Precedent or Intent? What Coding Agents Follow When Evolving Existing Code

Haichao Zhu[2] **Geyang Xu**[1,*] **Zhaorui Yang**[1] **Jiyuan Wang**[3] **Qian Zhang**[1]

[1]University of California, Riverside [2]Independent Researcher [3]Tulane University

Abstract

Existing code reveals past implementation decisions but rarely explains the reasoning behind them. When coding agents implement new features or modify existing functionality, past decisions may conflict with new task instructions. This raises a fundamental question for agentic software evolution: when multiple implementations satisfy the same functional requirements, how do coding agents decide whether to follow existing code or new task instructions?

We investigate this question through bidirectional, controlled experiments in purpose-built repositories, followed by validation in real-world open-source repositories. Our approach isolates three competing sources of guidance: (1) Agents' Default Preferences, measured without repository precedents; (2) Repository Precedents, examined by manipulating the frequency and location of existing code examples; and (3) Explicit Task Instructions, tested by introducing rules that conflict with existing code.

We conduct 5,625 trials across four coding agents and six implementation pairs, with validation in four open-source repositories spanning three programming languages. We find that agents share the same default implementation choices, yet overwhelmingly follow repository precedent when it conflicts with those defaults (359/360 trials). Crucially, not all precedents carry equal influence: three supporting local files yield 96% adoption, compared with only 4% for sibling-resource files, and changing file placement alone can shift adoption from 0% to 100% at a fixed example count. When new task intent conflicts with existing code, written priority rules achieve substantially higher success than code examples alone (59/60 versus 38/60 joint successes for two agents on two implementation pairs). The effectiveness of explicit rules also extends to open-source repositories, where a harness-loaded rule raises joint success from 0/120 to 117/120. These findings motivate design-decision traceability that preserves not only implementation choices but also the rationale and priorities that should guide future changes.

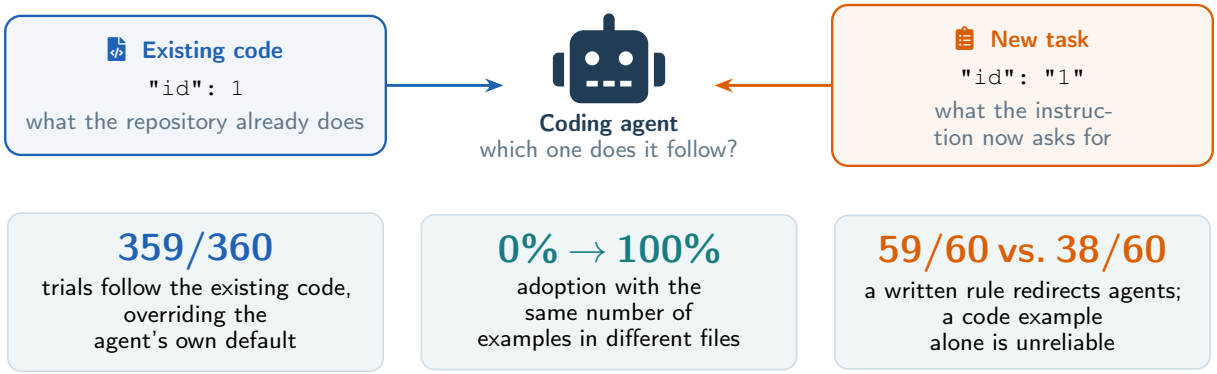


Figure 1: Coding agents that extend a repository are pulled between the implementation already in the code and the intent of the new task. In controlled experiments with four agents and six implementation pairs, existing code wins by default, and a written rule is what reliably changes course.

1 Introduction

Software development involves the continuous evolution of existing systems. As LLM-based coding agents take on increasingly complex software engineering tasks, their role is expanding beyond individual issue resolution to implementing changes across evolving repositories [35, 13, 40, 6, 15]. Unlike building from scratch, extending an existing repository means working inside decisions made during earlier development, whose rationale is often lost. These decisions influence how new functionality is implemented, even when multiple alternatives satisfy the same functional requirements [6, 40]. However,

*Corresponding author: gxu048@ucr.edu.

existing code rarely explains why a particular implementation was chosen or whether that decision should be preserved as requirements evolve. Coding agents must therefore reconcile past implementation decisions with the intent expressed in new task specifications.

Past Decisions vs. New Intent. Existing code records past implementation decisions. *Every new change inherits decisions made in the past, but how does a coding agent determine which decisions to preserve, particularly when a new task expresses a different intent?* This question raises two challenges in agentic software evolution.

First, the rationale behind past implementation decisions is often unavailable. Existing code reveals what previous developers chose, but rarely why [17, 38, 24]. For example, a repository may use an if-else chain rather than a dictionary lookup because of an earlier requirement or simply a developer's preference. Without this context, a coding agent cannot determine whether the existing code reflects a constraint that should be preserved.

Second, past implementation decisions may conflict with the new task intent. Different parts of a repository may follow different implementation patterns, while new task instructions may favor an alternative [30, 16]. For example, an agent adding an API endpoint may encounter existing handlers that serialize identifiers as integers, while the new task may specify a preference for string identifiers. Both options may satisfy functional requirements, but represent different implementation choices. The agent must determine whether to follow the existing precedent or the new instructions.

Despite substantial progress in evaluating coding agents in long-horizon software engineering tasks [13, 15, 40], how agents prioritize past implementation decisions and new task instructions remains poorly understood. In particular, we lack systematic evidence on the relative influence of *agents' default preferences, implementation patterns in task-related code and elsewhere in the repository, and explicit task guidance.*

Approach. We investigate how coding agents prioritize competing sources of guidance when modifying existing software repositories. We examine three factors: (1) agents' default implementation preferences, (2) repository precedent established by past implementations, and (3) explicit guidance in new task specifications. Our approach combines bidirectional controlled experiments in purpose-built repositories with a validation study in real-world open-source repositories.

We first establish agents' default preferences in a manually constructed Python/Flask repository. We design six implementation-choice cases that span naming, data representation, error handling, and control flow, each allowing two functionally correct alternatives, such as integer versus string identifiers. By removing relevant repository examples, we establish a baseline for two complementary controlled interventions:

- *Repository Precedent.* We examine how historical implementations shift agents' default choices. We distinguish local precedent, established by existing endpoints for the resource being extended, from sibling-resource precedent, established by endpoints for other resources in the same repository. Holding task requirements fixed, we vary the number and placement of supporting examples and introduce conflicting patterns across resources. These interventions isolate how the prevalence and location of existing patterns influence agents' choices.
- *Explicit Task Guidance.* We examine whether new instructions can override established precedent. Starting with code that favors one implementation, we compare four guidance conditions: no guidance, a written priority rule, a code example favoring the alternative, and both combined. We further test whether a rule reaches the agent equally well through the task description or through the harness's native instruction file. This reveals how different forms and channels of guidance redirect agents toward a new task intent.

Key Findings. Our study comprises 5,625 trials across six implementation pairs and four models: Claude Sonnet-4-6, Claude Opus-4-8, Codex gpt-5.6-sol, and Codex gpt-6-sol. We further evaluate four open-source repositories spanning Python, Go, and TypeScript: FastAPI RealWorld [20], FastAPI Clean Example [11], Go REST API [23], and ExpressJS API [14]. Three key findings emerge, highlighting the need to make implementation decisions and their intended priority explicit during software evolution.

First, *repository precedent overturns agents' shared default choices in 359/360 trials.* Without relevant repository precedents, all four agents favor the same implementation in each of the six pairs. Yet when all existing endpoints consistently use the opposite implementation, agents follow that precedent in 359/360 trials, despite unchanged functional requirements. In four open-source repositories, agents follow local precedent in all 945 functionally successful, classifiable error-handling implementations and 355 of 417 (85%) functionally successful response-envelope implementations.

