ContextCov: Bridging the Gap Between Developer Intent and Autonomous Agent Execution

Reshabh K Sharma
University of Washington
USA
reshabh@cs.washington.edu

Abstract

As Large Language Model (LLM) agents increasingly execute complex, autonomous software engineering tasks, developers rely on natural language instruction files such as `AGENTS.md` to express project-specific coding conventions, tooling restrictions, and architectural boundaries. However, because these instructions remain passive text, agents frequently exhibit what we term *Context Drift*, which includes deviations from repository constraints caused by context window saturation, hallucination, or conflicts with legacy code. In autonomous settings without real-time human supervision, such deviations rapidly compound into technical debt.

To ground autonomous agents in repository constraints, we introduce ContextCov, a framework that transforms passive natural language instructions into executable guardrails. Unlike prompt-only or reflection-only compliance approaches, ContextCov compiles documented constraints into three complementary checks: static AST queries for code patterns, runtime shell shims that intercept prohibited commands, and architectural validators that enforce structural rules. Acting as an automated, continuous reviewer, ContextCov intercepts agent actions and returns immediate, reproducible violation traces, enabling self-correction before non-compliant changes are finalized.

We evaluate ContextCov on SWE-bench Lite (300 tasks, 12 repositories). ContextCov extracts constraints with 82.81% precision, achieves 88.3% oracle compliance (vs. 67.0% vanilla, 50.3% self-reflection), and improves functional resolution to 57.3% (vs. 53.0% and 52.3%; McNemar $p < 0.05$). Executable feedback yields 3.4× better cost-efficiency than LLM-based reflection.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
Conference'17, Washington, DC, USA
© 2026 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

ACM Reference Format:
Reshabh K Sharma. 2026. ContextCov: Bridging the Gap Between Developer Intent and Autonomous Agent Execution. In . ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Motivating Scenarios

This section presents three motivating scenarios derived from Visual Studio Code's `copilot-instructions.md` [5] (Figure 1). We select Microsoft VS Code because it is one of the most widely used and starred open-source software repositories and provides a production-quality agent instruction file [12]. We use these scenarios to show how ContextCov maps natural-language instructions to executable enforcement points across process, source, and architecture levels.

Agents violate constraints for at least two distinct reasons: (1) *instruction omission*, where relevant constraints are not attended to during generation, and (2) *instruction override*, where the agent prioritizes locally common code patterns or short-term task completion over documented project guidance.

The scenarios below are exploratory motivating examples, not controlled benchmark reproductions. To construct them, we manually inspected the instruction file and ran interactive probing conversations with an LLM agent about how it would handle representative tasks; in multiple probes, the agent did not reliably acknowledge or apply relevant instruction-file constraints. ContextCov addresses these failure modes by transforming passive instructions into executable checks that provide immediate, actionable feedback to the agent.

1.1 Process Violation

Lines 1–6 of Figure 1 specify constraints for the TypeScript build process. Consider an agent tasked with fixing editor tests. It may attempt to run tests directly (e.g., `npm test`) before verifying compilation. However, VS Code uses an incremental watch-based system. Running tests prematurely can create cascading, noisy failures. If the agent runs `npm run compile`, it bypasses the intended workflow.

ContextCov addresses this by parsing process constraints into shell shims. When the agent invokes a forbidden command, it is blocked and receives immediate deterministic feedback via stderr:

```
1  ## Validating TypeScript changes
2  MANDATORY: Always check the `VS Code - Build` watch task
3  output for compilation errors before running ANY script.
4
5  - NEVER run tests if there are compilation errors
6  - NEVER use `npm run compile` to compile TypeScript
       files
7
8  ## Coding Guidelines
9  ### Style
10 - Use arrow functions `=>` over anonymous function
       expressions
11 - Only surround arrow function parameters when necessary
       .
12   For example, `(x) => x + x` is wrong but these are
         correct:
13     x => x + x
14     (x, y) => x + y
15 - Prefer `async/await` over `Promise` and `then` calls
16
17 ### Core Architecture (`src/` folder)
18 - `src/vs/base/` - Foundation utilities
19 - `src/vs/platform/` - Platform services and DI
20 - `src/vs/editor/` - Text editor implementation
21 - `src/vs/workbench/` - Main application workbench
22   - `workbench/browser/` - Core workbench UI components
23   - `workbench/services/` - Service implementations
24   - `workbench/contrib/` - Feature contributions
25   - `workbench/api/` - Extension host and VS Code API
26 - **Layered architecture** - from `base` to `workbench`
27
28 ### Code Quality
29 - You MUST NOT use storage keys of another component
       only
30   to make changes to that component. You MUST come up
         with
31   proper API to change another component.
```

Figure 1. Excerpt from VS Code's `copilot-instructions.md`, a production Agent Instruction file used to guide AI coding assistants. Lines 1–6 define process constraints for the TypeScript build workflow, lines 8–22 specify source-level coding conventions, and lines 24–38 establish architectural boundaries and design principles.

```
[ContextCov] BLOCKED: Process constraint violated
  Rule: "NEVER use npm run compile"
  Action: Check VS Code - Build watch task output instead.
```

The agent can then self-correct.

1.2 Source Violation

Lines 8–22 of Figure 1 specify coding conventions. Consider an agent writing natural-looking TypeScript: `watcher.onDidChange((event) => { this.handleChange(event); });`. While functionally correct, this violates VS Code's style guideline because the single parameter `event` should not be parenthesized.

