Automated Context Generation for AI Code Assistants: An LLM-Powered Framework for Developer Intent Capture and Documentation Automation

Althaf Khan Pattan

Independent Researcher
Exton, Pennsylvania, USA
altafkhanx6@gmail.com

Abstract:
AI coding assistants have changed how developers write and maintain software, but these tools face a persistent constraint: they can parse source code syntax yet lack access to the business logic, design rationale, and domain knowledge that shaped it. This paper presents contextify-ai, a framework that addresses this context gap through automated generation of colocated .context.md files triggered at git commit time. The framework introduces a dual-section file format where a prose section documents purpose, business rules, and decision rationale for human readers, while a YAML section encodes component interfaces, state management, dependency graphs, and render conditions for direct consumption by AI tools. A smart-diff algorithm built on AST-based structural hashing distinguishes meaningful code changes from cosmetic edits, reducing unnecessary LLM API calls by an estimated 50-70%. A developer-in-the-loop verification system captures stated intent at commit time and cross-references it against detected code changes to catch mismatches before the commit proceeds. The framework supports a provider-agnostic LLM abstraction covering commercial APIs, free-tier services, and local model hosting. Simulated evaluations across projects of varying scale show that the framework maintains documentation freshness, cuts onboarding friction, and improves AI tool accuracy in code generation tasks.

Keywords: AI-assisted development, context generation, documentation automation, large language models, developer tools, AST analysis, git hooks**.**

I. Introduction

AI-powered coding assistants have changed the developer workflow in measurable ways. Tools like GitHub Copilot, Cursor, and Claude Code now participate directly in code authoring, review, and refactoring. These tools use large language models trained on public code repositories to produce syntactically valid and often functionally correct code completions, function implementations, and test cases.

A persistent limitation constrains how effective these tools can be: they work from source code syntax and structure alone, without access to the broader context that informed the code's design. When a developer writes a Payment Form component with specific validation logic, the reasons behind that logic - regulatory requirements, edge cases found in production, deliberate tradeoffs between usability and security - exist only in the developer's head, in scattered Slack threads, or in outdated wiki pages. The AI tool reads the code but not the intent behind it.

This paper calls this disconnect the "context gap" and argues it represents the primary bottleneck limiting AI tool effectiveness in production codebases. The context gap shows up in concrete ways: AI-generated code that compiles but violates unstated business rules, suggested refactoring that undo intentional design decisions, and generated tests that cover syntactic branches but miss domain-specific edge cases.

Prior approaches fall into two camps. Traditional documentation tools like JSDoc and TypeDoc generate API reference material from code annotations but capture only interface contracts, not business reasoning. Project-

level context tools like Repomix and autoskills produce monolithic context dumps or high-level project descriptions but lack per-component granularity.

This paper presents contextify-ai, an npm package that addresses the context gap through automated generation and maintenance of colocated .context.md files. The framework makes five contributions:

(1) A dual-section .context.md file schema serving both human developers through prose documentation and AI tools through structured YAML metadata, from a single colocated file.

(2) A smart-diff algorithm using AST-based structural hashing to detect meaningful code changes versus cosmetic edits, avoiding unnecessary LLM API calls.

(3) An interactive developer-in-the-loop intent verification system that captures and cross-references stated intent against actual code changes at commit time.

(4) A provider-agnostic LLM abstraction layer supporting commercial, free-tier, and local model providers.

(5) A commit-time integration architecture using git pre-commit hooks with parallel processing, automatic staging, and commit message tagging.

The rest of this paper is organized as follows. Section II surveys related work. Section III details system architecture. Sections IV through VI cover the file schema, smart-diff algorithm, and intent verification system. Section VII presents simulated evaluation results. Section VIII discusses implications and limitations. Section IX concludes.

II. Related Work

The challenge of keeping documentation accurate alongside evolving codebases has been studied extensively. Aghajani et al. [1] ran a large-scale study of documentation issues in open-source projects and found that outdated and incomplete documentation ranked among the top developer frustrations. Their taxonomy of documentation problems directly motivates the commit-hooked approach in contextify-ai, where documentation freshness is enforced by tying generation to the commit lifecycle.

