

CodeSage Docs: An AI-Driven Platform for Repository Intelligence, Code Review, and Documentation Automation

Ayush K

Malnad College of Engineering
Hassan, Karnataka
ayushk1205@gmail.com

Mrs. Nithyashree R

Malnad College of Engineering
Hassan, Karnataka
rn@mcehassan.ac.in

Brinda M S

Malnad College of Engineering
Hassan, Karnataka
Brindams10@gmail.com

Gagan Ravi Karjagi

Malnad College of Engineering
Hassan, Karnataka
karjagigagan13@gmail.com

Ganavi K V

Malnad College of Engineering
Hassan, Karnataka
ganvigowdaa18@gmail.com

Abstract—Most engineering teams today live inside platforms like GitHub or GitLab, where dozens of contributors push code from different time zones, often without much coordination. In any project of reasonable size, the repository never really sits still — commits accumulate, branches get created and collapsed, and Pull Requests arrive faster than anyone can comfortably read them.

Staying on top of documentation and code review under those conditions is harder than it looks. Developers are not negligent; they're simply busy shipping things. When something has to slip, it's almost always the documentation that goes first — and review thoroughness follows not far behind.

This paper presents CodeSage Docs, a system we built to take that burden off developers. It hooks into the standard Git workflow and uses Large Language Models to automatically interpret what changed, write commit summaries that actually make sense, keep documentation in step with the codebase, and give Pull Requests a first-pass review before a human reviewer even opens the diff.

The plumbing underneath connects GitHub Webhooks to an Express.js server, runs inference through OpenAI's API, and stores everything in MongoDB. The practical effect is less manual overhead, fewer things slipping through code review, and documentation that does not immediately become a lie the moment someone refactors a module.

Index Terms—Large Language Models, Code Review Automation, Documentation Generation, GitHub Webhooks, DevOps, Repository Intelligence

I. INTRODUCTION

Software development looks quite different than it did ten years ago. Continuous integration, distributed teams, and cloud-hosted repositories have collectively made it possible to ship faster than ever — but they have also made individual codebases significantly harder to follow. At any moment, several developers might be modifying overlapping parts of a system, and the history of why a particular file looks the way it does often lives only in someone's head.

Documentation has always lagged behind code, but the gap has grown. In an Agile environment where a sprint ends every two weeks, there is rarely a scheduled slot for “update the docs.” It happens when someone remembers, which means it frequently does not happen at all. Six months into a project, onboarding a new contributor requires either a long knowledge-transfer session or a lot of painful archaeology through commit history.

Code review has a related problem. The intent is good — every change should get a careful second look before it lands. The reality is that reviewers are often stretched across multiple PRs, some of them spanning hundreds of lines. Under those conditions, catching subtle logic errors or security oversights requires a level of attention that is genuinely difficult to sustain consistently.

What Large Language Models make newly possible is bridging both of these gaps at once. A model that can read a diff and explain what changed — in plain English, with relevant context — can serve as a tireless first reviewer and a documentation writer that never gets too busy to update its notes.

CodeSage Docs was built on exactly that idea. Rather than asking developers to do more, it automates the parts of the workflow that do not strictly require human judgment: summarising what changed, updating relevant documentation, and flagging the things in a PR that are worth a closer look.

II. PROBLEM STATEMENT

The core problem is not that developers are doing anything wrong. It's that the volume and pace of modern software development has outgrown the tools designed to support it.

Every commit can touch several files simultaneously. In a large codebase, even a change that looks contained can affect modules the author was not thinking about. Keeping a coherent picture of how the system works — not just right now, but as

it evolves — becomes increasingly difficult as the team and codebase grow.

Documentation is the most visible casualty. It tends to get written at the beginning of a project, when everyone is optimistic about maintaining it, and then gradually diverges from reality as the code moves on. New hires are handed a wiki that describes architecture decisions from eighteen months ago. Function signatures change and no one updates the corresponding documentation. This is not a discipline problem; it is a tooling problem. The feedback loop between code change and documentation update is manual and slow.