ContextCov generates a Tree-sitter query to detect this exact AST pattern. The static linter reports the violation with precise coordinates, enabling the agent to self-correct before commit:

```
[ContextCov] Source violation at watcher.ts:127
  Rule: "Only surround arrow function parameters when
       necessary"
  Suggestion: Remove parentheses: event => {...}
```

1.3 Architectural Violation

Lines 24–38 of Figure 1 define VS Code's architectural constraints. We distinguish two subtypes: deterministic constraints, which can be verified programmatically, and semantic constraints, which require model-based judgment over intent and responsibility boundaries.

1.3.1 Deterministic Architectural Constraints. Lines 24–33 specify the layered architecture and directory structure. Consider an agent tasked with "Add a utility function for file path normalization" that must choose where to place new code. The agent might place it in `src/vs/workbench/common/` because that is where the immediate caller resides. However, VS Code's layered architecture requires such utilities in `src/vs/base/`, the foundation layer. Similarly, an agent might create a file directly under `src/vs/workbench/` rather than in allowed subdirectories such as `browser`, `services`, `contrib`, or `api`. These violations occur when the agent optimizes for local task completion instead of global architectural constraints.

ContextCov's architectural validator constructs a dependency graph and enforces the layered structure. If an agent attempts to create a file in an invalid location, it receives immediate feedback:

```
[ContextCov] Architectural violation: src/vs/workbench/
     myNewUtility.ts
  Rule: "workbench/ only allows subdirectories: browser,
       services, contrib, api"
  Suggestion: Move to appropriate subdirectory or to base/
```

More broadly, deterministic architectural violations include layer violations (dependencies crossing architectural boundaries), cyclic dependencies, and module placement errors. These can be detected using file-path validation and import-graph traversal.

1.3.2 Semantic Architectural Constraints. Lines 35–38 specify a design principle where components must not directly access another component's storage keys but should instead define proper APIs for cross-component changes. Such constraints cannot be verified through file paths or import graphs alone because they require understanding the code's intent. Consider an agent tasked with "Persist user preferences across sessions" that discovers another component already stores related settings. Rather than defining a proper API, the agent directly reads and writes to that component's storage keys. This violates encapsulation and couples the components in ways that are difficult to detect in code review.

ContextCov handles semantic constraints using an LLM-as-judge approach, where a model evaluates whether code

changes respect the guideline. These checks produce warnings rather than hard blocks to reduce unsafe false positives in uncertain cases.

```
[ContextCov] WARNING: Possible architectural violation
  Rule: "MUST NOT use storage keys of another component"
  File: src/vs/workbench/contrib/settings/storage.ts
  Concern: Appears to access 'terminal.integrated.fontSize'
           which belongs to the terminal component.
  Suggestion: Define an API in the terminal component
        instead.
```

2 ContextCov Design

ContextCov consists of two main phases. *Check generation* extracts and synthesizes constraints into executable checks, and *runtime enforcement* executes these checks during agent operation. Given an Agent Instruction file, ContextCov generates code-based checks that can be incrementally updated as the instruction file evolves. At runtime, source and architectural checks can be enforced over an entire repository, a git diff, or only unstaged changes, while process checks are enforced during agent execution. ContextCov is implemented in Python. Constraint refinement and intent routing are LLM-assisted steps using OpenAI's `gpt-5.2-chat`. Final generation for process checks, source checks, deterministic architectural checks, and semantic review rubrics is performed by Claude Code with Opus 4.5 because this setup produces more complete outputs than direct single-shot generation even when file structure context is provided.

2.1 System Overview

Figure 2 presents the high-level architecture. During extraction, Agent Instruction documents are parsed into a Markdown Abstract Syntax Tree, and a path-aware algorithm traverses the tree extracting constraints as slices with their hierarchical context preserved. An LLM refines each constraint into a standalone, unambiguous statement. During synthesis, each refined constraint is classified by an intent router into one of four enforcement domains and passed to specialized code generators that produce executable Python checks. These checks are stored in a JSON file.

The **Enforcement** phase executes these checks through specialized modules. A Process Interceptor uses PATH shims to intercept shell commands before execution. A Universal Static Linter uses Tree-sitter to analyze source code across multiple languages. An Architectural Validator constructs and queries dependency graphs to detect structural violations. Finally, an LLM-as-judge evaluates semantic architectural constraints that cannot be verified through static analysis alone.

This architecture provides several benefits. When Agent Instructions change, only affected constraints are re-extracted and re-synthesized, enabling efficient incremental updates. Using separate code synthesis modules improves generation

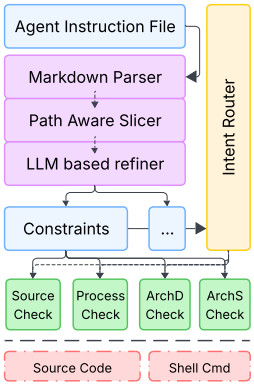

Figure 2. Architecture of ContextCov. The check generation phase comprises Extraction (parsing Agent Instructions into a Markdown AST and refining constraints via LLM) and Synthesis (routing constraints to domain-specialized code generators). The Enforcement phase executes checks through a Process Interceptor, Universal Static Linter, and Architectural Validator.

quality, as each expert generator is prompted with domain-appropriate patterns and examples. The multiple enforcement modules address the heterogeneity of agent actions. Process constraints require runtime interception, source constraints require AST analysis, deterministic architectural constraints require graph reasoning, and semantic architectural constraints require LLM-based judgment.

2.2 Constraint Extraction

The extraction phase transforms unstructured Markdown into a set of refined, standalone constraints. This is challenging because Agent Instructions are written for human readers who naturally understand context and scope from document structure.