(a) Existing local handler: `tasks_list.py`

```
1 @bp.get("/projects/<int:pid>/tasks")
2 def list_tasks(pid):
3     tasks = Task.query.filter_by(
4         project_id=pid).all()
5     return jsonify([
6         {"id": t.id,
7          "name": t.name}
8         for t in tasks
9     ])
```

(b) New handler: follows precedent

```
1 @bp.get("/projects/<int:pid>"
2         "/tasks/<int:tid>")
3 def get_task(pid, tid):
4     task = Task.query.filter_by(
5         id=tid, project_id=pid
6     ).first_or_404()
7     return jsonify({
8         "id": task.id,
9         "name": task.name
10    })
```

(c) New handler: follows new intent

```
1 @bp.get("/projects/<int:pid>"
2         "/tasks/<int:tid>")
3 def get_task(pid, tid):
4     task = Task.query.filter_by(
5         id=tid, project_id=pid
6     ).first_or_404()
7     return jsonify({
8         "id": str(task.id),
9         "name": task.name
10    })
```

Figure 2: Illustrative Flask handlers for the motivating example. (a) An existing local handler serializes task IDs as numbers (line 6). (b) and (c) Two implementations of the new endpoint retrieve the same task with an identical query (lines 4–6) and differ only in how they serialize the identifier (line 8). (b) carries over the repository precedent (blue), whereas (c) follows a new instruction to serialize identifiers as strings (orange).

Second, *a single well-placed example can outweigh multiple examples elsewhere.* With three supporting files, local precedents yield 96% adoption (347/360), compared with only 4% (13/360) for sibling-resource precedents. Even within local code, changing file placement while holding the number of examples fixed can shift adoption from 0% to 100%. In one setting, a single well-placed example even outperforms two examples in other files. These findings show that the influence of repository precedent cannot be explained by example frequency alone.

Third, *written priority rules override conflicting precedent more reliably than code examples alone (98% vs. 63%).* When all existing endpoints favor the opposite implementation, written priority rules achieve both functional success and target adoption in 59/60 trials, compared with 38/60 for code examples alone and 0/60 without guidance. The effectiveness of written rules also extends to four open-source repositories, where a rule in the harness's native instruction file raises joint success from 0/120 to 117/120.

Figure 1 summarizes these findings. This paper makes three contributions:

- We establish the prioritization of historical code and new task intent as a research problem in software evolution, introducing bidirectional controlled experiments to isolate their influence on coding agents.
- We reveal that local precedent overrides default preferences and that where examples are placed can change adoption from 0% to 100% at a fixed example count.
- We demonstrate that explicit rules override conflicting repository precedent more reliably than examples, motivating design-decision traceability to preserve implementation intent and priorities.

2 Motivating Example

Consider a coding agent asked to add the endpoint `GET /projects/<id>/tasks/<id>` to an existing Flask API in Figure 2. The endpoint retrieves a task and returns its information as JSON. Although the required behavioral specification is given, some concrete implementation decisions remain open. For example, the task identifier may be serialized as a JSON number (Figure 2(b)) or as a string (Figure 2(c)).

Past Decisions as Precedent. The existing task handler in Figure 2(a) serializes the identifiers as numbers using `task.id`. Such existing implementations establish *repository precedents*: past implementation decisions that may guide subsequent changes. An agent following this precedent would produce Figure 2(b). However, a repository does not necessarily follow a single implementation pattern across locations within a codebase. The two ID representations illustrated in Figure 2(b) and (c) may coexist in existing code, appearing in different files or even within the same file. The agent may therefore encounter conflicting historical implementations rather than a single pattern to follow.

New Intent. Following existing implementation decisions helps maintain consistency as software evolves. For example, reusing the numeric-ID representation in Figure 2(a) would keep the new endpoint consistent with existing task responses.

However, new requirements may require changes to earlier design choices. Suppose the new task explicitly states: "Serialize task identifiers as strings in the new endpoint. This instruction takes precedence over the ID representations used by existing handlers." The agent now *faces a conflict: following the existing precedent preserves consistency but contradicts the new instruction, whereas following the instruction satisfies the new intent but departs from the established implementations.*

This illustrates a fundamental challenge in software evolution: how coding agents choose between preserving past implementation decisions and following newly specified intent.

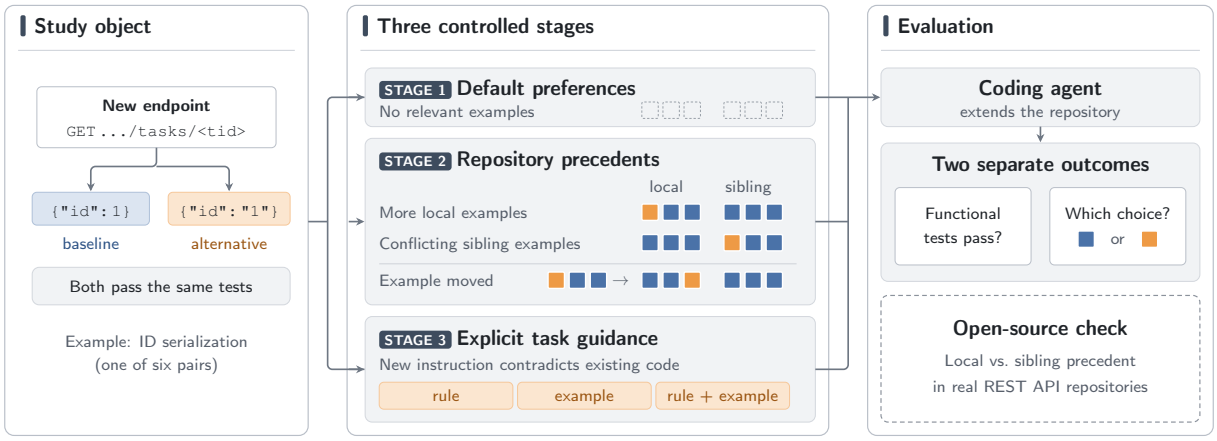


Figure 3: Research approach. Each task admits two functionally acceptable implementations (baseline in blue, alternative in orange). Within the controlled study, Stage 1 measures default choices without relevant examples, Stage 2 varies repository precedents in local and sibling-resource files (including which file holds the example at a fixed count), and Stage 3 adds task guidance that contradicts existing code. Every edit is scored separately for functional success and implementation choice; the local-versus-sibling comparison is also checked in open-source repositories.

The Research Problem. The resulting code reveals which implementation the agent selected, but not what guided that choice. An agent may follow its default preference, adopt a precedent found in the repository, or follow an explicit task instruction. Our research investigates how these three sources of guidance influence implementation choices, particularly when historical precedents and new task intent conflict. To distinguish their effects, we systematically vary repository precedents and task guidance while keeping the functional requirements unchanged.

3 Approach

3.1 Overview

We design a three-stage controlled study that separates the sources of guidance a coding agent may follow when it extends an existing repository (Figure 3). The design rests on one principle: every stage keeps the functional task and its hidden tests unchanged and varies only the guidance available to the agent. A change in the agent's implementation choice can therefore be attributed to the varied guidance rather than to a change in what the code must do.

Study Object. Each study task admits two functionally acceptable implementations, which we call an *implementation pair*. Before running any agent, we designate one implementation as the *baseline* and the other as the *alternative*. These labels are fixed in advance and do not encode which implementation an agent prefers. Pair-specific hidden tests accept either implementation, so the functional requirements stay fixed while the implementation choice remains open. For example, in the ID serialization case, the tests verify that the returned identifier matches the expected task ID, accepting either its numeric or string representation. We study six pairs spanning naming, data representation, and code structure (Table 1), and use ID serialization from Section 2 as the running example: a new endpoint may serialize task identifiers as integers (baseline) or strings (alternative).

Controlled Repository. All three stages use a purpose-built Flask Task Manager API. In its full form, the repository contains three endpoints for `tasks` (create, update, and list) and three for `projects` (create, update, and list), each in its own file, together with the data model, database setup, and application scaffold. The agent is asked to add a new endpoint for `task`, `GET /projects/<id>/tasks/<id>`. We therefore call the existing `task` endpoints *local* files, because they belong to the resource being extended, and the `project` endpoints *sibling-resource* files, because they implement an analogous resource that the new endpoint does not modify. Where an implementation pair requires additional functionality, we minimally extend the agent's assignment. For example, the field-casing case additionally requires the endpoint to return the parent `project` ID, and the Boolean-naming case requires adding a domain-model field.