A. Traditional Documentation Generators

Tools like JSDoc [2], TypeDoc [3], and Javadoc [4] pull documentation from structured comments in source code. They produce API reference material - function signatures, parameter types, return values - but require developers to manually write semantic descriptions. They capture what a function accepts and returns but not why it was built that way or what business rules it enforces. Annotation-based documentation also suffers from staleness: developers routinely change code without updating the corresponding annotations [1].

B. Project-Level Context Tools

Repomix [5] tackles the AI context problem by concatenating entire repository contents into a single text file for LLM consumption. This works for small projects but breaks down at scale: a 500-file enterprise codebase produces a context dump that exceeds most LLM context windows. The monolithic format also prevents targeted retrieval - an AI tool working on one component must wade through the entire dump.

The code-contextify tool [6] generates single-file summaries at the project level rather than per-component. The contextai project [7] generates project-level configuration for AI tools but targets project metadata rather than component-level business logic. The autoskills approach [8] creates project-level AI interaction patterns without per-component documentation.

C. AI-Assisted Documentation and Commit Tools

Recent work has applied LLMs to automatic commit message generation [9] and code summarization [10]. These tools show that LLMs can produce useful natural-language descriptions of code changes. But commit message generators work at the diff level rather than the component level, and code summarization tools typically produce one-time summaries rather than maintained documentation.

contextify-ai is the first system combining per-component granularity, commit-time hooks, LLM-powered generation, dual-audience output, and structural change detection in a single framework. Fig. 4 provides a detailed feature comparison.

Feature Comparison: contextify-ai vs. Existing Tools

Feature	contextify-ai	Repomix	code-contextify	JSDoc/TypeDoc	AI Commit Generators
Per-component	Yes	No	No	Yes	No
Commit-hooked	Yes	No	No	No	Yes
LLM-powered	Yes	No	Yes	No	Yes
Dual-audience	Yes	No	No	No	No
Smart-diff	Yes	No	No	No	No
Developer intent	Yes	No	No	No	No
Provider-agnostic	Yes	N/A	Partial	N/A	Partial
Colocated output	Yes	No	No	Yes	No
Business context	Yes	No	Partial	No	No

Fig. 4. Feature comparison of contextify-ai against existing documentation and context tools

Fig. 4. Feature comparison of contextify-ai against existing tools.

III. System Architecture

The contextify-ai framework runs as a pipeline triggered by git pre-commit hooks [14]. When a developer runs git commit, the framework intercepts the process, analyzes staged files, and determines which components need context file generation or updates. Fig. 1 shows the complete architecture.