The Challenge of Lexical Scope. Simple approaches to Agent Instruction segmentation fail to preserve *lexical scope*. Consider the following excerpt.

```
# Backend
## Testing
- Use pytest
## Database
- Use SQLAlchemy
```

The instruction "Use pytest" appears as a standalone bullet, but its meaning depends on its hierarchical context and applies to *Backend Testing*, not to the entire project or to Database operations. If we extract this constraint without its context, we might incorrectly enforce pytest usage for frontend code or database migrations.

Markdown Abstract Syntax Tree. To preserve this hierarchical context, we parse each Agent Instruction file into a tree structure. Internal nodes represent headers at various

levels (H1 through H6), leaf nodes represent content blocks (paragraphs, list items, code blocks), and edges represent containment relationships. This tree structure mirrors how humans mentally organize the document when reading it.

Path-Aware Slicing. For each leaf node representing a potential constraint, we compute its *header path*, the ordered sequence of ancestor headers from the document root to the leaf. In the example above, "Use pytest" has the header path `[Backend, Testing]`. We then use an LLM to rewrite the raw instruction into a standalone, unambiguous statement that incorporates this context, producing a refined constraint like "For backend testing, use the pytest framework for all unit and integration tests." This refinement step disambiguates scope and normalizes diverse phrasings into consistent constraint statements that can be enforced without additional context.

Stable Identifiers for Incrementality. To support incremental updates, we assign each constraint a stable identifier by hashing its header path and content. When a developer modifies one section of an Agent Instruction file, only constraints from that section need to be re-extracted and re-synthesized. Constraints from unmodified sections retain their identifiers and their previously generated checks remain valid. This is important for large instruction files where full re-extraction would be expensive.

2.3 Constraint to Check

This phase transforms refined constraints into executable Python code. Rather than using a single monolithic code generator, we route each constraint to a specialized generator based on its enforcement domain. This domain-specific approach improves code quality because each generator is optimized for its particular verification pattern.

The Intent Router. The Router classifies each constraint into one of four enforcement domains. For the **Process Domain**, the router extracts target binary names (e.g., `npm`, `yarn`), the enforcement level, and the applicable scope. For the **Source Domain**, it identifies target programming languages, the pattern type to detect, and the severity level. For the **Architectural Deterministic Domain**, it determines the constraint type (dependency direction, cycle detection, or boundary enforcement) and selects an appropriate graph algorithm. For the **Architectural Semantic Domain**, it preserves the natural language description of the design principle for later LLM evaluation.

The router's classification is performed by the LLM, prompted with examples of each domain to improve accuracy.

Specialized Code Generators. Each domain has a specialized Expert Generator that produces Python code tailored to that domain's verification needs. The **Process Expert** generates code that inspects command-line arguments and environment variables, returning a verdict that determines whether the command should proceed. The **Source Expert** generates code using Tree-sitter bindings that constructs AST queries and traverses parse trees, returning violations with precise file and line information. The **Architectural Expert** generates code using NetworkX that applies graph algorithms to dependency structures, detecting cycles, layer violations, or boundary crossings.

In our implementation, domain-specific outputs are generated through Claude Code with Opus 4.5 rather than by a single direct model call. The generation prompt provides the refined constraint, repository structure, and relevant files. The prompt also instructs an iterative self-test loop that samples representative outcomes and revises logic when evidence suggests false positives. In addition to this prompted behavior, our pipeline enforces programmatic validation for generated executable checks through syntax checks and dry-run execution before accepting the check. Semantic architectural generation produces reviewer rubrics rather than executable code, so it uses output-shape validation rather than code execution validation.

Tree-sitter grammars are bundled for Python, TypeScript, JavaScript, Go, and Rust. This coverage addresses the most common languages in repositories with Agent Instructions, though additional languages can be added by installing the corresponding Tree-sitter packages.

Check Storage. Synthesized checks are stored in a JSON file that can be reviewed and manually edited by developers before deployment, providing a layer of human oversight over the generated checks. The listing below shows an example entry.

```
{
  "constraints": {
    "a7f3b2...": {
      "domain": "PROCESS",
      "original_text": "NEVER use npm run compile",
      "refined_text": "Block npm run compile commands",
      "source_file": "copilot-instructions.md",
      "header_path": ["Validating TypeScript changes"],
      "enforcement_level": "block",
      "check_code": "def check(cmd, args, env): ..."
    }
  }
}
```

This separation of synthesis from enforcement provides several advantages. It enables incremental updates where only changed constraints trigger re-synthesis. It allows developers to inspect, modify, or disable specific checks before they take effect. And it ensures fast enforcement at runtime with no LLM calls required, since all checks are pre-compiled Python code.

2.4 Runtime Check Enforcement

The enforcement phase executes synthesized checks through four specialized modules, each designed for a particular type of constraint.

Process Interceptor. Enforcing process constraints requires intercepting shell commands before execution. We use $PATH manipulation for simplicity and portability. ContextCov creates a shim directory containing wrapper scripts for monitored binaries and prepends this directory to $PATH. Each monitored binary (e.g., npm, yarn, pip) has a corresponding wrapper script.

```
#!/bin/bash
# .contextcov/bin/npm
exec python -m contextcov.shim npm "$@"
```

When a command is executed, the shell resolves the binary via $PATH and finds our shim first. The shim loads relevant checks from the JSON file and executes each check with the command arguments and environment. If any check fails, the shim prints an error message to stderr explaining which constraint was violated and exits with code 1, preventing the command from running. If all checks pass, the shim uses `which -a` to find the real binary and executes it with the original arguments, making the interception transparent to the agent.

