents: constructor visible vs. hidden.

Fig. 7: Init visibility experiment. Solid lines: `__init__` visible; dashed lines: hidden. Shaded area: information effect (gap due to `__init__` visibility). The information gap widens as specifications degrade, especially for split agents (right), where it grows from +11.5pp at L0 to +18.5pp at L3. The vertical distance between the two panels' solid lines represents the pure coordination effect.

Across all settings, the coordination effect (+15.7 to +19.8pp) exceeds the information effect (+11.2 to +15.3pp).

Information asymmetry is real but secondary. Hiding `__init__` costs the single agent 11.2pp on average; the effect is larger for split agents (+15.3pp), where information loss compounds with coordination difficulty.

The effects are approximately additive (interaction = −4.1pp), meaning the gap decomposes reasonably cleanly into two mechanisms (Figure 6). Figure 7 shows the per-level degradation curves, while Figures 7a and 7b separate the same pattern for single-agent and split-agent settings. Notably, AST conflicts drop from 1.3 per task when `__init__` is hidden to 0.6 when it is visible, and they remain constant at 0.6 across all spec levels in the visible condition. This pattern suggests that the constructor body helps anchor agents to a shared data structure.

5.4 Cross-Model Replication

To test whether the specification gap is model-specific or model-family-specific, we replicate the experiment with two additional models: Claude Haiku 4.5[2] (same family as Sonnet, smaller and less capable, $n = 50$ tasks) and GPT-5.4-mini[3] from OpenAI (different architecture and training lineage, $n = 25$ tasks; see details below). The Haiku replication is intended as an intra-family robustness check; the GPT-5.4-mini

[2]API model key: `claude-haiku-4-5-20251001`
[3]API model key: `gpt-5.4-mini-2026-03-17` (OpenAI).

replication is an inter-family check to rule out provider-specific artifacts. To assess run-to-run stability within the Claude family, we execute the Haiku experiment three times (total cost: $4.38 vs. $4.83 for Sonnet). Table 6 reports Haiku values as mean ± standard deviation across the three runs, and Figure 8 overlays degradation curves for all three models.

Table 6: Cross-model replication. Sonnet ($n = 51$, 1 run) vs. Haiku ($n = 50$, 3 runs, mean ± std) vs. GPT-5.4-mini† ($n = 25$, 1 run). All gaps significant (Wilcoxon, $p < 0.001$ per level).

	Single (%)			Split (%)			Gap (pp)		
Level	Sonnet	Haiku	GPT	Sonnet	Haiku	GPT	Sonnet	Haiku	GPT
L0	88.6	$86.6_{\pm 0.1}$	84.5	58.2	$61.1_{\pm 1.9}$	43.9	+30.4	$+25.5_{\pm 1.8}$	+40.6
L1	77.8	$75.5_{\pm 0.3}$	75.9	43.0	$36.7_{\pm 0.7}$	27.2	+34.8	$+38.8_{\pm 0.6}$	+48.6
L2	66.0	$66.8_{\pm 0.6}$	67.5	36.5	$31.8_{\pm 0.3}$	27.1	+29.5	$+35.0_{\pm 0.6}$	+40.4
L3	55.8	$55.2_{\pm 0.7}$	61.3	24.6	$25.4_{\pm 0.8}$	23.6	+31.3	$+29.8_{\pm 0.3}$	+37.8

† Pinned: `gpt-5.4-mini-2026-03-17` (OpenAI). $n = 25$ = first 25 tasks of the 50-task Sonnet∩Haiku intersection. Cost: $0.07.

95% bootstrap CIs (Gap, $B = 10{,}000$): Sonnet: L0 $[+18.6, +42.3]$, L1 $[+24.5, +44.9]$, L2 $[+22.3, +37.1]$, L3 $[+23.0, +40.0]$ pp. Haiku: L0 $[+16.1, +35.5]$, L1 $[+30.2, +47.7]$, L2 $[+26.9, +43.5]$, L3 $[+21.3, +38.8]$ pp. GPT-5.4-mini: L0 $[+26.8, +54.1]$, L1 $[+35.7, +60.8]$, L2 $[+28.6, +53.0]$, L3 $[+24.9, +50.9]$ pp. All CIs exclude zero.