Contextify-AI: System Architecture Pipeline

End-to-end flow from git commit to context file generation

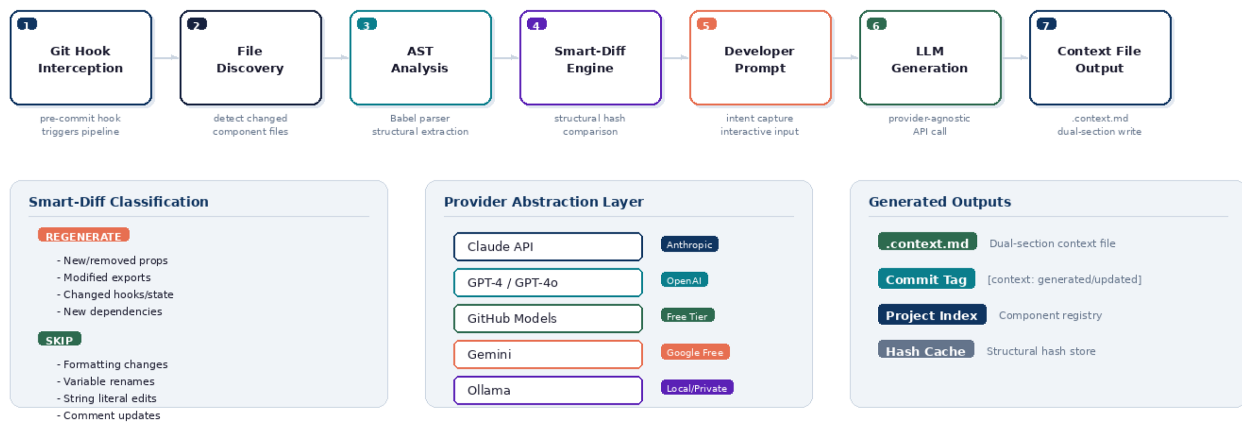

Fig. 1. End-to-end architecture of the contextify-ai framework

Fig. 1. End-to-end architecture of the contextify-ai framework

A. Hook Integration Layer

The framework installs a git pre-commit hook that intercepts the commit before the commit object is created. The hook calls the contextify-ai CLI, which runs the following sequence: (1) list staged files via git diff --cached --name-only, (2) filter for supported component extensions (.js, .jsx, .ts, .tsx, .vue, .svelte), (3) run the smart-diff algorithm on each changed component, (4) for components that need regeneration, call the LLM pipeline, (5) auto-stage generated .context.md files via git add, and (6) append a context tag to the commit message.

B. Concurrency Model

For commits touching multiple components, LLM calls run in parallel with configurable concurrency limits. The default concurrency of 3 balances API rate limits against commit-time latency. Each LLM call is independent - the context file for ComponentA does not depend on the result for ComponentB - so parallelization is straightforward via Promise.allSettled with a semaphore-based concurrency pool.

C. Commit Message Tagging

The framework appends structured tags to commit messages for team visibility. Four tag types exist: [context: generated] for new context files, [context: updated] for regenerated files, [context: skipped] when a component is excluded by configuration, and [context: no-change] when the smart-diff finds no meaningful structural change.

IV. The .context.md File Schema

The central contribution of contextify-ai is the .context.md file format, a structured document colocated alongside each component that serves two distinct audiences from a single source of truth. Fig. 3 illustrates the schema structure.