Universal Static Linter. ContextCov uses Tree-sitter as a parser for source code analysis [4]. Tree-sitter provides incremental parsing, graceful error recovery, and a uniform API across languages. These properties make it suitable for analyzing code in diverse repositories. The linter scans source files matching configured patterns, parses each file with the appropriate grammar based on file extension, and runs all applicable checks. Each check receives the parse tree, the raw source bytes, and the file path, allowing it to construct queries, traverse the AST, and report violations with precise locations. The following example shows a generated check that detects `Promise.then()` usage.

```
def check(tree, source_bytes, filepath):
    """Check␣for␣Promise.then()␣usage."""
    query = LANGUAGE.query("""
␣␣␣␣␣␣␣␣(call_expression
␣␣␣␣␣␣␣␣␣␣function:␣(member_expression
␣␣␣␣␣␣␣␣␣␣␣␣property:␣(property_identifier)␣@method)
␣␣␣␣␣␣␣␣␣␣(#match?␣@method␣"^then$"))␣""")
    violations = []
    for node, name in query.captures(tree.root_node):
        line = source_bytes[:node.start_byte].count(b'\n') +
            1
        violations.append(Violation(filepath, line,
            "Use␣async/await␣instead␣of␣.then()"))
    return violations
```

Architectural Validator. Deterministic architectural constraints are enforced by constructing and querying a dependency graph. The graph is built by parsing import statements using language-appropriate techniques. Python's `ast` module extracts `import` and `from ... import` statements, Tree-sitter queries extract `import` declarations and `require()` calls from JavaScript and TypeScript, and regex-based extraction handles Go import blocks. The resulting graph is stored as a NetworkX `DiGraph` where nodes represent modules and edges represent dependencies.

Architectural checks operate on this graph to detect violations such as layer crossings, cycles, or improper module placement. The following example enforces that files under `src/vs/workbench/` must reside in one of the allowed subdirectories.

```
def check(graph):
    """Enforce␣workbench/␣subdirectory␣structure."""
    allowed = {'browser', 'services', 'contrib', 'api'}
    violations = []
    for node in graph.nodes:
        if node.startswith('src/vs/workbench/'):
            parts = node.split('/')
            if len(parts) > 3:
                subdir = parts[3]
                if subdir not in allowed:
                    violations.append(ArchViolation(
                        source=node,
                        message=f"Must␣be␣in␣{allowed}"))
    return violations
```

Semantic Architectural Validator. Some architectural constraints cannot be verified through static analysis alone because they require understanding the intent behind code. For example, the constraint "Do not use another component's storage keys, define a proper API instead" cannot be checked by examining import graphs or file paths. It requires understanding what constitutes a "storage key," which code belongs to which "component," and whether an access pattern constitutes a violation.

For these semantic constraints, we use an LLM-as-judge approach [10]. When evaluating code changes, the LLM receives the constraint text, the relevant code snippet, and surrounding context such as file path and neighboring code. The LLM then judges whether the code respects the design principle, providing a verdict and explanation. Semantic checks produce `WARNING` verdicts rather than hard blocks, flagging potential violations for human review while avoiding false positives from overly strict interpretation.

2.5 Handling Ambiguity

A key design decision is how to handle ambiguous constraints. Agent Instructions are often written informally, and the same intent can be expressed in many ways. When a constraint like "Use `pnpm`" is encountered, should it block only explicit `npm` invocations, or also `yarn` and `bun`? Should it apply globally or only in certain directories?

We adopt a fail-closed philosophy where ambiguous constraints are interpreted strictly. "Use `pnpm`" triggers blocking of `npm`, `yarn`, and `bun` globally, even though a human reader might interpret this more narrowly. This approach may cause false positives, but we argue that this is the better trade-off for autonomous operation. False positives can be overridden by developers who understand the context, either by refining the Agent Instructions to be more precise or by marking

specific checks as exceptions. False negatives allow violations to slip through undetected and accumulate as technical debt. This philosophy aligns with security principles where the system denies the action when uncertainty remains and requires explicit authorization.

3 Evaluation

We evaluate ContextCov on the SWE-bench-Lite dataset to assess its extraction accuracy and its dynamic impact on autonomous agents. Our evaluation is driven by four research questions:

- **RQ1 (Extraction Accuracy):** How accurately does ContextCov translate natural-language Agent Instructions into semantically correct executable constraints?
- **RQ2 (Enforcement Effectiveness):** To what extent does active executable feedback reduce constraint violations compared to LLM-based self-reflection?
- **RQ3 (Functional Correctness):** Does enforcing strict constraints negatively impact the agent's ability to functionally resolve issues?
- **RQ4 (Overhead):** What is the relative cost and latency of executable feedback compared to LLM-based self-reflection?

Aggregated statistics are derived from 300 SWE-bench-Lite tasks executed across three agent modes.

3.1 Experimental Setup

We evaluate on the 12 repositories comprising the 300 tasks in SWE-bench-Lite: `astropy`, `django`, `matplotlib`, `seaborn`, `flask`, `requests`, `xarray`, `pylint`, `pytest`, `scikit-learn`, `sphinx`, and `sympy`. Because historical SWE-bench issues predate modern agentic workflows, we systematically generated repository-grounded `AGENTS.md` files for all 12 repositories. To ensure construct validity, we used a few-shot LLM generator to extract explicit constraints strictly from existing developer documentation (e.g., `CONTRIBUTING .md`), linting configurations (`pyproject.toml`, `setup.cfg`), and CI workflows (`.github/workflows/`, `pytest.ini`, `tox.ini`). We strictly prompted the generator against hallucination, requiring all rules to be supported by existing repository artifacts.