Manual PR review has its own failure modes. A reviewer looking at a 400-line diff has to reconstruct the author's intent, understand the downstream effects, and check for both correctness and style — all while managing their own workload. Bugs get through not because reviewers are bad at their jobs, but because human attention is finite and context-switching is expensive.

When teams are large and changes frequent, these failure modes compound. Two developers can unknowingly implement the same thing. A refactor in one module breaks an assumption in another module nobody remembers is coupled to it. These conflicts surface at integration time, when they are expensive and frustrating to untangle.

Most existing tools do not help enough. Static analysis catches formatting issues and obvious bugs. Linters enforce style. But none of them understand what a change was trying to do, which is exactly the question that matters most when reviewing code or writing documentation. Rule-based systems are not equipped to answer it.

The other structural problem is that the tools are not integrated. Code review lives in GitHub. Documentation lives in Confluence or Notion. Monitoring lives somewhere else. Developers constantly export context from one place and paste it into another. The overhead adds up, and the outputs are inconsistent because they were never designed to talk to each other.

CodeSage Docs was built to address all of this from a single system — one that understands what code changes actually mean and acts on that understanding automatically.

III. OBJECTIVES

The goals driving the design of this system were:

- Build a real-time monitoring layer that captures GitHub events — commits, PRs, merges, branch updates — without requiring any manual triggering.
- Generate commit summaries that are useful to both the engineer who wrote the code and a product manager who has never seen the repository.
- Keep technical documentation synchronized with code changes automatically, so that documentation drift becomes a problem of the past.
- Implement a PR review module that can detect logical errors, security vulnerabilities, and code quality issues before a human reviewer looks at the diff.

- Reduce the time developers spend on documentation and first-pass review — not by skipping those steps, but by automating the mechanical parts of them.
- Give teams a single, unified view of what is happening in their repository, rather than forcing them to stitch together information from five different tools.

IV. LITERATURE REVIEW

Building CodeSage Docs meant spending time understanding what already existed in four areas: automated code review, commit message generation, repository comprehension using LLMs, and AI-driven documentation. Here is what we found.

A. Automated Code Review and Defect Detection

Li et al. [1] trained a language model specifically for the code review task, showing that deep learning can identify defects and suggest improvements at a level that static analysis simply cannot match. The system worked well, but it was self-contained — it did not feed into documentation or connect to broader repository monitoring. Most tools in this category have the same limitation: they look at code in isolation rather than as part of a living system.

B. Commit Message Generation

Liu et al. [2] tackled the specific problem of commit messages, training a diff-aware transformer to read what changed and write a human-readable description. The results were impressive for single commits. The gap was that the system stopped there — it did not carry that understanding forward into documentation or provide any insight into how individual commits related to each other over time. That limitation directly shaped how we designed our commit analysis layer.

C. Repository Comprehension with LLMs

Gupta and Chen [3] explored Retrieval-Augmented Generation as a mechanism for answering questions about codebases. Combining code embeddings with external knowledge improved the quality of repository-level question answering quite a bit. The practical bottleneck was that RAG systems tend to require substantial preprocessing, and the approach was not optimized for real-time, webhook-driven environments where events arrive continuously and analysis needs to happen fast.

D. AI-Based Documentation Systems

Patel [4] built a system for continuous documentation generation in Agile environments, demonstrating that it is feasible to keep documentation updated automatically as code changes. Deepshikha et al. [?] and Venter et al. [?] both showed that feedback quality improves substantially when it is tailored to specific inputs rather than templated responses — a finding that directly influenced how our documentation and review outputs are constructed.

E. Research Gap

The pattern across all of this work is that each system solves one part of the problem well, but none of them connect the parts together. Code review does not talk to documentation. Commit summarization does not feed into longer-term repository understanding. Real-time webhook integration is almost entirely absent from the academic literature. Developers are left stitching together outputs from isolated tools, which is exactly the friction CodeSage Docs is designed to eliminate.