Three Stages. Each stage (Figure 3) isolates one source of guidance. Stage 1 removes all existing endpoints to measure default choices (Section 3.2); Stage 2 controls how many and which local and sibling-resource files use the alternative (Section 3.3); and Stage 3 sets every endpoint opposite a target and supplies

Table 1: Implementation pairs. Both implementations in a pair pass the same hidden functional tests; adoption refers to the alternative.

Category	Pair	Baseline	Alternative
Naming	Field casing	`project_id`	`projectId`
	Boolean naming	`completed`	`is_completed`
Data representation	ID serialization	`"id": 1`	`"id": "1"`
	Response envelope	`{...}`	`{"data": {...}}`
Code structure	Error handling	`return ..., 404`	`raise NotFound`
	Dispatch	`if/elif chain`	`LABELS[priority]`

explicit guidance through the task description or the harness's native instruction file (Section 3.4). All stages assess functional success and implementation choice, as described in Section 3.5.

3.2 Stage 1: Establishing Default Preferences

Zero-Precedent Repository. Stage 1 establishes agents' default implementation choices without relevant repository precedents. We remove all six existing task and project endpoint files from the controlled repository while retaining the data model, database setup, and application scaffold. The resulting *zero-precedent* repository remains sufficient for implementing the new endpoint but contains no relevant endpoint examples. We construct one variant per implementation pair, retaining the necessary pair-specific task extensions.

For each pair and agent, we run independent sessions on fresh repository copies, using the same functional task as in Stage 2 without prescribing an implementation choice. We classify the resulting implementations (Section 3.5) and define the option chosen in more than half of an agent's sessions as its default. These defaults provide a reference for measuring the influence of repository precedent and explicit guidance in subsequent stages, independently of the predefined baseline and alternative labels.

In the running example, the `task` model retains its integer primary key, but no existing handler demonstrates whether IDs should be returned as `task.id` or `str(task.id)`. In this setting, all four models default to returning numeric IDs (`task.id`), as confirmed by the zero-precedent results in Section 4.2.

3.3 Stage 2: Manipulating Repository Precedents

Stage 2 restores the six endpoints and rewrites the construct under study in a controlled subset of them. For ID serialization, we change an existing endpoint from `"id": task.id` to `"id": str(task.id)`; all other code in that endpoint stays identical. The other pairs are rewritten analogously, changing only the construct under study. Throughout Stage 2, the task description asks the agent to follow the conventions of the existing code without naming the implementation choice, mirroring common contribution guidelines. Any shift toward the alternative must therefore come from the code itself.

Support Level and Conditions. We call the number of files in a group (local or sibling-resource) that use the alternative the group's *support level* ($k/3$). For each pair, we construct eight conditions (Figure 4, upper rows):

- *Agreeing groups:* all six existing endpoints use the baseline ("Both baseline"; $0/3$, $0/3$) or the alternative ("Both alternative"; $3/3$, $3/3$).
- *Local support:* local support $1/3$, $2/3$, or $3/3$, with all sibling-resource files at the baseline.
- *Sibling-resource support:* sibling-resource support $1/3$, $2/3$, or $3/3$, with all local files at the baseline.

We call this design *bidirectional* because each group is varied while the other is held at the baseline, so the influence of the two groups can be compared at the same support level. Whenever one group supports the alternative, the two groups conflict; at $3/3$ they disagree completely, and comparing local $3/3$ with sibling-resource $3/3$ shows which group prevails in a direct conflict. Comparing these conflicting conditions with the agreeing ones shows whether agreement between the groups strengthens adoption.

File Identity. In the eight conditions, each intermediate support level uses one fixed assignment of the alternative to files; at local support $1/3$, for example, it appears in `tasks_list` (Figure 4, "default").

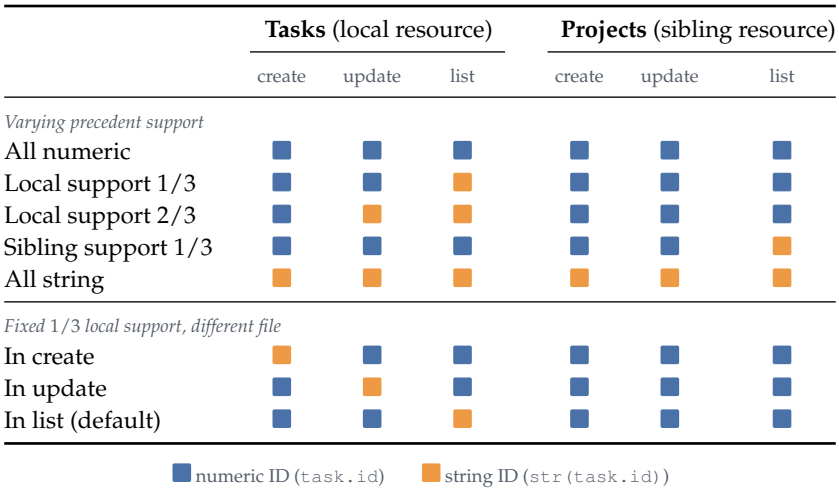


Figure 4: Illustrative repository interventions for ID serialization. Each square is one existing endpoint file. The upper rows vary how many local or sibling-resource files use string IDs; the lower rows hold the local support level at 1/3 and change only which file carries the alternative ("default" marks the assignment used for local support 1/3 above). The requested new endpoint is absent from every variant.

Raising the support level therefore also changes *which* files carry the alternative, so example count and file identity are confounded. To separate them, we hold the support level fixed and vary only the placement: at 1/3 support, we place the alternative in the create, update, or list endpoint (Figure 4, lower rows); at 2/3 support, in each of the three pairs of these endpoints. We first conduct this analysis exploratorily for ID serialization with one agent, including a control that changes only module registration and import order, and then replicate it in three further pair–agent settings (Section 4.3).

3.4 Stage 3: Introducing Explicit Task Guidance

Stage 3 asks whether new task intent can override repository precedent when the two conflict. We designate a *target* implementation and set all six existing endpoints to the opposite implementation, so that no file in the repository supports the target and any adoption must come from the guidance. In the running example, all existing endpoints serialize IDs as integers, whereas the target uses strings. The functional task and hidden tests remain unchanged.

We compare four guidance conditions, all supplied in the initial task description:

- **C0 (no guidance):** the task does not mention the target; this is the control.
- **C1 (written rule):** a statement of the target and its priority over existing code, without code, such as *"Serialize task identifiers as strings in the new endpoint. This instruction takes precedence over the ID representations used by existing handlers."*
- **C2 (code example):** a snippet that demonstrates the target, such as `"id": str(task.id)`, without stating its priority.
- **C3 (rule and example):** both C1 and C2.

Comparing C1 with C2 separates *stating* which implementation should govern a new change from *demonstrating* what that implementation looks like; comparing C3 with C1 and C2 shows whether the two forms complement each other. A separate experiment varies the delivery channel: the same written rule appears either in the task description or in the harness's native instruction file (`CLAUDE.md` or `AGENTS.md`), with the task description otherwise unchanged.

3.5 Measuring Outcomes

We evaluate each generated edit on two separate outcomes. *Functional success* indicates whether the edit passes the pair-specific hidden tests, which accept either implementation; sessions that produce no implementation count as functional failures. *Implementation choice* indicates which alternative the edit adopts, determined by pair-specific classifiers that inspect the produced code. Outputs that use neither or both alternatives are reported as unclassifiable rather than assigned a label. For Stages 1 and 2, we report the *adoption rate*: the share of classifiable outputs that use the designated alternative. For Stage 3, where the task prescribes a target, we use a *joint outcome* that requires both functional success and adoption of

Table 2: Experimental scale: 5,625 trials in 375 cells, each run in 15 independent sessions on fresh repository instances.