Given these instruction files, ContextCov generated domain-specific checks (source, process, architectural deterministic, and architectural semantic) and executed them across each repository to identify legacy violations and during the benchmark run to capture live violations.

We use OpenCode [7], an open-source agentic coding framework, as our agent scaffold. The coding LLM is GPT-5.2; the Self-reflection critic uses Claude Opus 4.5. We run all 300 tasks from SWE-bench Lite across three agentic configurations:

- **Vanilla (baseline):** the agent receives `AGENTS.md` in context but no active execution-time compliance feedback.
- **Self-reflection (baseline):** a critic LLM reviews candidate patches against `AGENTS.md` and returns natural-language feedback in a macro-loop.
- **ContextCov:** the same loop structure, but feedback is produced by executable checks with deterministic violation traces.

Feedback-conditioned runs (Self-reflection and ContextCov) share a maximum correction budget of five rounds.

3.2 RQ1: Extraction and Synthesis Quality

Across the 12 repositories, ContextCov compressed 1,498 lines of instruction files into 665 discrete, scope-annotated constraint segments (see Section 2.2), synthesizing 788 total executable checks (241 source, 296 process, 158 deterministic architectural, and 93 semantic architectural). To evaluate syntactic validity, we found that 780 of the 781 Python-based checks successfully parsed, indicating robust generation at the syntactic layer.

To evaluate semantic correctness, we ran these checks across the repositories, identifying 5,435 potential legacy violations. We uniformly sampled 384 violations (achieving a 95% confidence level with a ±4.72% margin of error) for manual, blind human annotation. By comparing the original instruction, the generated check, and the flagged code, we determined an extraction precision of 82.81% (318/384 True Positives). To validate our automated evaluation pipeline, we provided the same sample to an LLM-as-a-Judge (Claude Opus 4.5). The LLM judge achieved an 85.42% binary agreement with human annotators (Cohen's Kappa = 0.61), indicating substantial agreement and validating the automated judge for subsequent dynamic evaluations.

3.3 RQ2: Effectiveness of Active Constraint Enforcement

We define a **final clean** patch as one with zero actionable violations upon completion. For ContextCov, violations are intercepted in-loop; for baselines, we apply the oracle post-hoc.

ContextCov produced clean patches in 88.3% (265/300) of instances versus 67.0% for Vanilla and 50.3% for Self-reflection. Notably, LLM self-policing *introduces* more drift than no feedback at all. ContextCov also reduced violation density to 0.14 per patch (vs. 0.62 Vanilla, 1.01 Self-reflection), with 100% of residual violations limited to source-level checks—all process and architectural violations were eliminated.

ContextCov also converges faster: 1.67 mean rounds versus 2.85 for Self-reflection. Self-reflection exhausted the five-round cap on 85 instances (28%), while ContextCov hit the cap on only 29 (10%). Deterministic traces avoid the "hallucination looping" pattern where LLMs repeatedly fail to address the same issue.

Table 1. Repository-level statistics for extraction, check synthesis, and harness resolution. Seg. = constraint segments; Src = source checks; Proc = process checks; A-D = architectural-deterministic checks; A-S = architectural-semantic checks; Chk = total checks; Van. = Vanilla; SR = Self-reflection; CC = ContextCov; Δ = difference versus Vanilla.

	Extraction & Synthesis							Harness Resolution					
Repository	**Lines**	**Seg.**	**Src**	**Proc**	**A-D**	**A-S**	**Chk**	*n*	**Van.**	**%**	**SR**	**CC**	**Δ**
astropy	151	54	25	17	15	12	69	6	3	50	3	2	−1
django	110	61	35	22	5	11	73	114	68	60	70	77	+9
matplotlib	157	71	30	24	20	9	83	23	8	35	10	9	+1
seaborn	106	44	15	23	13	0	51	4	2	50	2	2	0
flask	102	52	25	38	15	1	79	3	1	33	1	1	0
requests	100	54	16	30	15	6	67	6	6	100	2	6	0
xarray	44	17	4	12	1	2	19	5	1	20	2	2	+1
pylint	255	130	13	76	27	19	135	6	3	50	2	0	−3
pytest	156	55	29	19	13	6	67	17	9	53	9	10	+1
scikit-learn	23	12	2	2	0	5	9	23	16	70	13	15	−1
sphinx	104	43	13	17	14	9	53	16	6	38	5	7	+1
sympy	190	72	34	16	20	13	83	77	36	47	38	41	+5
Total	**1,498**	**665**	**241**	**296**	**158**	**93**	**788**	**300**	**159**	**53**	**157**	**172**	+13

Table 2. Enforcement efficiency: oracle compliance, feedback rounds, and post-hoc LLM critic analysis (300 instances). Rnds_c = average rounds for clean patches; Avg. viol. = average violations per patch.