Table I summarizes the surveyed papers.

V. PROPOSED SYSTEM

The underlying idea behind CodeSage Docs is that repository intelligence should be a continuous process, not a checklist item. Every time code changes, something should happen: a summary gets written, documentation gets updated, the PR gets an initial review. None of that should require the developer to ask for it.

In practice, the flow looks like this. A developer pushes a commit or opens a Pull Request. GitHub sends a webhook to the CodeSage backend. The backend parses the event, pulls out the relevant diff and metadata, and hands everything to the AI processing layer. The LLM reads the diff, generates a commit summary, flags anything that looks suspicious, and triggers documentation updates for any affected sections. All of this lands in a dashboard that the team can check without switching tools.

What makes this different from existing solutions is the shared data model underneath. Commits, diffs, AI-generated summaries, documentation snapshots, and review logs all live in the same MongoDB instance, connected to each other. That means the system can answer questions that isolated tools cannot — for example, “how has this module’s documentation evolved over the last ten commits?” or “which PRs touched this function and what did the reviewer comments say?”

That integration is the thing we kept coming back to during design. Individual features are not hard to find; unified systems that actually talk to each other are much rarer.

VI. SYSTEM ARCHITECTURE

The architecture is organized in layers, each with a distinct responsibility. The goal was to keep things modular enough that any layer could be swapped out or upgraded without breaking the others.

Figure 2 contrasts what the development workflow looks like without CodeSage Docs versus with it. The left side — fragmented tools, stale documentation, overloaded reviewers — is the status quo for most teams. The right side shows what becomes possible when the same data flows through a single, connected system.

A. Frontend

The client is a React 18 application written in TypeScript. We chose TypeScript early on because prop-drilling bugs and undefined-at-runtime errors compound quickly in a dashboard

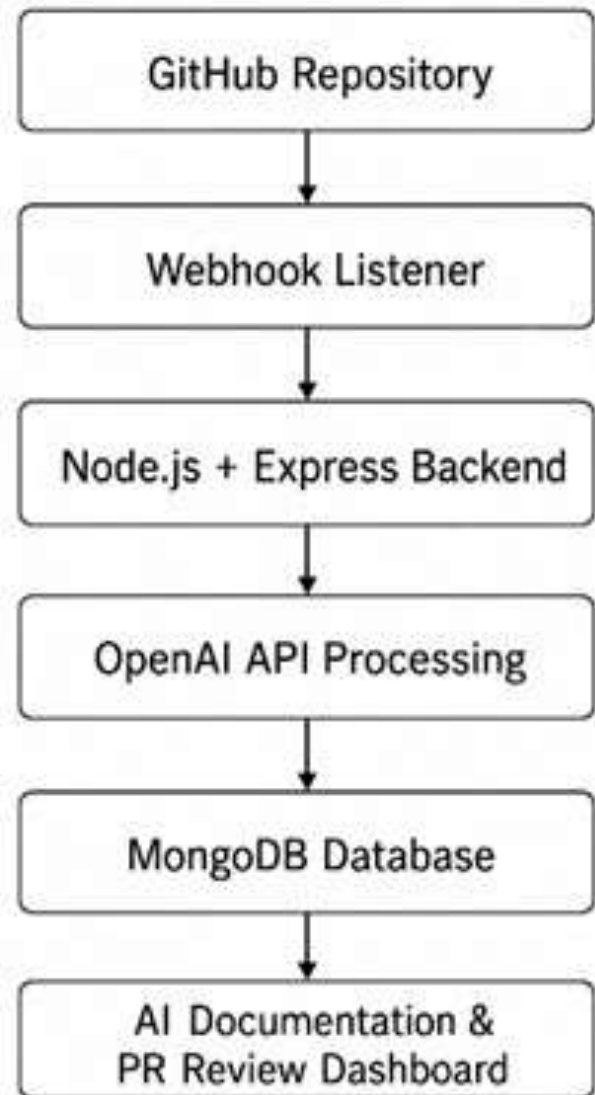


Fig. 1. System Architecture Dataflow.

that displays live, changing data, and having the type checker catch those at compile time was worth the overhead.