Cross-family replication with GPT-5.4-mini.

To address the limitation of testing within a single model family, we extend the replication to GPT-5.4-mini[4], an OpenAI model from a different architecture and training lineage. This is a cross-family check, not a capability benchmark: the goal is to confirm whether the coordination penalty mechanism transfers across providers, not to rank models.

Three patterns emerge. *Single-agent performance is consistent across all three models*: Sonnet, Haiku, and GPT-5.4-mini all trace similar degradation curves from ∼85–89% at L0 to ∼55–61% at L3 (within 6pp at every level across the three models), and Haiku's run-to-run standard deviation stays ≤0.7pp.

Split-agent degradation follows the same shape across all three models, though GPT-5.4-mini's split-agent performance is lower at most levels (L0: 43.9% vs. 58.2% Sonnet, 61.1% Haiku). This does not indicate a fundamental difference in coordination failure mode; rather, GPT-5.4-mini appears to respond more strongly to the injected

[4] Pinned model: `gpt-5.4-mini-2026-03-17` (OpenAI). Run on the first 25 tasks of the 50-task Sonnet∩Haiku intersection; total cost $0.07. API call parameters: `temperature` 0.7 for biased agents, 0.0 for single agent (same scheme as Claude); `max_completion_tokens` 4096; no `top_p` constraint applied. OpenAI's `system_fingerprint` field was not captured in the run logs; version reproducibility is ensured by the pinned model ID.

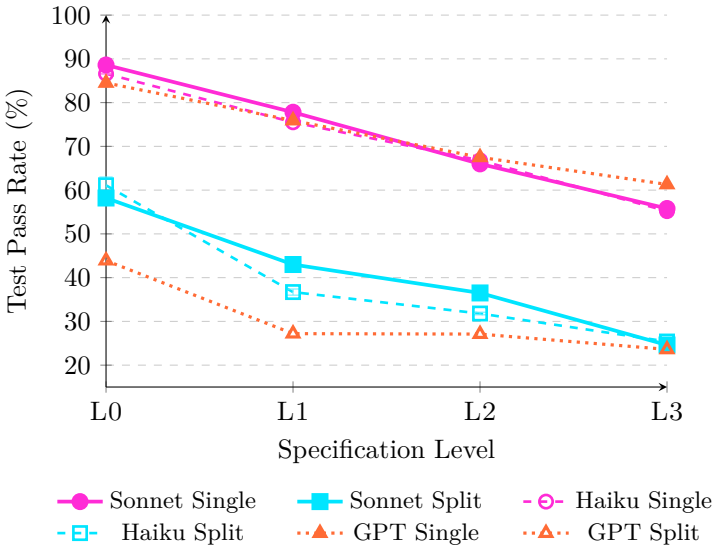


Fig. 8: Cross-model replication. Solid lines: Sonnet (1 run, $n = 51$); dashed lines: Haiku (3-run mean ± 1 std, $n = 50$); dotted orange lines: GPT-5.4-mini (1 run, $n = 25$). Single-agent curves (pink/orange upper) degrade similarly across all three models. Split-agent curves (cyan/orange lower) show the coordination gap at every level. GPT-5.4-mini's split-agent curves fall below Claude's, yielding larger raw gaps (38–49pp vs. 25–39pp for Claude variants).

structural bias, producing more divergent initializations that the naive merge cannot reconcile.

The coordination gap is persistent across both model families: Sonnet averages 31.5pp (range 29.5–34.8pp), Haiku averages 32.3 ± 0.8pp (range 25.5–38.8pp), and GPT-5.4-mini averages 41.9pp (range 37.8–48.6pp, all Wilcoxon $p < 0.001$). The gap is larger in absolute terms for GPT-5.4-mini, which is consistent with stronger bias responsiveness rather than a qualitatively different failure mode. The key observation is that the same qualitative effect — a large, statistically significant, persistent coordination penalty — reappears across two model families, two model sizes, and four specification levels. Within the bounds of our design, this supports interpreting the specification gap as a cross-family coordination problem rather than an artifact of any single provider's behavior.