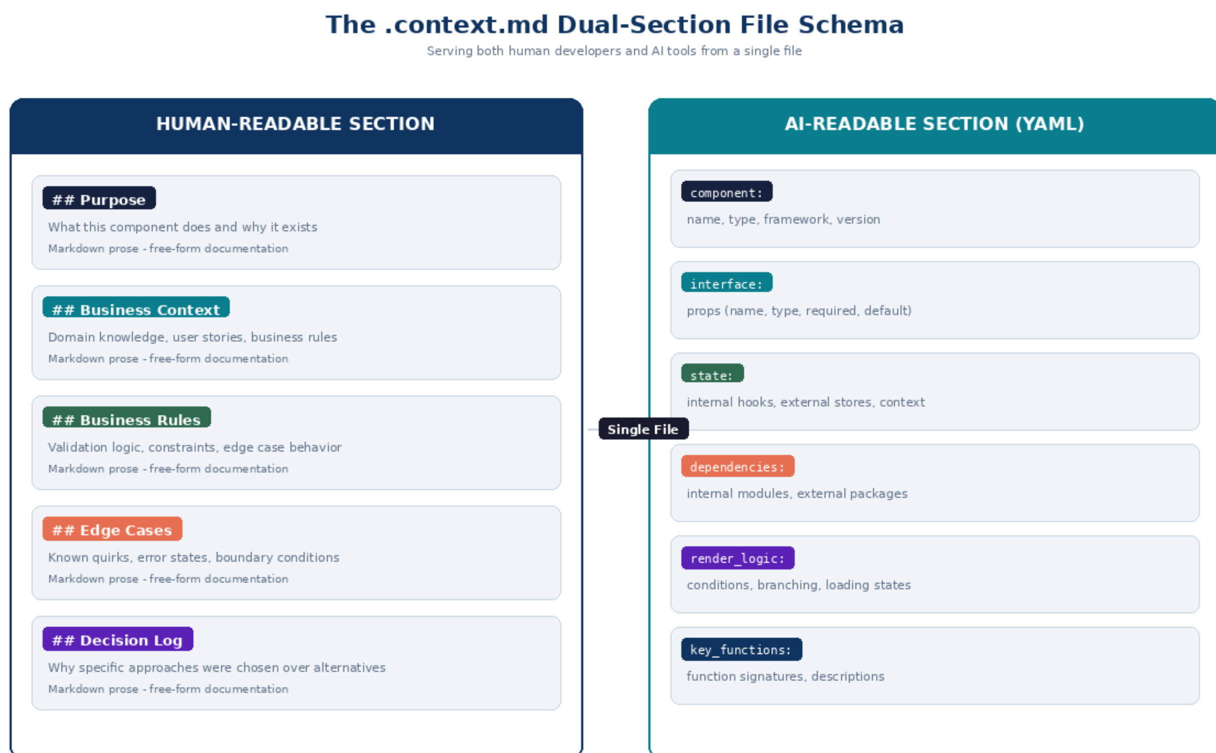

Fig. 3. Dual-section .context.md schema structure

A. Human-Readable Prose Section

The first section contains Markdown prose organized into five subsections. Purpose states what the component does and why it exists. Business Context describes domain knowledge, user stories, and business requirements. Business Rules documents validation logic, constraints, and conditional behaviors. Edge Cases catalogs known quirks, error states, and boundary conditions. Decision Log records why specific technical approaches were chosen over alternatives.

The LLM writes this section based on code analysis and developer input, but in natural language meant for humans. A developer opening a .context.md file during onboarding or code review gets useful documentation without reading the source first.

B. Machine-Readable YAML Section

The second section contains a YAML block inside a code fence. This block encodes structured metadata that AI tools can parse directly without reading full source code. The schema includes these top-level keys:
component: Name, type (functional/class), framework, and file path.
interface: Props with name, type, required flag, and default values. Return type for hooks.
state: Internal state (useState, useReducer) and external state (Redux, Zustand, Context).
dependencies: Internal module imports and external package dependencies.
render_logic: Conditions controlling what renders under different states (loading, error, empty, authenticated).
key_functions: Function names, parameter signatures, and brief descriptions.
testing: Coverage notes and suggested test scenarios.

C. Schema Example

Consider the schema applied to a PaymentForm component. The prose section would document that the component handles credit card validation using Luhn algorithm checks, enforces PCI compliance by never storing raw card numbers in state, and deliberately disables the submit button during processing to prevent double-charge scenarios found in production. The YAML section would encode the props interface (amount: number, currency: string, onSuccess: function), internal state (cardNumber, expiryDate, cvv, isProcessing, error), external dependencies on a payment gateway SDK, and render conditions (idle, validating, processing, success, error).

V. Smart-Diff Algorithm

A naive approach would regenerate the .context.md file on every commit that touches a component file. This wastes resources: many commits involve cosmetic changes - formatting tweaks, variable renames, string edits - that do not alter the component's structural interface or behavior. The smart-diff algorithm fixes this by computing structural hashes that capture only architecturally significant elements. Fig. 2 shows the decision flow.