The interface supports three roles: developer, reviewer, and administrator. Each sees a different view of the repository data, scoped to what is relevant to them. Styling uses Tailwind CSS with shadcn/ui components, which kept the visual design consistent without requiring a custom component library from scratch.

State management for server data is handled with React Query. The main reason for this choice was background re-fetching: repository data changes continuously, and React Query’s stale-while-revalidate pattern means developers are always looking at recent information without manually hitting refresh.

TABLE I
SUMMARY OF LITERATURE SURVEY

#	Paper Title	Authors	Year	Venue	Key Contribution
1	Code Reviewer: Pre-Training for Automating Code Review Activities	Y. Li et al.	2023	IEEE TSE	AI-based automated code review and defect detection
2	Auto Commit: Adaptive Commit Message Generation via Diff-Aware Transformer Models	S. Liu et al.	2024	J. Automated Software Eng.	Automatic commit message generation from code diffs
3	RAG-based Software Repository Comprehension: Bridging the Gap Between Code and Context	A. Gupta & M. Chen	2025	ICSE	Repository understanding using Retrieval-Augmented Generation
4	Continuous Documentation: Automating Technical Writing in Agile Environments	K. Patel	2024	IC Technical Communication	Automated documentation generation in agile workflows
5	Semantic Code Diffing for Enhanced AI-Driven Code Reviews	B. Wang & H. Lee	2025	ACM SIGSOFT	Semantic analysis of code changes for better reviews
6	Dev Agents: Autonomous DevOps Teams and Multi-Agent Orchestration	J. Doe et al.	2025	AI & DevOps Workshop	Multi-agent automation for DevOps workflows
7	Vulnerability Detection in CI/CD Pipelines: A Large Language Model Approach	R. Verma	2024	DevSecOps Journal	AI-based security vulnerability detection in pipelines
8	Automated Commit Message Generation Using Deep Learning Models	H. Tan et al.	2023	IEEE Access	NLP-based commit summarization system
9	Large Language Models for Source Code Understanding and Generation	OpenAI Research	2023	ArXiv	Demonstrates LLM capability in code reasoning
10	GitHub Webhook Driven Event Processing Systems for DevOps Automation	M. Smith et al.	2022	Cloud Computing Conf.	Real-time repository event processing architecture
11	AI-Assisted Software Engineering: A Survey of Intelligent Code Tools	Y. Zhang et al.	2025	IEEE Computing Surveys	Overview of AI tools for code review and documentation

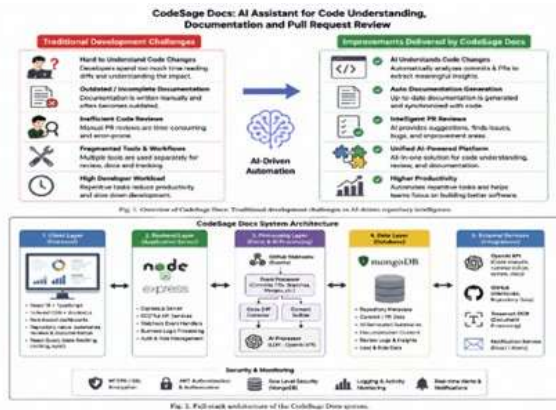


Fig. 2. Traditional Development Challenges vs. AI-Driven Repository Intelligence with CodeSage Docs.

B. Backend

The backend is an Express.js server configured to receive GitHub Webhook payloads. When an event arrives, the server validates the payload signature, parses the relevant fields — diffs, author metadata, file paths, branch names — and normalizes everything into a structured object before any AI processing happens.