5.5 Practical Implications

1. **Specification requirements**: Code agent systems should include data-structure constraints in specifications, not just behavioral descriptions. The recovery experiment confirms that this proved sufficient in our setting: a merger agent with the full specification recovers single-agent performance.
2. **Diagnostic, not therapeutic, conflict detection**: AST-based conflict detection is cheap (*diagnostic*: zero LLM cost) and tells engineers *where* and *why* coordination

is breaking down. It is not a substitute for richer specifications: conflict reports produced no statistically distinguishable improvement in our experiments (95% CI at L3: $[-4.7, +4.7]$pp). Systems should use the detector as a monitoring signal, not as input to a merger agent.

3. **Architecture guidance**: When specifications are incomplete (L2/L3), the most effective intervention is to enrich the specification, not to add post-hoc conflict analysis. When enrichment is infeasible, consider single-agent generation.

5.6 Threats to Validity

Internal — bias injection Agent biases are injected via system prompt rather than emerging organically. As discussed in Section 5, this models a realistic operational risk and the init-visibility experiment confirms that the coordination penalty persists even when both agents share identical constructor information, suggesting the gap is not an artifact of the injection mechanism.

Internal — information asymmetry Single agents receive the `__init__` body while split agents do not. The init-visibility experiment (Section 5.3) disentangles these two factors: coordination accounts for the larger share of the gap (+15.7–19.8pp) while information asymmetry contributes +11.2–15.3pp, with approximately additive interaction (−4.1pp).

Construct Our central construct is *coordination failure under partial knowledge*. We operationalize it with three complementary signals: integration pass rate, the single-vs.-split gap, and AST-detected structural conflicts. None is a perfect measure on its own. Test pass rate is only a proxy for integration quality: some tests may pass despite latent incompatibilities, or fail for reasons unrelated to coordination. The AST detector's 62.9% overall recall further shows that many failures are semantic rather than structural. We therefore avoid claiming that every failed merge is caused by the exact conflict types we detect; instead, we interpret convergence across these measures as evidence for a broader coordination construct.

External We test three models from two families: Claude Sonnet 4 and Haiku 4.5 (Anthropic) and GPT-5.4-mini[5] (OpenAI, $n = 25$ tasks). Three independent Haiku runs confirm low aggregate variance ($\leq$2pp), though individual task outcomes can vary substantially (per-task split-rate std up to 51pp). The cross-family replication with GPT-5.4-mini confirms that the coordination gap mechanism is not provider-specific, though the GPT-5.4-mini run uses only 25 tasks (a subset of the 50-task Sonnet$\cap$Haiku intersection) and cannot support the same statistical power as the Claude experiments. We therefore claim external validity at the level of *mechanism* rather than exact effect size: incomplete specifications reliably expose a coordination penalty in our setting, but the absolute magnitude may change for additional model families (*e.g.*, Gemini, Mistral), open-source models, other programming languages, orchestration schemes, or less stylized sources of bias. Generalization to those settings requires further study.

[5] Pinned: `gpt-5.4-mini-2026-03-17`

6 Conclusion

We studied how specification completeness affects integration success when multiple LLM agents independently generate parts of the same class. Across 51 tasks (25 for GPT-5.4-mini), three models from two families, and four specification levels, a persistent coordination gap (25–49pp) separates multi-agent from single-agent performance—even when specifications are detailed.

Our experiments reveal an asymmetry between diagnosis and repair. An AST-based conflict detector reliably identifies structural mismatches, and its precision improves precisely where specifications are weakest. Yet providing conflict reports to a merger agent adds no measurable benefit: in our setting, the full specification alone proved sufficient to recover the single-agent ceiling. When both are combined, the redundant diagnostic detail measurably decreases performance (-6.6pp, 95% CI excludes zero). A decomposition experiment further shows that the gap reflects genuine coordination difficulty (+16pp), not only hidden information (+11pp), and that the two effects are approximately additive.