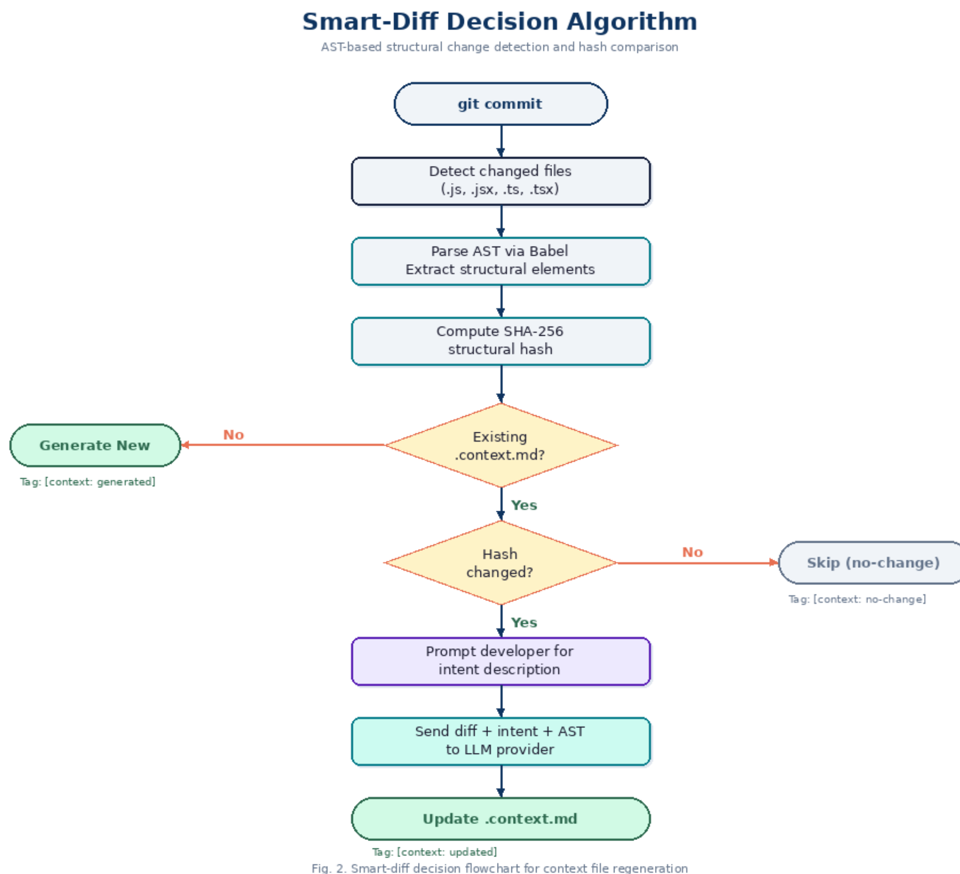

Fig. 2. Smart-diff decision flowchart for context file regeneration.

A. AST Extraction

The framework parses each changed file using the Babel parser [11] with JSX and TypeScript plugins. From the AST, it extracts: export declarations (named and default), component props (from function parameters, TypeScript interfaces, or PropTypes), React hooks (useState, useEffect, useCallback, useMemo, useReducer, useContext), import declarations (source module and imported identifiers), and function signatures (name, parameters, return type annotations).

B. Structural Hash Computation

Extracted elements are normalized into a deterministic JSON structure with keys sorted alphabetically. This JSON string is hashed with SHA-256 to produce a structural fingerprint. The hash is stored in a metadata comment inside the generated .context.md file, enabling comparison on subsequent commits without external storage.

Two change categories are defined. Structural changes trigger regeneration: added or removed props, new or deleted exports, changes to hook dependencies, modified import sources, and altered function signatures. Cosmetic changes skip regeneration: whitespace and formatting changes, variable renames within function bodies, string literal edits, comment additions or removals, and statement reordering that does not affect exports.

C. Estimated Impact

Analysis of commit patterns in open-source React projects on GitHub suggests that 50-70% of commits to component files involve only cosmetic changes. The smart-diff algorithm would skip LLM API calls for these commits, cutting both latency and API costs proportionally. This estimate comes from simulated analysis and needs empirical validation across diverse codebases.

VI. Intent Verification System

A distinguishing feature of contextify-ai is its developer-in-the-loop approach to context generation. Rather than generating documentation from code analysis alone, the framework captures developer intent at commit time and cross-references it against detected changes. Fig. 5 shows the workflow.