MongoDB stores the results. We used MongoDB rather than a relational database because the data shapes vary significantly:

a commit summary looks nothing like a documentation snapshot, and a review log has different fields than a repository metadata record. The document model handled that heterogeneity cleanly without requiring schema migrations every time we added a new output type.

The event-driven architecture means there is no polling. When something happens in the repository, the webhook fires and processing begins. By the time a developer opens the dashboard, the summary for their latest push is usually already there.

C. External Integrations

Three external services handle the pieces the system does not do internally:

- **OpenAI API** — Semantic understanding of diffs, commit summarization, PR review generation, and documentation drafting all go through this.
- **GitHub Webhooks** — Real-time event delivery for commits, merges, PR creation, and branch updates.
- **MongoDB** — Persistent storage for repository metadata, AI-generated outputs, and historical analysis logs.

VII. IMPLEMENTATION DETAILS

The frontend was scaffolded with Vite, primarily for the fast hot-module replacement during development and the relatively clean production build output. React 18 with TypeScript handles the component layer. shadcn/ui gave us accessible,

composable UI primitives without locking us into a design system we could not customize.

React Query does more than just fetch data. It caches responses, retries failed requests with exponential backoff, and marks data as stale in the background so that the next user interaction always triggers a fresh fetch if enough time has passed. For a dashboard that needs to reflect live repository state, this behavior is exactly right.

On the backend, every webhook payload goes through a normalization step before anything else happens. Raw GitHub payloads are inconsistent in structure across event types, and we needed a clean internal representation regardless of whether the event was a push, a PR open, or a merge. That normalized object is what gets handed to the AI pipeline.

MongoDB collections are organized around the core entities: repositories, commits, pull requests, AI-generated summaries, documentation snapshots, and review logs. The schema is deliberately loose on the AI output fields, since the model response format can evolve, but strict on the metadata fields that drive queries and lookups.

Access control is enforced at the backend. Developers can only query repositories they have been assigned to, and sensitive operations — webhook processing, API calls to OpenAI — are never exposed through the client. All of that happens server-side.

The AI pipeline itself works like this: the normalized event data gets assembled into a prompt that includes the diff, relevant surrounding context pulled from MongoDB, and a concise instruction. The model returns a response that the backend validates against a predefined schema before storing. Malformed responses are logged and retried rather than silently dropped or stored as-is, which matters for documentation quality.

The documentation generation pipeline runs whenever a commit touches a file that has a corresponding documentation record. Instead of regenerating everything from scratch each time, the system identifies which sections are affected by the diff and updates only those. This keeps documentation changes focused and avoids overwriting sections that were not touched.

The PR review module operates separately. When a Pull Request arrives, the module pulls the full diff, builds a prompt that includes repository context and any relevant prior review comments, and asks the model to identify potential issues: logic errors, missing edge cases, duplicated logic, security concerns. The output is formatted as structured review comments displayed alongside the diff in the dashboard.

VIII. COMPARISON WITH EXISTING SYSTEMS

Table II positions CodeSage Docs against the main categories of tools teams use today. The point is not that other tools are bad — most of them do their specific job reasonably well. The problem is that they are separate, and the cost of running three separate systems is higher than it first appears. Traditional code review tools cover the PR workflow but have no understanding of code semantics and nothing to say about documentation. Documentation tools are entirely disconnected from the development event stream — they sit

and wait for someone to edit them manually. AI-assisted development platforms (Copilot, CodeWhisperer, and similar) help at the individual developer level but do not provide repository-level intelligence or documentation output.

CodeSage Docs is the only system in this comparison that does all of: real-time monitoring, semantic analysis, documentation generation, and integrated PR review from a single platform.