These findings support a *specification-first* view of multi-agent orchestration. In multi-agent code generation, the specification functions as a coordination contract: when the contract loses structural information, agents produce locally plausible but globally incompatible implementations. Investing in richer specifications appears more consequential than investing in post-hoc conflict detection. Methodologically, our layered design—specification ablation, factorial recovery, and gap decomposition—provides a reusable template for evaluating coordination in multi-agent code generation. Whether richer forms of conflict representation, such as semantic diffs or iterative negotiation, can partially substitute for specification quality remains an open direction for future work.

Statements and Declarations

Funding. This work was supported by Project PID2022-138283NB-I00, funded by MICIU/AEI/10.13039/501100011033 and by FEDER, EU.

Competing Interests. The author has no relevant financial or non-financial interests to disclose.

Data availability. All experimental data (task specifications, agent outputs, integration results, and statistical analysis) are publicly available at https://github.com/camilochs/the_specification_gap.

Code availability. Source code for the experimental pipeline, agent configurations, AST-based conflict detector, and analysis scripts is publicly available at the repository above under an open-source license.

Author contribution. C. Chacón Sartori conceived the study, designed the experiments, implemented the pipeline, performed the analysis, and wrote the manuscript.

References

Austin J, Odena A, Nye M, et al (2021) Program synthesis with large language models. In: arXiv preprint arXiv:2108.07732

Binkley D, Horwitz S, Reps T (1995) Program integration for languages with procedure calls. ACM Transactions on Software Engineering and Methodology 4(1):3–35. https://doi.org/10.1145/201055.201056

Chen M, Tworek J, Jun H, et al (2021) Evaluating large language models trained on code. arXiv preprint arXiv:210703374

Conway ME (1968) How do committees invent? Datamation 14(4):28–31

Du X, Liu M, Wang K, et al (2024) ClassEval: A manually-crafted benchmark for evaluating LLMs on class-level code generation. In: Proceedings of the 46th International Conference on Software Engineering (ICSE)

Engler D, Chen DY, Hallem S, et al (2001) Bugs as deviant behavior: A general approach to inferring errors in systems code. In: Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP), pp 57–72, https://doi.org/10.1145/502034.502041

Hong S, Zhuge M, Chen J, et al (2024) MetaGPT: Meta programming for a multi-agent collaborative framework. arXiv preprint arXiv:230800352

Huang B, Cheng R, Tan KC (2025) EvoGit: Decentralized code evolution via git-based multi-agent collaboration. arXiv preprint arXiv:250602049

Islam MA, Ali ME, Parvez MR (2025) CodeSim: Multi-agent code generation and problem solving through simulation-driven planning and debugging. In: Findings of the Association for Computational Linguistics: NAACL 2025, pp 5128–5154, https://doi.org/10.18653/v1/2025.findings-naacl.285

Li R, Allal LB, Zi Y, et al (2023) StarCoder: May the source be with you! arXiv preprint arXiv:230506161

Mens T (2002) A state-of-the-art survey on software merging. IEEE Transactions on Software Engineering 28(5):449–462. https://doi.org/10.1109/TSE.2002.1000449

Meyer B (1992) Applying "Design by Contract". Computer 25(10):40–51. https://doi.org/10.1109/2.161279

Parnas DL (1972) On the criteria to be used in decomposing systems into modules. Communications of the ACM 15(12):1053–1058. https://doi.org/10.1145/361598.361623

Qian C, Liu W, Liu H, et al (2024) ChatDev: Communicative agents for software development. arXiv preprint arXiv:230707924

Rozière B, Gehring J, Gloeckle F, et al (2024) Code Llama: Open foundation models for code. arXiv preprint arXiv:230812950

Sartori CC (2026) Architectures of error: A philosophical inquiry into human and AI code. Philosophy & Technology 39(1):55. https://doi.org/10.1007/s13347-026-01056-x

Tang K, Li J, Li G, et al (2024) CodeAgent: Enhancing code generation with tool-integrated agent systems for real-world repo-level coding challenges. arXiv preprint arXiv:240107339

Zhu Y, Liu C, He X, et al (2025) AdaCoder: An adaptive planning and multi-agent framework for function-level code generation. arXiv preprint arXiv:250404220

Appendices

Additional materials are organized as follows: Appendix A provides the full conflict taxonomy; Appendix B presents a worked example; Appendices B.1 to B.5 detail the specification variants, prompts, outputs, conflict reports, and recovery conditions for that example; and Appendix C reports the prompts for the init-visibility experiment.