	Oracle Compliance & Feedback Rounds						Post-hoc LLM Critic				
Method	**Clean**	**Rate**	**Avg. viol.**	**Rounds**	Rnds_c	**Hit cap**	**Patch source**	**Clean**	*n*	**Avg.**	**Total**
ContextCov	265	88.3%	0.14	1.67	1.31	29 (10%)	ContextCov finals	30.1% (88/292)	292	1.15	336
Vanilla	201	67.0%	0.62	—	—	—	Vanilla finals	29.2% (87/298)	298	1.15	344
Self-reflection	151	50.3%	1.01	2.85	2.30	85 (28%)	Self-refl. finals	50.3% (151/300)	300	1.01	304

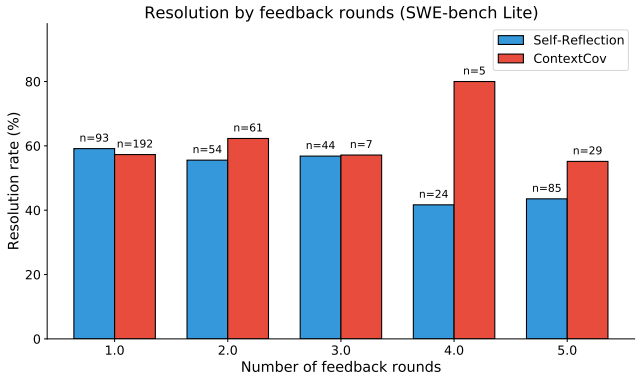

Figure 3. Harness resolution rate versus number of feedback rounds.

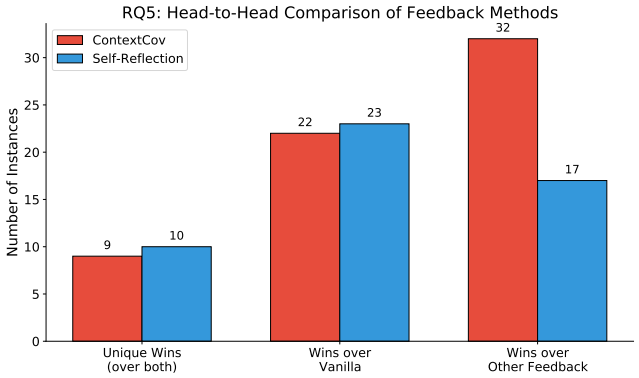

Figure 4. Head-to-head wins when one method resolves an instance another does not.

On the 49 instances where ContextCov and Self-reflection disagreed on pass/fail, ContextCov resolved successfully 32 times versus 17 for Self-reflection, confirming that deterministic enforcement yields better patch trajectories.

We also ran the self-reflection critic post-hoc on Vanilla and ContextCov finals. The critic rates both similarly (~29–30% clean), suggesting its complaints are orthogonal to ContextCov's constraint categories. Self-reflection's own finals achieve 50.3% critic-clean—the highest—yet the lowest harness resolution (52.3%), indicating LLM critics fixate on preferences unrelated to functional correctness (Table 2, right).

Table 3. Functional correctness: harness resolution rates and pairwise wins (n=300). vs Van. = wins versus Vanilla; vs other FB = wins versus the other feedback method.

Method	Resolved	Rate	vs Van.	vs other FB	Unique
ContextCov	172	57.3%	+22	+32	9
Self-reflection	157	52.3%	+23	+17	10
Vanilla	159	53.0%	—	+25	2

Table 4. Cross-tabulation of oracle cleanliness and harness resolution (n=300 per method). Clean = zero actionable violations; Dirty = one or more violations.

Method	Oracle	Pass	Fail	Total
ContextCov	Clean	156	109	265
	Dirty	16	19	35
Self-refl.	Clean	85	66	151
	Dirty	72	77	149
Vanilla	Clean	110	91	201
	Dirty	49	50	99

3.4 RQ3: Impact on Functional Correctness

To evaluate functional utility, we applied the official SWE-bench-Lite evaluation harness to the final patches generated by each method. We use the test pass/fail outcome as our primary proxy for whether executable constraints introduce net harm (e.g., via false-positive pressure that steers the agent away from a valid fix) or net lift.

ContextCov successfully resolved 57.3% (172/300) of the SWE-bench instances, outperforming both the Vanilla baseline (53.0%) and the Self-reflection baseline (52.3%). To determine statistical significance, we applied McNemar's test on paired pass/fail outcomes. The discordant pairs were asymmetric in favor of ContextCov, and the tests rejected the null hypothesis of equal marginal failure rates for ContextCov versus Vanilla ($p = 0.031$) and ContextCov versus Self-reflection ($p = 0.046$). These results strongly reject the hypothesis of net harm, demonstrating that active, deterministic constraint enforcement actively improves end-to-end patch resolution.

Table 3 also reports directional wins (method A resolves, B does not) alongside "unique" counts—instances where only that method passes the harness while both alternatives fail. ContextCov achieves 22 wins over Vanilla and 32 over Self-reflection; critically, it claims 9 unique successes that neither baseline could match. This complements the McNemar analysis: statistical significance arises because ContextCov's wins outnumber its losses, not from isolated outliers.

To ensure the aggregate gains were not driven by an isolated anomaly, we analyzed resolution rates across the 12 individual repositories (Table 1, right). ContextCov's gains were primarily clustered in the largest domain subsets, notably `django` (+9 resolved over Vanilla) and `sympy` (+5 over Vanilla). ContextCov performed equally to baselines on mid-sized subsets like `requests` and `seaborn`. We observed a minor regression on `pylint` (0/6 resolved vs. 3/6 for Vanilla), indicating that while the approach generalizes well across most domains, highly specific framework constraints may occasionally restrict viable patch trajectories, motivating future per-ecosystem tuning.

To understand *why* agents fail, we cross-tabulated actionable oracle cleanliness against harness outcomes. As shown in Table 4, ContextCov's failures are largely independent of constraint violations, unlike the baselines. For ContextCov, the vast majority of harness failures (109 of 128) occurred on oracle-clean patches. This crucial finding indicates that when ContextCov fails to resolve an issue, it is rarely because the constraints trapped the agent; rather, the agent simply failed to deduce the underlying functional logic of the bug. Conversely, for the Self-reflection and Vanilla baselines, oracle-dirty patches were significantly correlated with harness failures (48.3% vs. 56.3% resolution for SR; 49.5% vs. 54.7% for Vanilla), suggesting that uncorrected Context Drift directly contributes to functional task failure.