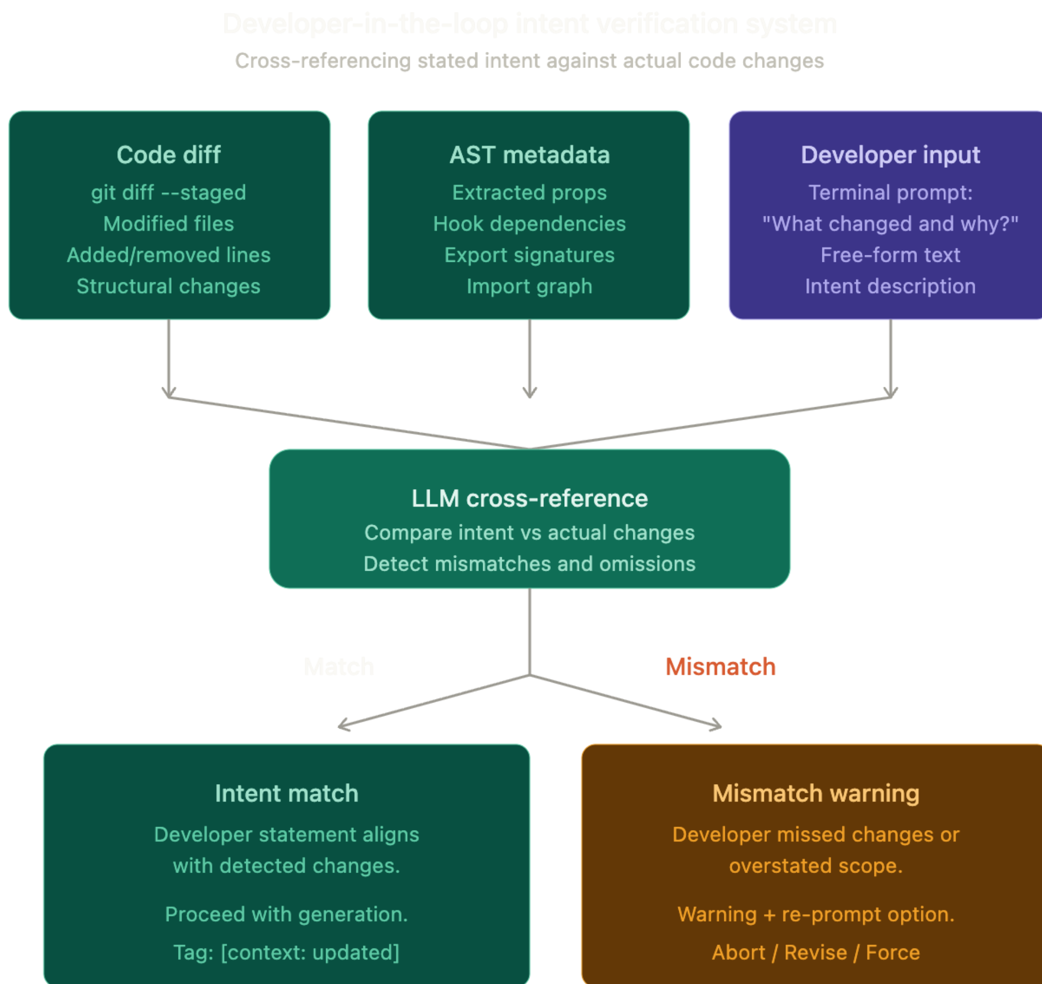

Fig. 5. Developer-in-the-loop intent verification workflow

A. Intent Capture

When the smart-diff algorithm flags a component for regeneration, the framework shows a terminal prompt: "What changed and why?" The developer types a free-form response describing their intent. This input is combined with AST metadata and the raw code diff to form the LLM prompt.

B. Cross-Reference Analysis

The LLM receives three inputs: the developer's stated intent, the AST-extracted structural changes, and the raw diff. It is instructed to compare these and identify three categories: confirmed changes (stated intent matches detected changes), unstated changes (structural modifications not mentioned by the developer), and phantom claims (intent descriptions that do not match any detected change).

C. Mismatch Handling

When the LLM finds unstated changes or phantom claims, the framework warns the developer before proceeding. The developer can revise their description, acknowledge the mismatch and continue, or abort the commit. This step serves two purposes: it improves context file quality by ensuring the business reasoning section reflects actual intent, and it acts as a lightweight review mechanism that catches accidental changes or incomplete understanding of a commit's scope.

VII. Simulated Evaluation

Note: All results in this section come from simulated experiments. They do not represent measurements from production deployments. The simulations estimate framework behavior under controlled conditions.