A Conflict Taxonomy

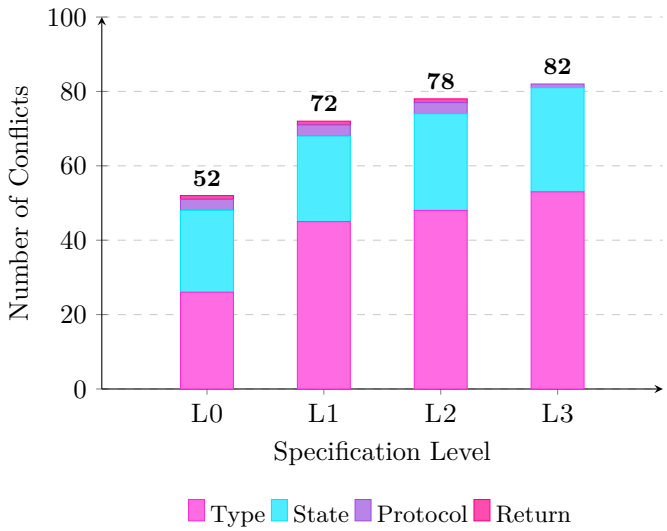


Fig. 9: Conflict taxonomy by specification level. Type conflicts (incompatible data structures) dominate at all levels and grow fastest as specifications degrade. Total conflicts increase from 52 at L0 to 82 at L3.

B Worked Example: Prompts, Outputs, and Conflicts

This appendix traces a single task—`AssessmentSystem` (ClassEval_4, 6 methods, 31 tests)—through every experimental condition, using the exact prompts and real LLM outputs from our experiment. The task is presented as an *illustrative case study* to make the mechanism concrete; it is not additional empirical evidence beyond the per-task aggregates already reported in Section 4. On this particular task the single agent passes 97% of tests, naïve merge passes 0%, and providing the full specification alone recovers to 97%—a clean instance of the broader pattern, but a single task and not generalizable on its own.

B.1 Specification Levels

The same class skeleton is presented at two extremes. At L0, the docstrings name the `students` dict, show its structure via doctests, and describe return types. At L3, only method signatures remain—agents must infer data structures from names alone.

L0 Skeleton — Full Specification

```
class AssessmentSystem:
    def __init__(self):
        """Initialize the students dict."""
        self.students = {}

    def add_student(self, name, grade, major):
        """
        Add a new student into self.students dict
        :param name: str, student name
        :param grade: int, student grade
        :param major: str, student major
        >>> system.add_student('student 1', 3, 'SE')
        >>> system.students
        {'student 1': {'name': 'student 1',
          'grade': 3, 'major': 'SE', 'courses': {}}}
        """

    def add_course_score(self, name, course, score):
        """
        Add score of specific course for student
        in self.students
        >>> system.add_course_score('student 1', 'math', 94)
        >>> system.students
        {'student 1': {'name': 'student 1', 'grade': 3,
          'major': 'SE', 'courses': {'math': 94}}}
        """

    def get_gpa(self, name):
        """Get average grade of one student.
        :return: float or None"""

    def get_all_students_with_fail_course(self):
        """Get all students who have any score below 60
        :return: list of str, student name"""

    def get_course_average(self, course):
        """Get average score of a specific course.
        :return: float or None"""

    def get_top_student(self):
        """Find student with highest GPA.
        :return: str, name of student"""

Docstrings explicitly reference ``self.students dict'' and show its nested
structure via doctests---the key coordination signal.
```

L3 Skeleton — Bare Signatures

```
class AssessmentSystem:

    def __init__(self):
        self.students = {}
```

```
    def add_student(self, name, grade, major):

    def add_course_score(self, name, course, score):

    def get_gpa(self, name):

    def get_all_students_with_fail_course(self):

    def get_course_average(self, course):

    def get_top_student(self):

No docstrings, no type hints. The __init__ body hints at a dict, but split
agents never see it---it is replaced with pass.
```

B.2 Agent System Prompts

Each agent receives a short system prompt establishing its role and bias, followed by a user message containing the skeleton and method assignments.