Finally, we analyzed the overlap in success across the three methods. Of 300 instances, 127 passed under all three configurations, while 109 failed across the board. Notably, at least one method passed on 191 instances, yielding an implied union success rate of 63.7%. This indicates that while ContextCov is the strongest individual method, different feedback mechanisms occasionally shepherd the agent down disparate, successful architectural paths.

3.5 RQ4: Overhead and Cost-Efficiency

To measure operational overhead, we tracked the volume of logged characters sent to the LLM (as a proxy for token cost) and the wall-clock time required to generate feedback during the correction loops.

ContextCov significantly reduces the overhead associated with in-loop evaluations. The Self-reflection agent required a mean of 19,455 input characters per instance on the critic path, taking an average of 133.7 seconds of wall-clock time to generate feedback. By contrast, ContextCov required a mean of only 6,292 input characters (predominantly for semantic architectural checks) and reduced the mean feedback generation time to just 27.7 seconds. While end-to-end execution time remains dominated by the agent's environment setup and test execution, ContextCov reduces the isolated feedback latency by nearly 80%.

A critical factor in deploying autonomous agents is the predictability of their operating costs. The Self-reflection agent exhibited massive dispersion in its LLM load, with a standard deviation of 22,684 characters (compared to its mean of 19,455), indicating a heavy-tailed distribution where

Table 5. Feedback cost proxies (300 instances). LLM input chars = characters sent to the LLM for feedback generation.

Metric	Method	Mean	Sum
LLM input chars	Self-refl.	19,455	5.84M
	ContextCov	6,292	1.89M
Feedback time (s)	Self-refl.	133.7	40,110
	ContextCov	27.7	8,310
		Mean	**Median**
End-to-end (s)	Self-refl.	461.4	379.2
	ContextCov	422.4	240.0
	Vanilla	327.9	184.7

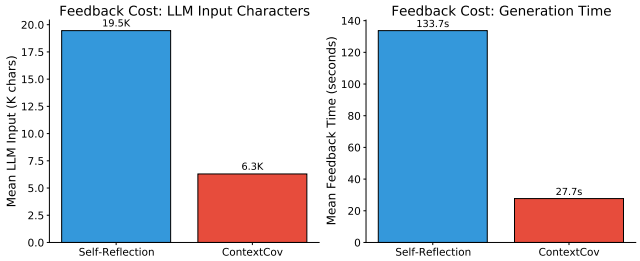

Figure 5. Mean logged LLM input characters (left) and mean feedback-generation wall time (right) for Self-reflection versus ContextCov.

a subset of complex instances causes reflection costs to spiral. ContextCov drastically stabilizes this variance, exhibiting a standard deviation of only 5,891 characters (median 4,168), providing a highly predictable, constrained cost profile suitable for production deployment.

To evaluate the return on investment for each feedback mechanism, we normalized the total number of successful harness resolutions against the aggregate volume of logged characters used to generate feedback. The Self-reflection agent achieved approximately 26.9 task resolutions per million logged characters. ContextCov achieved 91.1 resolutions per million characters. This represents a **3.4× improvement in cost-efficiency**, demonstrating that ContextCov's deterministic interceptors not only yield higher functional success rates but do so at a fraction of the computational cost of LLM-based reflection.

4 Discussion

The shift to autonomous agents fundamentally changes the role of project documentation from descriptive guides for humans to prescriptive execution contexts for AI. Context Drift is a symptom of the mismatch between how we document intent and how agents consume it.

Documentation as Code and Interpretability. ContextCov forces a paradigm shift where documentation must co-evolve with the codebase. By adopting a strict, fail-closed interpretation of documented constraints, the tool treats instruction ambiguity not as a minor annoyance, but as a build-breaking defect. When an agent hits an enforcement block, it triggers a feedback loop: either the agent corrects its action, or developers must refine the `AGENTS.md`. Furthermore, compiling natural language into deterministic Python checks (AST queries and NetworkX graphs) provides executable interpretability. Developers can audit the generated checks to debug vague prompts, gaining visibility into exactly how an LLM interprets subjective instructions like "clean architecture."

Guardrails vs. Sandboxes. It is important to note that ContextCov is a workflow guardrail, not a kernel-level security sandbox (e.g., seccomp, gVisor). It is designed to prevent a well-intentioned but hallucinating agent from degrading repository conventions, assuming authorized maintainers and a non-adversarial environment.

5 Related Work

Documentation and Specifications. Prior work addresses the "stale docs" problem for descriptive documentation (comments, APIs) [6, 14, 15] and explores translating natural language into formal specifications or linting rules (e.g., ESLint [1], Semgrep [2]) [11, 13]. ContextCov shifts this focus to prescriptive agent instructions, leveraging domain-routed synthesis to enforce rules dynamically via ASTs, dependency graphs, and shell shims.

Agent Alignment and Conformance. While agent alignment frameworks (e.g., Constitutional AI [8]) focus on general-purpose safety, ContextCov provides domain-specific alignment for software engineering contexts. Similarly, while existing architectural conformance tools (e.g., Sonargraph [3], ArchJava [9]) require architects to manually encode rules in tool-specific formats, ContextCov lowers the barrier to entry by directly extracting structural and process constraints from existing developer documentation.