Study	Design	Trials
Controlled repository		
RQ1: zero precedent	6 pairs × 4 agents	360
RQ2/3: support levels	6 pairs × 4 agents × 8 cond.	2,880
RQ2: file identity (expl.)	4 placements + order control	75
RQ2: file identity (repl.)	3 settings × 6 placements	270
RQ4: explicit guidance	2 pairs × 2 agents × 4 cond.	240
RQ4: native-file rule	2 pairs × 2 agents × 3 cond.	180
Open-source repositories		
Error handling	4 repos × 4 assign. × 4 agents	960
Response envelope	14 assign.[a] × 2 agents	420
Native-file rule (envelope)	4 repos × 2 agents × 2 cond.	240
Total		**5,625**

[a]4 assignments in each of 3 repositories with a sibling resource, 2 in `fastapi-clean`.

the target. Separating the two outcomes prevents a passing test suite from hiding which implementation an agent selected.

3.6 Open-Source Repository Validation

The controlled repository is small and written in a single language. We therefore test whether its central Stage 2 contrast, local versus sibling-resource precedent, also holds in established codebases. We select four open-source REST API repositories in Python, Go, and TypeScript and study two implementation choices that occur across their frameworks. In *error handling*, an endpoint either propagates a typed error to a central handler (P) or constructs the error response directly (D); for example, it may raise a not-found error that a central handler converts into a response, or build an HTTP 404 response itself. In *response envelope*, an endpoint returns either a flat JSON object (F) or the payload under a sole top-level `data` key (D).

Following Stage 2, we rewrite the existing handlers of the resource being extended (local) and of an analogous resource (sibling) to an assigned pattern, independently of each other. This yields four configurations in which the two groups agree or conflict (e.g., PP, PD, DP, and DD, with the local pattern first); in a repository without a sibling resource, only the local pattern varies. The agent then implements a new local endpoint without guidance about the choice. We evaluate functional success with repository-specific hidden tests and classify the resulting implementation. Because each repository fixes the number and identity of its handlers, this study tests the influence of local versus sibling-resource precedent, but not the support level or file placement, for which the controlled repository remains the source of evidence. We also test a Stage 3 counterpart in the same repositories. With every local handler returning a flat payload, we ask whether a rule in the harness's native instruction file makes the agent wrap the payload of a new endpoint under a sole top-level `data` key, compared with a contemporaneous no-guidance control. Because each repository fixes its own handlers, this test measures the effect of the rule against real conflicting code, not the support level or file placement.

4 Evaluation

We answer four research questions:

- **RQ1 (Default Preferences):** Which implementation do coding agents choose among functionally equivalent alternatives without relevant repository precedent?
- **RQ2 (Local Precedent):** How do implementation patterns in task-related code influence agents' choices?
- **RQ3 (Sibling-Resource Precedent):** How do patterns in sibling resources affect implementation choices, particularly when they conflict with local precedent?
- **RQ4 (Explicit Guidance):** Can explicit rules or examples steer agents toward a specified implementation despite conflicting repository precedent?

RQ1–RQ4 use the controlled repository; Section 4.6 tests the local-versus-sibling contrast of RQ2 and RQ3, and the effect of an explicit rule (RQ4), in open-source repositories.

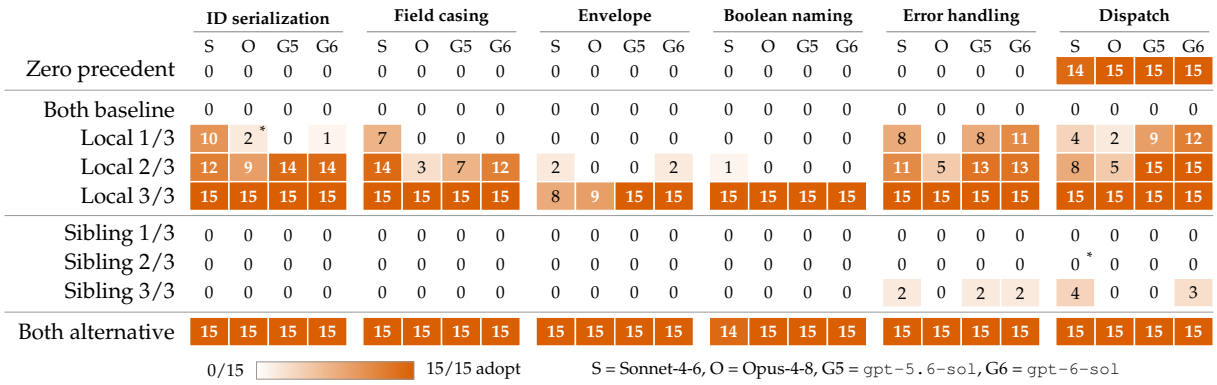


Figure 5: Adoption of the designated alternative in the controlled repository (RQ1–RQ3). Each cell gives the number of runs, out of 15 classifiable runs (* = 14), that adopt the alternative; shading is proportional to the adoption rate. "Local $k/3$" and "Sibling $k/3$" vary the number of files in that group that use the alternative while the other group stays at the baseline; at intermediate levels the alternative appears in a fixed set of files. For dispatch, the designated alternative (dictionary lookup) is the agents' default, so the zero-precedent row is shaded.

4.1 Experimental Setup

Agents. We run Claude Sonnet-4-6 and Claude Opus-4-8 through Claude Code [2], and `gpt-5.6-sol` and `gpt-6-sol` through Codex CLI [21]. We refer to each model–harness combination as an *agent*.

Trials. Each trial is an independent agent session on a fresh repository instance: the agent explores and edits the repository autonomously, and we run the saved result against the hidden functional tests and the pair-specific classifier. Each experimental cell is run 15 times, for 5,625 trials in total (Table 2).

Analysis. Adoption rates are computed over classifiable outputs. For RQ1–RQ3, we report exact binomial intervals, Cochran–Armitage tests for trends across support levels, and Fisher exact tests for pairwise comparisons; the tests specific to file placement, RQ4, and the open-source study are given in the corresponding subsections. Figure 5 reports the adoption counts of every RQ1–RQ3 cell.

4.2 RQ1: Default Implementation Preferences

Setup. We run each pair on its zero-precedent repository (Section 3.2) with all four agents, for 360 trials. All 360 runs pass the hidden tests and are classifiable.

Default Choices. Without repository precedent, agents serialize identifiers as integers, name response fields in `snake_case`, return the resource as a flat JSON object, name Boolean fields without an `is_` prefix, construct error responses directly in the handler, and dispatch priority labels through a dictionary lookup (Table 3). In dispatch, the default is the designated alternative, which is why the corresponding zero-precedent cells in Figure 5 are shaded. The six defaults share one property: each is the most direct expression of the required behavior in Python and Flask. Agents return the model attribute as stored (`task.id`, `project_id`), return the payload without a wrapper, use the shorter Boolean name, handle an error where it arises, and map values through a lookup table. Each alternative instead adds a conversion, a wrapper, a naming convention, or an indirection that the task itself does not motivate. Choosing such an alternative therefore requires information from outside the task, which is precisely what repository precedent and task guidance supply.

Consistency. The defaults are near-deterministic and agent-independent. Of the 360 runs, 359 choose the default, and 23 of the 24 pair–agent cells do so unanimously; the only deviation is a single Sonnet run on dispatch. All four agents, from two providers and running in two harnesses, share the same default in every pair. Differences among agents in RQ2–RQ4 therefore do not stem from different starting preferences.

Defaults vs. Precedent. Repository precedent overrides these defaults. When all six existing endpoints use the implementation opposite the default, agents follow the repository in 359 of 360 runs: 299/300 in the five pairs whose default is the baseline ("Both alternative" in Figure 5) and 60/60 in dispatch ("Both baseline"). Dictionary dispatch, chosen in 59 of 60 zero-precedent runs, is chosen in none of 60 runs once every endpoint uses an `if/elif` chain.