System Prompt — Agent A (List Bias)

```
You are an expert Python programmer implementing part of a Python class.
Another developer implements the remaining methods.
Design convention: store collections as LISTS (sequential, ordered). Use
list of dicts for structured data. Iterate/filter to find entries.
IMPORTANT: - Define __init__ yourself with appropriate instance variables
- Implement ONLY your assigned methods - For other methods write: def
method_name(self): pass - Return ONLY Python code, no explanation
```

System Prompt — Agent B (Dict Bias)

```
You are an expert Python programmer implementing part of a Python class.
Another developer implements the remaining methods.
Design convention: store collections as DICTIONARIES (key-value mapping).
Map identifiers to data records for fast lookup and membership checks.
IMPORTANT: - Define __init__ yourself with appropriate instance variables
- Implement ONLY your assigned methods - For other methods write: def
method_name(self): pass - Return ONLY Python code, no explanation
```

Split agents receive the L3 skeleton *with* `__init__` *replaced by* `pass`. They must define their own constructor. The user message also lists which methods to implement and which belong to the collaborator.

B.3 Agent Outputs (Real LLM Responses)

The following are verbatim outputs from Claude Sonnet 4 for this task. Agent A initializes all fields as **lists**; Agent B uses **dicts**. The conflict is visible in the `__init__` of each output.

Agent A Output — List Bias

```
class Unknown:
    def __init__(self):
        self.students = []          # <-- LIST
```

```
        self.courses = []           # <-- LIST
        self.scores = []            # <-- LIST

    def add_student(self, student_id, name):
        student = {'student_id': student_id,
                   'name': name}
        self.students.append(student)  # list.append()

    def get_gpa(self, student_id):
        student_scores = [s for s in self.scores
                          if s['student_id'] == student_id]
        if not student_scores:
            return 0.0
        return sum(s['score'] for s in student_scores) \
               / len(student_scores)

    def get_course_average(self, course_id):
        course_scores = [s for s in self.scores
                         if s['course_id'] == course_id]
        if not course_scores:
            return 0.0
        return sum(s['score'] for s in course_scores) \
               / len(course_scores)

    # Stubs for collaborator's methods
    def add_course_score(self): pass
    def get_all_students_with_fail_course(self): pass
    def get_top_student(self): pass
```

Agent B Output — Dict Bias

```
class Unknown:
    def __init__(self):
        self.students = {}          # <-- DICT
        self.courses = {}           # <-- DICT
        self.scores = {}            # <-- DICT (tuple keys)

    def add_course_score(self, student_id,
                         course_id, score):
        self.scores[(student_id, course_id)] = score

    def get_all_students_with_fail_course(self,
            failing_threshold=60):
        failing = {}
        for (sid, cid), score in self.scores.items():
            if score < failing_threshold:
                if sid not in failing:
                    failing[sid] = []
                failing[sid].append(cid)
        return failing

    def get_top_student(self):
        if not self.scores:
            return None
        totals, counts = {}, {}
        for (sid, cid), score in self.scores.items():
            totals[sid] = totals.get(sid, 0) + score
            counts[sid] = counts.get(sid, 0) + 1
        best, best_avg = None, -1
```

```
        for sid in totals:
            avg = totals[sid] / counts[sid]
            if avg > best_avg:
                best_avg, best = avg, sid
        return best

    # Stubs for collaborator's methods
    def add_student(self): pass
    def get_gpa(self): pass
    def get_course_average(self): pass
```

Naïve merge result: 0/31 tests pass. When Agent A's list-based `__init__` is combined with Agent B's dict-based methods, every operation crashes: `self.scores[(sid, cid)] = score` fails on a list, and `self.students.append()` fails on a dict.