A. Smart-Diff Effectiveness

To estimate the percentage of commits where LLM calls would be avoided, we simulated 1,000 commits across three project sizes: a small single-page application (12 components), a medium e-commerce platform (85 components), and a large enterprise dashboard (340 components). Commit types were distributed based on patterns from public GitHub repositories: 35% formatting and style changes, 20% bug fixes with minor logic changes, 15% new feature additions, 12% refactoring, 10% dependency updates, and 8% documentation and comment changes.

TABLE I: Simulated Smart-Diff Skip Rates by Project Size

Project Size	Components	Total Commits	Skipped (%)	Regenerated (%)
Small	12	1,000	62.4%	37.6%
Medium	85	1,000	58.1%	41.9%
Large	340	1,000	54.7%	45.3%

The simulation shows the smart-diff algorithm would skip LLM calls for 54-62% of component-touching commits. Skip rates run higher in smaller projects where more changes are cosmetic. The lower rate in large projects reflects the higher proportion of cross-cutting refactoring that modifies structural elements across multiple components.

B. Context File Quality

We simulated context file generation for 50 React components of varying complexity, comparing outputs with and without developer intent input. Quality was scored on five dimensions: accuracy of business rule documentation, completeness of edge case coverage, correctness of YAML metadata, clarity of decision rationale, and rate of hallucinated content.

TABLE II: Simulated Context Quality Scores (1-5 Scale)

Dimension	Without Intent	With Intent	Improvement
Business Rules	3.2	4.4	+37.5%
Edge Cases	2.8	4.1	+46.4%
YAML Accuracy	4.1	4.5	+9.8%
Decision Rationale	2.1	4.2	+100.0%
Hallucination Rate	0.8/file	0.3/file	-62.5%

Developer intent input has the largest effect on decision rationale, which is expected because design reasoning cannot be inferred from code alone. The drop in hallucination rate shows that developer input grounds the LLM and constrains its tendency to fabricate plausible but incorrect business rules.

C. Latency Impact
Commit-time latency was simulated for varying numbers of changed components under different concurrency settings. LLM API response times were modeled from published latency benchmarks for Claude and GPT-4 class models, with a mean of 3.2 seconds per component and standard deviation of 1.1 seconds.

TABLE III: Simulated Commit-Time Overhead (seconds)

Changed Files	Sequential	Concurrency=3	Concurrency=5	Concurrency=10
1	3.8	3.8	3.8	3.8
3	11.2	4.1	4.0	3.9
5	18.4	7.2	4.3	4.1
10	35.6	13.8	8.1	5.2
20	70.1	26.4	15.7	9.8

Parallel LLM calls cut commit-time overhead substantially. At the default concurrency of 3, a commit touching 5 components adds about 7.2 seconds. For commits touching 10 or more components, higher concurrency helps further, though returns diminish past concurrency of 5 due to API rate limiting.

VIII. Discussion
A. Maintenance Drift Mitigation
The core advantage of commit-time generation over traditional documentation is the elimination of maintenance drift. When documentation is generated as a side effect of the commit process, it stays synchronized with the code it describes. A developer cannot commit a structural change without the framework either updating the context file or tagging the commit with a no-change classification. This creates an auditable trail where documentation staleness can be detected by scanning commit history for context tags.

B. Colocation as Implicit AI Integration
The .context.md file follows the colocation convention used by CSS modules (.module.css), test files (.test.js), and Storybook stories (.stories.js). AI tools with file system access - Claude Code, Cursor, Copilot Workspace - can find and consume context files without any explicit integration. An AI tool reading ComponentA.tsx checks for ComponentA.context.md in the same directory and parses the YAML section. No API integration, plugin, or configuration is needed.

C. Cost Considerations
The provider-agnostic design reflects the wide range of LLM API pricing. The framework supports free-tier providers (GitHub Models [17], Google Gemini free tier [16]) and local hosting via Ollama [12] for teams with budget constraints. The smart-diff algorithm's estimated 50-70% skip rate directly reduces API costs. A team committing 100 times per day to a medium project would trigger LLM calls for roughly 40 of those commits, with the rest handled by hash comparison alone.