Answer to RQ1. Without relevant repository precedents, all four agents favor the same implementation in each of the six pairs: integer IDs, `snake_case` fields, flat responses, unprefixed Boolean names, errors returned directly in the handler, and dictionary-based dispatch. These default choices are nearly deterministic (359/360 trials). Yet when all existing endpoints use the opposite implementation, agents follow repository precedent in 359/360 trials.

Implication: Implementation preferences measured without relevant repository precedents do not necessarily predict agents' choices when extending existing code.

4.3 RQ2: Influence of Local Precedent

Setup. We use the local-support conditions of Stage 2 (Section 3.3), in which zero to three local files use the alternative while the sibling-resource files stay at the baseline, for all six pairs and four agents (2,880 trials shared with RQ3). For file placement, we move the alternative among local files at 1/3 and 2/3 support, first exploratorily for ID serialization with Sonnet (75 trials, including an import-order control) and then in three replication settings, ID serialization with `gpt-6-sol` and with Opus-4-8, and field casing with Sonnet (270 trials). We compare placements in the replications with omnibus within-block Monte Carlo randomization tests (99,999 randomizations) and Holm correction over the six registered tests (two support levels in each setting).

Support Level. Adoption increases with local support. Pooled over six pairs and four agents, adoption of the alternative rises from 0% (0/360) without local support to 21% (74/359) at 1/3, 49% (175/360) at 2/3, and 96% (347/360) at 3/3 (Figure 5). Adoption never decreases with local support in any of the 24 pair–agent combinations, and Cochran–Armitage tests find positive trends for all six pairs with Sonnet ($p \leq 5.6 \times 10^{-5}$) and Opus ($p \leq 1.3 \times 10^{-5}$). The curves share a direction but not a threshold. A single local file suffices for Sonnet to adopt string IDs (10/15), centrally handled errors (8/15), and `camelCase` (7/15), whereas Boolean naming stays at or below 1/15 through 2/3 support for every agent and reaches 15/15 only at 3/3. Response envelope never reaches full adoption for the Claude agents (Sonnet 8/15, Opus 9/15 at 3/3). Agents also differ in which pairs they respond to: at 1/3 support, Sonnet adopts string IDs more often than `gpt-6-sol` (10/15 vs. 1/15, Fisher exact $p = 0.0017$), whereas for dictionary dispatch the order reverses (4/15 vs. 12/15, $p = 0.0092$).

File Placement. Holding local support fixed, changing which files contain supporting examples substantially changes adoption in three of four tested pair–agent settings (Figure 6). In the exploratory ID-serialization experiment with Sonnet, a single supporting file yields 1/15–10/15 adoptions depending on placement ($p = 9.6 \times 10^{-4}$). Two of three replications confirm the effect: with `gpt-6-sol`, relocating two ID-serialization examples changes adoption from 0/15 to 15/15 (Holm-adjusted $p \leq 6 \times 10^{-5}$); with Sonnet, relocating one field-casing example changes adoption from 0/15 to 11/15 ($p \leq 6 \times 10^{-5}$). The Opus ID-serialization replication shows no detectable placement effect ($p = 0.148$).

Placement can even outweigh example count. For field casing with Sonnet, one example in `tasks_list` yields 11/15 adoptions, whereas two examples in the other local files yield none. Where placement matters, the most influential configurations include `tasks_list`, although agents differ in how much supporting evidence they require. These results show that example frequency alone cannot explain the influence of repository precedent.

Table 3: RQ1: default implementation in the zero-precedent repository, and the number of runs (of 15) in which each agent chooses it. "Label" gives the predefined role of the default in each pair (B = baseline, A = alternative). S = Sonnet-4-6, O = Opus-4-8, G5 = gpt-5.6-sol, G6 = gpt-6-sol.

Pair	Default implementation	Label	S	O	G5	G6
ID serialization	integer: `"id": 1`	B	15	15	15	15
Field casing	`snake_case`: `project_id`	B	15	15	15	15
Envelope	flat: `{...}`	B	15	15	15	15
Boolean naming	no prefix: `completed`	B	15	15	15	15
Error handling	direct: `return ..., 404`	B	15	15	15	15
Dispatch	lookup: `LABELS[p]`	A	14	15	15	15

	One file (1/3)				*Two files (2/3)*			
	C	U	**L**	p	C+U	**C+L**	**U+L**	p
ID / Sonnet-4-6 (expl.)	1	3	10	9.6×10^{-4}	–	–	–	0.11
ID / `gpt-6-sol`	0	0	0	1	0	8	15	$\leq 6 \times 10^{-5}$
ID / Opus-4-8	0	0	0	1	14	11	15	0.148
Casing / Sonnet-4-6	0	0	11	$\leq 6 \times 10^{-5}$	0	11	9	3.3×10^{-4}

Figure 6: File-placement effects at fixed local support (15 trials per cell). C, U, L denote `tasks_create`, `tasks_update`, `tasks_list`; bold headers indicate placements containing L. Rows show one exploration and three replications; the p column gives the placement test within each support level (χ^2 for exploration, Holm-adjusted within-block randomization for replications). "–" denotes omitted results (11–15/15).

Answer to RQ2. Local precedent strongly influences implementation choices: adoption rises from 0% without local support to 96% with full local support, with no decrease across support levels in any of the 24 pair–agent settings. However, at a fixed support level, changing file placement can shift adoption from 0% to 100%, with detectable effects in three of four tested settings.

Implication: Example frequency alone is insufficient to characterize the influence of repository precedent. Where examples appear can matter as much as, or even more than, how many appear.

4.4 RQ3: Influence of Sibling-Resource Precedent

Setup. We use the sibling-resource conditions of Stage 2, in which one to three sibling-resource files use the alternative while all local files stay at the baseline (1,080 trials), and compare them with the local-support and agreeing conditions of RQ2.

Conflicting Sibling Precedent. Sibling-resource precedent rarely overrides local code. Across 1,079 classifiable sibling-only trials, agents adopt the sibling-supported alternative 13 times (1.2%): 6/269 for Sonnet, 5/270 for `gpt-6-sol`, 2/270 for `gpt-5.6-sol`, and 0/270 for Opus. In four pairs (ID serialization, field casing, response envelope, and Boolean naming), each agent records 0/180 adoptions. All 13 adoptions occur at full sibling support, for centrally handled errors (2/15 each for Sonnet and both GPT agents) and dictionary dispatch (4/15 for Sonnet, 3/15 for `gpt-6-sol`). The location of the examples, not their number, decides the outcome: three supporting sibling-resource files yield 13/360 adoptions, whereas three supporting local files yield 347/360. Agents do see the sibling code. In all 45 Sonnet ID-serialization trials with sibling support, the `projects_*` files appeared in context visible to the orchestrating agent, and 22 transcripts explicitly discuss the conflicting ID format; the agents nevertheless follow the local files.

Agreeing Sibling Precedent. When sibling-resource code agrees with local code, it reinforces the local pattern. For response envelope, Sonnet adopts the alternative in 8/15 runs when only the local files use it and in 15/15 runs when both groups do (Fisher exact $p = 0.0063$); Opus rises from 9/15 to 15/15. In the other pairs, local support alone already yields full or near-full adoption.

Answer to RQ3. When sibling-resource precedent conflicts with local code, agents adopt the sibling pattern in only 13/1,079 trials (1.2%). With three supporting files, adoption reaches 347/360 for local precedents but only 13/360 for sibling-resource precedents. However, sibling-resource precedent can reinforce matching local patterns, increasing response-envelope adoption from 8/15 to 15/15 in one tested setting.

Implication: Repository conventions do not necessarily propagate across resource boundaries through examples alone. Agents predominantly follow the implementation patterns of the resource they extend.