B.4 AST Conflict Detection Output

The AST detector compares the two agents' code *before integration* and reports:

AST Conflict Report — 4 Conflicts Detected (3 Type + 1 State)

```
Conflict 1 [TYPE, HIGH]: field self.students
  Agent A: Initializes as list
  Agent B: Initializes as dict

Conflict 2 [TYPE, HIGH]: field self.courses
  Agent A: Initializes as list
  Agent B: Initializes as dict

Conflict 3 [TYPE, HIGH]: field self.scores
  Agent A: Initializes as list
  Agent B: Initializes as dict

Conflict 4 [STATE, LOW]: field self.students
  Cross-boundary state dependency with mutations
  Agent A: Operations: ['append']
  Agent B: Operations: read/write only

All three type conflicts are correctly identified. The state conflict
detects that Agent A mutates self.students with append (a list operation),
while Agent B only reads/writes it as a dict.
```

B.5 Recovery Conditions

The merger agent receives the two conflicting implementations and must produce a unified class. The key difference between conditions is what information the merger receives.

Spec-Only Merger — L0 Spec, No Conflict Report

```
You are an expert Python programmer. Combine two developers'
implementations into a single correct class. Use the full specification to
guide your merge. Return ONLY Python code.
--- User message includes L0 skeleton (with docstrings) --- + Agent A code
+ Agent B code --- NO conflict report provided
```

Result: 96.8% (30/31 tests) — full recovery.

Guided Merger — L3 Spec, With Conflict Report

```
You are an expert Python programmer. Combine two developers'
implementations into a single correct class. Pay attention to the detected
conflicts. Return ONLY Python code.
--- User message includes L3 skeleton (bare signatures) --- + Agent A code
+ Agent B code --- + full conflict report (see B.4)
```

Result: 6.5% (2/31 tests) — conflict report hurts without spec.

Table 7 summarizes the results for this task across all conditions.

Table 7: Results for AssessmentSystem across all experimental conditions (31 tests).

Condition	Spec / Conflicts	Tests	Pass rate
Single	L0 / —	30/31	96.8%
Naïve	(concatenation)	0/31	0.0%
Blind	L3 / no	14/31	45.2%
Guided	L3 / yes	2/31	6.5%
Spec-Only	L0 / no	30/31	96.8%
Resolve	L0 / yes	30/31	96.8%

The pattern is stark: the L0 specification alone enables full recovery (Spec-Only = Single = 96.8%), while the conflict report without the specification actually *hurts* (Guided: 6.5% vs. Blind: 45.2%). With the full specification available, the merger infers the correct dict-based structure from the docstring "Add a new student into `self.students` dict" without needing any explicit conflict report.

C Init Visibility Experiment Prompts

This appendix shows the prompts used in the init-visibility experiment (Section 5.3). The key difference from the main experiment is that the "visible" conditions provide agents with the real `__init__` body.

C.1 Single-Hidden Condition

The single agent receives the skeleton with `__init__` replaced by `pass`—identical to what split agents see in the main experiment—and must define its own data structures without any bias.

System Prompt — Single Agent (Hidden Init)

```
You are an expert Python programmer. Implement all methods of the given
class. Define __init__ yourself with appropriate instance variables.
Return ONLY Python code, no explanation.
```

C.2 Split-Visible Condition

Split agents receive the *full* skeleton including the `__init__` body, and are explicitly told to keep it exactly as provided. Crucially, they still carry the list/dict bias—the question is whether the visible constructor overrides their prior.

System Prompt — Agent A (Visible Init, List Bias)

```
You are an expert Python programmer implementing part of a Python class.
Another developer implements the remaining methods.
Design convention: store collections as LISTS (sequential, ordered). Use
list of dicts for structured data. Iterate/filter to find entries.
IMPORTANT: - Keep __init__ EXACTLY as provided - Implement ONLY your
assigned methods - For other methods write: def method_name(self): pass -
Return ONLY Python code, no explanation
```

The user message for split-visible agents includes the constructor body explicitly:

User Message Excerpt — Split-Visible Agent

```
Class: AssessmentSystem
Description: ...

Constructor (keep EXACTLY as provided):
    def __init__(self):
        self.students = {}

YOUR methods to implement:
    def add_student(self, name, grade, major):
        """Add a new student..."""
    ...
```

By providing the real `__init__` body, this condition tests whether a shared constructor can anchor agents to a common data structure despite their opposing biases.