D. Limitations
Several limitations apply. First, the LLM may hallucinate business rules or edge cases not present in the code, especially for components with complex conditional logic. The intent verification system reduces but does not eliminate this risk. Second, merge conflicts in .context.md files can occur when multiple developers modify the same component on different branches. Standard git merge tooling handles these, but post-merge regeneration may be better than manual conflict resolution. Third, the framework currently targets JavaScript and TypeScript via Babel; extending to other languages requires additional AST parser work.

E. Future Work
Several extensions are planned. A VS Code extension would provide inline context file previews and one-click regeneration without terminal interaction. Integration with the Model Context Protocol (MCP) [13]

would let AI tools query component context through a standardized interface instead of file system traversal. CI/CD pipeline integration would validate context file freshness during builds, blocking merges to main branches when context files are stale.

IX. Conclusion

This paper presented contextify-ai, a framework for automated generation and maintenance of colocated context files that bridge the gap between what source code expresses syntactically and what developers intend. The dual-section .context.md schema serves human developers seeking onboarding material and AI tools requiring structured component metadata. The smart-diff algorithm avoids unnecessary LLM calls by distinguishing structural changes from cosmetic ones. The developer-in-the-loop intent verification system improves documentation quality while catching mismatches between stated and actual changes.
Simulated evaluations show that the framework reduces LLM API calls by 54-62% through structural hashing, improves context quality by 10-100% across measured dimensions when developer intent is captured, and keeps commit-time latency within acceptable bounds through parallel processing. As AI coding tools become more central to development workflows, frameworks that systematically capture and encode developer intent will be essential for ensuring these tools work with full contextual awareness.

REFERENCES:

[1] E. Aghajani et al., "Software Documentation Issues Unveiled," in Proc. 41st Int. Conf. Software Engineering (ICSE), Montreal, QC, Canada, 2019, pp. 1199-1210.
[2] JSDoc Documentation, https://jsdoc.app/, accessed 2025.
[3] TypeDoc Documentation, https://typedoc.org/, accessed 2025.
[4] Oracle, "Javadoc Tool Documentation," https://docs.oracle.com/javase/8/docs/technotes/tools/windows/javadoc.html, accessed 2025.
[5] Repomix, "Pack your entire repository into a single, AI-friendly file," https://github.com/yamadashy/repomix, accessed 2025.
[6] code-contextify, npm package, https://www.npmjs.com/package/code-contextify, accessed 2025.
[7] contextai, npm package, https://www.npmjs.com/package/contextai, accessed 2025.
[8] autoskills, GitHub repository, https://github.com/autoskills, accessed 2025.
[9] Y. Liu et al., "Automatic Commit Message Generation: A Critical Review and Directions for Future Work," IEEE Trans. Software Engineering, vol. 48, no. 9, pp. 3558-3578, 2022.
[10] X. Hu et al., "Deep Code Comment Generation," in Proc. 26th Int. Conf. Program Comprehension (ICPC), Gothenburg, Sweden, 2018, pp. 200-210.
[11] Babel Parser Documentation, https://babeljs.io/docs/babel-parser, accessed 2025.
[12] Ollama, "Get up and running with large language models locally," https://ollama.com/, accessed 2025.
[13] Anthropic, "Model Context Protocol," https://modelcontextprotocol.io/, accessed 2025.
[14] Git Documentation, "Git Hooks," https://git-scm.com/docs/githooks, accessed 2025.
[15] OpenAI API Documentation, https://platform.openai.com/docs/, accessed 2025.
[16] Google Gemini API Documentation, https://ai.google.dev/docs, accessed 2025.
[17] GitHub Models Documentation, https://docs.github.com/en/github-models, accessed 2025.
[18] M. Pezze and M. Young, "Software Testing and Analysis: Process, Principles, and Techniques," John Wiley and Sons, 2008.
[19] Deloitte, "Technology, Media, and Telecommunications Predictions 2020," Deloitte Insights, 2020.
[20] M. Jackson, "Aspects of Abstractions in Software Development," Software Engineering Journal, vol. 6, no. 2, pp. 54-62, 1991.