4.5 RQ4: Redirecting Choices with Explicit Guidance

Setup. We test ID serialization and response envelope with Sonnet-4-6 and `gpt-6-sol`, with all six endpoints opposite the target. The content experiment runs 15 randomized blocks of C0–C3 per setting (240 trials); the delivery experiment compares no rule, a task-description rule, and a native-file

	Guidance content				*Rule delivery*		
	C0 none	**C1** rule	**C2** example	**C3** both	**none**	**task**	**file**
ID / Sonnet-4-6	0	15	0	15	0	15	15
ID / `gpt-6-sol`	0	14	15	15	0	15	15
Envelope / Sonnet-4-6	0	15	8	15	0	15	15
Envelope / `gpt-6-sol`	0	15	15	15	0	15	15
Pooled (of 60)	**0**	**59**	**38**	**60**	**0**	**60**	**60**

Figure 7: RQ4 joint success (passing functional tests and adopting the target) out of 15 trials per cell, with all six existing endpoints favoring the opposite implementation. Left (240 trials): C0 = no guidance, C1 = written rule, C2 = code example, C3 = both. Right (180 trials): no rule, task-description rule, or harness-loaded native-file rule (`CLAUDE.md`/`AGENTS.md`).

rule (`CLAUDE.md`/`AGENTS.md`) (180 trials). Both use the joint outcome and exact block-conditional randomization tests with Holm correction.

Rules vs. Examples. Without guidance (C0), no run adopts the target in any setting (0/60; Figure 7). A written rule that states the target and its priority over existing code (C1) yields 59 of 60 joint successes, 14–15/15 in every setting (Holm-adjusted $p \leq 0.00159$ vs. C0). The rule combined with an example (C3) yields 15/15 in every setting (risk difference 1.00; Holm-adjusted $p = 2.44 \times 10^{-4}$ vs. C0), and adding the example yields no gain over the rule alone (C3 vs. C1, $p_{\text{Holm}} = 1$). An example alone (C2) is unreliable: it yields 38 of 60 joint successes, 15/15 for `gpt-6-sol` on both pairs but only 8/15 for Sonnet on response envelope and 0/15 on string IDs, the pair on which Sonnet followed local examples most readily in RQ2. An example thus acts as one more precedent that competes with the repository, whereas a stated priority resolves the conflict. Guidance does not cost functional success: 238 of the 240 runs pass the hidden tests.

Rule Delivery. A rule delivered through the native instruction file is followed as reliably as a rule in the task description (Figure 7, right panel). Without a rule, no run succeeds (0/60); with the rule in the task description, all 60 runs do, and with the rule in the native instruction file, again all 60, 15/15 in every setting. The primary native-file versus no-rule contrasts each have a risk difference of 1.00 (Holm-adjusted $p = 0.000244$). The two channels do not differ detectably, but with every cell at the ceiling the data cannot show equivalence: the pooled 60/60 versus 60/60 comparison is compatible with a difference of up to about 6 percentage points in either direction (Newcombe interval). The task-rule condition also reproduces the earlier written-rule result (C1: 14–15/15 in each setting).

Answer to RQ4. Despite unanimous conflicting repository precedent, written priority rules achieve both functional success and target adoption in 59/60 trials, compared with 38/60 for code examples alone and 0/60 without guidance. In a separate experiment, rules in the task description and harness-loaded native instruction files each achieve 60/60. In four open-source repositories, a native-file rule raises joint success from 0/120 to 117/120.

Implication: Explicitly stating the priority of new task intent is more reliable than providing examples alone. Because both implementation choices satisfy the same functional requirements, verifying that intent requires measuring implementation choice alongside functional success.

4.6 Generalization to Open-Source Repositories

Setup. We evaluate four REST API repositories spanning Python, Go, and TypeScript: FastAPI Real-World [20], FastAPI Clean Example [11], Go REST API [23], and ExpressJS API [14] (Section 3.6). We apply the design of Section III-F to these four repositories. For error handling, all four agents run 15 trials for each of the four local/sibling assignments (PP, PD, DP, DD) in each repository (960 trials). For response envelope, Sonnet-4-6 and `gpt-6-sol` run the same design (420 trials; `fastapi-clean` has no sibling resource, so only FN and DN). For explicit guidance, a native-file rule is compared with a no-guidance control in the same repositories and agents (240 trials). For statistical analysis, we use Wilson intervals and stratified Fisher and Mantel–Haenszel tests for error handling. Response-envelope comparisons use Holm-adjusted tests for local and sibling effects, with block-bootstrap intervals (99,999 draws). Explicit-guidance comparisons use paired exact tests with Holm correction across eight repository–agent settings.

Error Handling. Agents follow the local pattern without exception. Of the 960 trials, 948 pass the hidden tests (98.8%; Table 4), and all 945 classifiable successes follow the local pattern (Wilson 95% CI 99.6–100%). In every repository–agent stratum, PP implementations propagate errors and DD implementations construct responses directly (16 Fisher tests, all $p < 0.001$; Mantel–Haenszel $\chi^2 \geq 106.2$, $p < 10^{-9}$ per agent), whereas changing only the sibling-resource pattern has no effect (all 16 Fisher tests, $p = 1.0$).

Response Envelope. Local precedent again dominates, though less uniformly (Table 6). Of the 420 trials, 417 pass the hidden tests. With flat local handlers, no implementation adopts the envelope (0/207); with enveloped local handlers, 148 of 210 do (70%). `gpt-6-sol` adopts the envelope in 11–15 of 15 runs in every repository, whereas Sonnet adopts it in 15/15 runs in `expressjs-api` and `fastapi-clean` but in none of 30 runs in `go-rest`. The local contrast is significant in 7 of 8 strata (differences 0.30–1.00), and none of the 12 sibling contrasts is significant. As in the controlled repository, response envelope is the choice that local precedent determines least completely.

Explicit Guidance. A rule in the native instruction file redirects agents in real repositories. Without guidance, no run adopts the envelope (0/120), consistent with the 0/207 observed under flat local handlers in the response-envelope study above. With the rule, 117 of 120 runs both pass the hidden tests and adopt the envelope (97.5%; Table 5), an improvement of 80–100 percentage points in every repository–agent stratum (all eight Holm-adjusted $p = 4.9 \times 10^{-4}$, the floor of the exact test with 15 blocks). `gpt-6-sol` follows the rule in 60 of 60 runs and Sonnet in 57 of 60. All three failures occur with Sonnet in `fastapi-realworld`: two runs keep a flat response and one adopts the envelope but fails a functional test, so raw target-shape adoption is 118/120 and functional success is 119/120.

Summary of the open-source study. Across four open-source repositories spanning three languages, agents follow local precedent in all 945 functionally successful, classifiable error-handling implementations and 355/417 (85%) functionally successful response-envelope implementations. Sibling-resource patterns have no detectable effect on either choice. An explicit rule in the harness's native instruction file redirects implementation choices despite conflicting local precedent, achieving 117/120 joint successes compared with 0/120 without guidance.

Implication: The strong influence of local precedent and the effectiveness of explicit priority rules extend to the studied open-source repositories. However, adherence to local precedent varies across implementation choices, repositories, and agents.

5 Threats to Validity

Construct Validity. Our local–sibling comparison jointly varies task relatedness and repository location, so it does not distinguish their individual effects. The experiments control which precedents are available, not which ones agents attend to. The zero-precedent condition measures task-specific choices without relevant endpoint examples rather than context-free model preferences. Native instruction-file experiments evaluate rules loaded by the harness, not autonomous discovery. Our classifiers capture selected implementation patterns, and the outcomes concern implementation choice, not maintainability or overall code quality.

Internal Validity. In the support-level experiments, example count and file assignment vary together. The separate placement analysis reveals sensitivity to file identity but does not explain the influence of

Table 4: Open-source error-handling study. Task success uses all 960 trials; local adoption uses 945 functionally successful, classifiable implementations. Agent and repository rows summarize the same trials.