6 Limitations and Future Work

ContextCov may flag false positives if the LLM extraction misinterprets intent, if the synthesized check is overly strict, or if instructions lack explicit exceptions. However, these surface as direct signals to refine the `AGENTS.md` file. Additionally, our evaluation on the SWE-bench-Lite Python split does not exhaust all agent frameworks or language ecosystems. While Tree-sitter supports cross-language AST parsing, broader external validity requires replication on other benchmarks and build systems.

Future directions include ablations over constraint families to identify which domains drive the highest resolution gains, scaling evaluations to the full SWE-bench dataset, and investigating human-in-the-loop patch reviews.

7 Conclusion

As autonomous agents tackle increasingly complex, multi-file tasks, verifying architectural boundaries and process requirements by manual inspection becomes impractical. The specifications that reside in developer documentation are passive text; without automated enforcement, Context Drift accumulates silently as technical debt.

ContextCov addresses this challenge by transforming passive agent instructions into active, executable invariants. Through hierarchical constraint extraction, domain-specialized code synthesis, and multi-layer runtime enforcement, ContextCov provides immediate, deterministic feedback to agents. Our evaluation on SWE-bench-Lite demonstrates that ContextCov accurately extracts constraints and enforces them with significantly higher compliance than LLM-based self-reflection. Furthermore, it achieves this at a fraction of the computational cost and latency, all while actively improving the agent's functional issue resolution rate.

Ultimately, as autonomous agents become standard contributors, frameworks like ContextCov will be essential for maintaining repository integrity and bridging the gap between developer intent and autonomous execution.

References

[1] [n. d.]. Find and fix problems in your JavaScript code - ESLint - Pluggable JavaScript Linter. https://eslint.org/. Accessed: 2026-02-16.

[2] [n. d.]. GitHub - semgrep/semgrep: Lightweight static analysis for many languages. https://github.com/semgrep/semgrep. Accessed: 2026-02-16.

[3] [n. d.]. hello2morrow. https://www.hello2morrow.com/products/sonargraph. Accessed: 2026-02-16.

[4] [n. d.]. Introduction - Tree-sitter — tree-sitter.github.io. https://tree-sitter.github.io/tree-sitter/. [Accessed 26-04-2026].

[5] [n. d.]. vscode/.github/copilot-instructions.md at main · microsoft/vscode — github.com. https://github.com/microsoft/vscode/blob/main/.github/copilot-instructions.md. [Accessed 26-04-2026].

[6] Emad Aghajani, Csaba Nagy, Olga Vega-Márquez, Mario Linares-Vásquez, Laura Moreno, Gabriele Bavota, and Michele Lanza. 2019. Software Documentation Issues Unveiled. In *Proceedings of the 41st International Conference on Software Engineering*. 1199–1210. doi:10.1109/ICSE.2019.00122

[7] Anomaly. [n. d.]. OpenCode: An Open-Source Agentic Coding Framework. https://opencode.ai. Accessed: 2026-04-26.

[8] Yuntao Bai, Saurav Kadavath, Sandipan Kundu, Amanda Askell, Jackson Kernion, Andy Jones, Anna Chen, Anna Goldie, Azalia Mirhoseini, Cameron McKinnon, Carol Chen, Catherine Olsson, Christopher Olah, Danny Hernandez, Dawn Drain, Deep Ganguli, Dustin Li, Eli Tran-Johnson, Ethan Perez, Jamie Kerr, Jared Mueller, Jeffrey Ladish, Joshua Landau, Kamal Ndousse, Kamile Lukosuite, Liane Lovitt, Michael Sellitto, Nelson Elhage, Nicholas Schiefer, Noemi Mercado, Nova DasSarma, Robert Lasenby, Robin Larson, Sam Ringer, Scott Johnston, Shauna Kravec, Sheer El Showk, Stanislav Fort, Tamera Lanham, Timothy Telleen-Lawton, Tom Conerly, Tom Henighan, Tristan Hume, Samuel R. Bowman, Zac Hatfield-Dodds, Ben Mann, Dario Amodei, Nicholas Joseph, Sam McCandlish, Tom Brown, and Jared Kaplan. 2022. Constitutional AI: Harmlessness from AI Feedback. arXiv:2212.08073 [cs.CL] https://arxiv.org/abs/2212.08073

[9] Craig Chambers and David Notkin. 2002. ArchJava: Connecting Software Architecture to Implementation. *Proceedings - International Conference on Software Engineering* (03 2002). doi:10.1145/581339.581365

[10] Giuseppe Crupi, Rosalia Tufano, Alejandro Velasco, Antonio Mastropaolo, Denys Poshyvanyk, and Gabriele Bavota. 2025. On the Effectiveness of LLM-as-a-judge for Code Generation and Summarization. arXiv:2507.16587 [cs.SE] https://arxiv.org/abs/2507.16587

[11] Shalini Ghosh, Daniel Elenius, Wenchao Li, Patrick Lincoln, Natarajan Shankar, and Wilfried Steiner. 2016. ARSENAL: automatic requirements specification extraction from natural language. In *NASA Formal Methods Symposium*. Springer, 41–46.

[12] Microsoft. n.d.. GitHub - microsoft/vscode: Visual Studio Code. https://github.com/microsoft/vscode. [Accessed 2026-04-26].

[13] Rahul Pandita, Xusheng Xiao, Hao Zhong, Tao Xie, Stephen Oney, and Amit Paradkar. 2012. Inferring method specifications from natural language API descriptions. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 815–825.

[14] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. 2007. /* iComment: Bugs or bad comments?*/. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*. 145–158.

[15] Gias Uddin and Martin P Robillard. 2015. How API Documentation Fails. *IEEE Software* 32, 4 (07 2015), 68–75. doi:10.1109/MS.2014.80