Group	Task success	Local adoption	Unclass.
Sonnet-4-6	228/240	228/228	0
Opus-4-8	240/240	240/240	0
gpt-5.6-sol	240/240	240/240	0
gpt-6-sol	240/240	237/237	3
FastAPI RealWorld	240/240	240/240	0
FastAPI Clean	236/240	233/233	3
Go REST	233/240	233/233	0
ExpressJS API	239/240	239/239	0
All	**948/960**	**945/945**	**3**

Table 5: Open-source explicit guidance: runs (of 15) that both pass the hidden tests and wrap the payload under `data`, when all local handlers return flat payloads. No rule: contemporaneous control; native rule: the rule in `CLAUDE.md` (Sonnet-4-6) or `AGENTS.md` (`gpt-6-sol`).

	Sonnet-4-6		`gpt-6-sol`	
Repository	No rule	Native rule	No rule	Native rule
FastAPI RealWorld	0/15	12/15	0/15	15/15
Go REST	0/15	15/15	0/15	15/15
ExpressJS API	0/15	15/15	0/15	15/15
FastAPI Clean	0/15	15/15	0/15	15/15
Total	0/60	57/60	0/60	60/60

Table 6: Open-source response-envelope study. Entries give functionally successful data-envelope (D) implementations out of 15 trials. Columns specify local and sibling-resource patterns (F = flat, D = envelope); FN/DN indicate no sibling resource. Δ is the D-adoption difference between D-local and F-local assignments, with Holm-adjusted p-values across eight repository–agent strata. S = Sonnet-4-6; G6 = gpt-6-sol.

		Local flat		Local envelope			
Repository	Agent	FF/FN	FD	DF/DN	DD	Δ	p
RealWorld	S	0/15	0/15	3/15	6/15	0.30	0.0078
RealWorld	G6	0/15	0/15	12/15	11/15	0.77	1.2×10^{-6}
Go REST	S	0/15	0/15	0/15	0/15	0.00	1
Go REST	G6	0/15	0/15	15/15	15/15	1.00	1.5×10^{-8}
ExpressJS	S	0/15	0/15	15/15	15/15	1.00	1.5×10^{-8}
ExpressJS	G6	0/15	0/15	15/15	14/15	0.97	2.2×10^{-8}
Clean	S	0/15	—	15/15	—	1.00	2.4×10^{-4}
Clean	G6	0/15	—	12/15	—	0.80	0.0015

`tasks_list`. The zero-precedent and precedent conditions were collected under different execution settings. Technical interruptions required retries and disrupted scheduling in RQ4. An initial misconfigured Opus file-placement run was excluded and rerun; open-source outcome audits also led to revised classifications and recapture of incomplete patches. These changes and the trial accounting are documented in the artifact. Finally, differences between agents cannot be attributed to models alone, as each agent combines a model with its harness.

External Validity. Fifteen repetitions per cell characterize variability under fixed conditions, not across tasks or repositories. The controlled experiments use one Flask API task family. The open-source study extends the local–sibling comparison to four repositories in three languages, but covers only two implementation choices, with explicit guidance evaluated for one. Support-level and file-placement effects remain untested beyond the controlled repository. The placement and guidance experiments cover selected settings, and the study evaluates agents from two providers.

6 Related Work

Coding Agents and Software Evolution. Code-generation benchmarks such as HumanEval and MBPP evaluate the functional correctness of standalone programs [5, 3]. SWE-bench moved evaluation to resolving real issues in existing repositories [13], and agents such as SWE-agent [35], AutoCodeRover [37], OpenHands [29], and Agentless [32] navigate, edit, and test codebases to resolve them. Recent work extends evaluation from isolated issues to the continuous evolution of a repository: SWE-Milestone [6] and SWE-EVO [15] evaluate agents on multi-step, long-horizon evolution tasks, Needle in the Repo [40] diagnoses maintainability failures in agent edits that pass functional tests, and Pinna et al. [22] compare agents through pull-request acceptance.

These studies ask whether agent changes succeed and remain maintainable. We ask which of several functionally acceptable implementations an agent selects, and what in the repository and task determines that choice.

Repository Context and Conventions. Software is repetitive and locally predictable [10, 26], and project-specific conventions can be learned from existing code [1]; LLM-generated code nonetheless deviates from human coding style [30]. Repository-level techniques retrieve or plan over cross-file context to improve generation and repository-wide edits [25, 36, 7, 31, 16, 4], and repository instruction files such as `AGENTS.md` change agent behavior without generally improving task success [9].

These approaches treat context as a means to correctness. We hold correctness fixed and vary the available context, isolating how local precedent, sibling-resource precedent, and explicit instructions each govern the selected implementation.

Preferences, Conflicting Guidance, and Design Rationale. LLMs show systematic preferences in code generation, such as favoring Python and widely used libraries [27], and their in-context behavior depends on the choice, order, and position of demonstrations [39, 19, 8, 18]. When context contradicts what a model has learned, LLMs may follow either source [33, 34], and instruction-hierarchy work trains models to rank instructions by their origin [28]. We study the corresponding conflict in software evolution, where an agent's default, existing code, and new task intent compete, and measure which prevails. In software engineering, design decisions are first-class artifacts whose rationale is often lost [12, 17, 38, 24]. Our

results show that agents act on recorded decisions without their rationale, and follow explicit priorities once these are stated.

7 Conclusion

As coding agents increasingly participate in software evolution, functional correctness alone cannot explain which implementation decisions they preserve or change. This study establishes implementation choice as a distinct dimension of agentic software engineering and yields three takeaways: (1) repository precedent overrides agents' shared defaults; (2) where precedent appears can matter more than how often, as local code dominates sibling resources and file placement alone can decide adoption; and (3) new intent must be stated, not only demonstrated, as written priority rules redirect agents more reliably than code examples.

Our findings call for evaluating not only whether agents produce correct changes, but how they reconcile past decisions with evolving intent. This requires design-decision traceability that preserves the rationale and priorities guiding future evolution.

References

[1] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. 2014. Learning Natural Coding Conventions. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. 281–293. doi:10.1145/2635868.2635883

[2] Anthropic. 2026. Claude Code: Overview. Official documentation. Accessed: 2026-10-16. https://code.claude.com/docs/en/overview

[3] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. 2021. Program Synthesis with Large Language Models. arXiv:2108.07732. arXiv:2108.07732

[4] Ramakrishna Bairi et al. 2024. CodePlan: Repository-Level Coding Using LLMs and Planning. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 675–698. doi:10.1145/3643757

[5] Mark Chen et al. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374. arXiv:2107.03374

[6] Gangda Deng, Zhaoling Chen, Zhongming Yu, Haoyang Fan, Yuhong Liu, Yuxin Yang, Dhruv Parikh, Rajgopal Kannan, Le Cong, Mengdi Wang, Qian Zhang, Viktor Prasanna, Xiangru Tang, and Xingyao Wang. 2026. SWE-Milestone: Evaluating AI Agents on Continuous Software Evolution. In *Proceedings of the 43rd International Conference on Machine Learning (ICML) (Proceedings of Machine Learning Research, Vol. 306)*. PMLR, 23853–23890. https://proceedings.mlr.press/v306/deng26d.html

[7] Yangruibo Ding, Zijian Wang, Wasi Uddin Ahmad, Hantian Ding, Ming Tan, Nihal Jain, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. 2023. CrossCodeEval: A Diverse and Multilingual Benchmark for Cross-File Code Completion. In *Advances in Neural Information Processing Systems (NeurIPS)*, Vol. 36. 46701–46723. doi:10.52202/075280-2023

[8] Yu Fei, Yifan Hou, Zeming Chen, and Antoine Bosselut. 2023. Mitigating Label Biases for In-context Learning. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (ACL), Volume 1: Long Papers*. 14014–14031. doi:10.18653/v1/2023.acl-long.783

[9] Thibaud Gloaguen, Niels Mündler, Mark Müller, Veselin Raychev, and Martin Vechev. 2026. Evaluating AGENTS.md: Are Repository-Level Context Files Helpful for Coding Agents? arXiv:2602.11988. arXiv:2602.11988

[10] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the Naturalness of Software. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*. 837–847. doi:10.1109/ICSE.2012.6227135

[11] ivan-borovets and contributors. 2026. fastapi-clean-example. GitHub repository. Accessed: 2026-10-16. https://github.com/ivan-borovets/fastapi-clean-example/tree/5d68d52c9635c7da74d4d45ffb58493f267531aa

[12] Anton Jansen and Jan Bosch. 2005. Software Architecture as a Set of Architectural Design Decisions. In *Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture (WICSA)*. 109–120. doi:10.1109/WICSA.2005.61

[13] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-World GitHub Issues?. In *The Twelfth International Conference on Learning Representations (ICLR)*. https://openreview.net/forum?id=VTF8yNQM66

[14] larswaechter and contributors. 2026. expressjs-api. GitHub repository. Accessed: 2026-10-16. https://github.com/larswaechter/expressjs-api/tree/309553e6c599c7b5f150ebef990ff3f543eee92b

[15] Tue Le, Minh V. T. Thai, Dung Nguyen Manh, Huy Phan Nhat, and Nghi D. Q. Bui. 2026. SWE-EVO: Benchmarking Coding Agents in Long-Horizon Software Evolution Scenarios. arXiv:2512.18470. arXiv:2512.18470 [cs.SE]

[16] Jia Li, Hao Zhu, Huanyu Liu, Xianjie Shi, He Zong, Yihong Dong, Kechi Zhang, Siyuan Jiang, Zhi Jin, and Ge Li. 2025. Aligning LLMs to Fully Utilize the Cross-file Context in Repository-level Code Completion. In *Proceedings of the 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1477–1489. doi:10.1109/ASE63991.2025.00125

[17] Jenny T. Liang, Maryam Arab, Minhyuk Ko, Amy J. Ko, and Thomas D. LaToza. 2023. A Qualitative Study on the Implementation Design Decisions of Developers. In *Proceedings of the 45th International Conference on Software Engineering (ICSE)*. 435–447. doi:10.1109/ICSE48619.2023.00047

[18] Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2024. Lost in the Middle: How Language Models Use Long Contexts. *Transactions of the Association for Computational Linguistics* 12 (2024), 157–173. doi:10.1162/tacl_a_00638

[19] Yao Lu, Max Bartolo, Alastair Moore, Sebastian Riedel, and Pontus Stenetorp. 2022. Fantastically Ordered Prompts and Where to Find Them: Overcoming Few-Shot Prompt Order Sensitivity. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (ACL), Volume 1: Long Papers*. 8086–8098. doi:10.18653/v1/2022.acl-long.556

[20] nsidnev and contributors. 2026. fastapi-realworld-example-app. GitHub repository. Accessed: 2026-10-16. https://github.com/nsidnev/fastapi-realworld-example-app/tree/029eb7781c60d5f563ee8990a0cbfb79b244538c

[21] OpenAI. 2026. Codex CLI. Official documentation. Accessed: 2026-10-16. https://learn.chatgpt.com/docs/codex/cli

[22] Giovanni Pinna, Jingzhi Gong, David Williams, and Federica Sarro. 2026. Comparing AI Coding Agents: A Task-Stratified Analysis of Pull Request Acceptance. In *Proceedings of the 23rd International Conference on Mining Software Repositories (MSR)*. 792–796. doi:10.1145/3793302.3793576

[23] qiangxue and contributors. 2026. go-rest-api. GitHub repository. Accessed: 2026-10-16. https://github.com/qiangxue/go-rest-api/tree/c3101332148b4aa45f637605e394cb0d033dae97

[24] Pankajeshwara Nand Sharma, Bastin Tony Roy Savarimuthu, and Nigel Stanger. 2021. Extracting Rationale for Open Source Software Development Decisions: A Study of Python Email Archives. In *Proceedings of the 43rd International Conference on Software Engineering (ICSE)*. 1008–1019. doi:10.1109/ICSE43902.2021.00095

[25] Disha Shrivastava, Hugo Larochelle, and Daniel Tarlow. 2023. Repository-Level Prompt Generation for Large Language Models of Code. In *Proceedings of the 40th International Conference on Machine Learning (ICML) (Proceedings of Machine Learning Research, Vol. 202)*. PMLR, 31693–31715.

[26] Zhaopeng Tu, Zhendong Su, and Premkumar T. Devanbu. 2014. On the Localness of Software. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. 269–280. doi:10.1145/2635868.2635875

[27] Lukas Twist, Mark Harman, Don Syme, Joost Noppen, Helen Yannakoudakis, Detlef Nauck, and Jie M. Zhang. 2026. A Study of LLMs' Preferences for Libraries and Programming Languages. In *Findings of the Association for Computational Linguistics: ACL 2026*. 331–351. doi:10.18653/v1/2026.findings-acl.15

[28] Eric Wallace, Kai Xiao, Reimar Leike, Lilian Weng, Johannes Heidecke, and Alex Beutel. 2024. The Instruction Hierarchy: Training LLMs to Prioritize Privileged Instructions. arXiv:2404.13208.

[29] Xingyao Wang et al. 2025. OpenHands: An Open Platform for AI Software Developers as Generalist Agents. In *The Thirteenth International Conference on Learning Representations (ICLR)*.

[30] Yanlin Wang, Tianyue Jiang, Mingwei Liu, Jiachi Chen, Mingzhi Mao, Xilin Liu, Yuchi Ma, and Zibin Zheng. 2025. Beyond Functional Correctness: Investigating Coding Style Inconsistencies in Large Language Models. *Proceedings of the ACM on Software Engineering* 2, FSE (2025), 690–712. doi:10.1145/3715749

[31] Di Wu, Wasi Uddin Ahmad, Dejiao Zhang, Murali Krishna Ramanathan, and Xiaofei Ma. 2024. Repoformer: Selective Retrieval for Repository-Level Code Completion. In *Proceedings of the 41st International Conference on Machine Learning (ICML) (Proceedings of Machine Learning Research, Vol. 235)*. PMLR, 53270–53290.

[32] Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2025. Demystifying LLM-Based Software Engineering Agents. *Proceedings of the ACM on Software Engineering* 2, FSE (2025), 801–824. doi:10.1145/3715754

[33] Jian Xie, Kai Zhang, Jiangjie Chen, Renze Lou, and Yu Su. 2024. Adaptive Chameleon or Stubborn Sloth: Revealing the Behavior of Large Language Models in Knowledge Conflicts. In *The Twelfth International Conference on Learning Representations (ICLR)*.

[34] Rongwu Xu et al. 2024. Knowledge Conflicts for LLMs: A Survey. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 8541–8565. doi:10.18653/v1/2024.emnlp-main.486

[35] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik R. Narasimhan, and Ofir Press. 2024. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. In *Advances in Neural Information Processing Systems (NeurIPS)*, Vol. 37. 50528–50652. doi:10.52202/079017-1601

[36] Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. 2023. RepoCoder: Repository-Level Code Completion Through Iterative Retrieval and Generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 2471–2484. doi:10.18653/v1/2023.emnlp-main.151

[37] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. AutoCodeRover: Autonomous Program Improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 1592–1604. doi:10.1145/3650212.3680384

[38] Jiuang Zhao, Zitian Yang, Li Zhang, Xiaoli Lian, Donghao Yang, and Xin Tan. 2024. DRMiner: Extracting Latent Design Rationale from Jira Issue Logs. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 468–480. doi:10.1145/3691620.3695019

[39] Zihao Zhao, Eric Wallace, Shi Feng, Dan Klein, and Sameer Singh. 2021. Calibrate Before Use: Improving Few-Shot Performance of Language Models. In *Proceedings of the 38th International Conference on Machine Learning (ICML) (Proceedings of Machine Learning Research, Vol. 139)*. PMLR, 12697–12706.

[40] Haichao Zhu, Qian Zhang, Jiyuan Wang, Zhaorui Yang, and Yuxin Qiu. 2026. Needle in the Repo: Diagnosing Maintainability Failures in AI-Generated Repository Edits. In *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 892–903. doi:10.1145/3832783.3834401