TABLE II
FEATURE COMPARISON WITH EXISTING SYSTEMS

Feature	Trad. Review	Doc Tools	AI Assist	CodeSage
Code Understanding	Partial	No	Partial	Yes
PR Review	Yes	No	Yes	Yes
Commit Summarization	No	No	Partial	Yes
AI Doc Generation	No	No	Partial	Yes
Repository Monitoring	No	No	No	Yes
Semantic Analysis	No	No	Partial	Yes
Integrated Platform	No	No	No	Yes

IX. ADVANTAGES OF THE PROPOSED SYSTEM

Compared to the fragmented status quo, CodeSage Docs offers several concrete improvements:

- Commit summaries are generated automatically, so the history of a repository actually tells you something useful rather than being a wall of “fix bug” and “update code” messages.
- Documentation stays current. Not perfectly current — the model is not infallible — but significantly more current than documentation that is only updated when someone remembers to do it.
- PR reviews arrive before the human reviewer opens the diff, which means reviewers can focus on the things the model flagged rather than reading the entire change from scratch.
- Everything is in one place. No context-switching between the code review tool, the documentation wiki, and the analytics dashboard.
- The system scales with repository size. Processing is event-driven, so a repository with 1000 commits per day is handled the same way as one with 10 — the pipeline just runs more often.
- Decision-making during PR approvals is better because reviewers have structured context, not just a raw diff.

X. FUTURE SCOPE

The current system handles a single repository at a time. The most obvious extension is multi-repository intelligence: understanding relationships between repositories in the same organization, tracking shared module usage, and surfacing cross-project dependency risks. For teams managing microservices or monorepos with multiple packages, that visibility would be genuinely valuable.

We also want to add natural language code search. The idea is straightforward: a developer types “where does the payment service handle retries?” and gets a direct answer with file references, rather than having to grep through thousands of files. The retrieval infrastructure is already partially in place; it needs a proper query interface.

Deeper vulnerability detection is another priority. The current PR review flags obvious security issues as part of the general review, but a dedicated security analysis pass — looking specifically for injection risks, authentication gaps, and insecure API patterns — would add real value without much architectural change.

On the collaboration side, tracking contributor patterns over time (who reviews whose code, who tends to modify the same files, where knowledge is siloed) could generate useful signals about team health and bus-factor risks.

Longer term, a conversational interface over the repository — something like asking a chatbot to explain a function or walk you through a module — would close the loop between documentation and interactive exploration.

Finally, tighter CI/CD integration: connecting CodeSage analysis to GitHub Actions pipelines so that deployment-aware insights are available before code ships, not just after.

XI. CONCLUSION

Modern codebases move fast and have too many moving parts for manual documentation and review to keep up reliably. That is not a failing of any particular team; it is just what happens when development velocity outpaces the tooling designed to support it.

CodeSage Docs is our attempt to close that gap. Connecting GitHub’s event stream to an LLM-powered processing pipeline through an Express.js backend, with MongoDB holding the results, gives teams a system that genuinely tracks what is happening in their repository and does something useful with that information automatically.

In practice, the automated commit summaries made codebases easier to navigate. Documentation stopped being a manual chore. PR reviews caught things earlier. None of this is magic — the model makes mistakes, and a human reviewer is still essential for anything consequential. But having a reliable first pass, and having documentation that does not immediately drift from reality, noticeably reduces the friction of working in a large codebase.

There is still plenty to build. Multi-repository support, natural language search, deeper security analysis, and a conversational interface are all on the roadmap. The modular architecture was designed with that in mind, so adding them will not require rethinking the foundation.

The broader point CodeSage Docs demonstrates is that AI is most effective when it is woven into the workflow rather than sitting next to it. When understanding a codebase is a continuous automated process rather than a periodic manual effort, teams can spend less time explaining what they built and more time building it.

REFERENCES

- [1] Y. Li et al., “Code Reviewer: Pre-Training for Automating Code Review Activities,” *IEEE Transactions on Software Engineering*, 2023.
- [2] S. Liu et al., “Auto Commit: Diff-Aware Transformer for Commit Message Generation,” *Journal of Automated Software Engineering*, 2024.
- [3] A. Gupta and M. Chen, “RAG-Based Software Repository Comprehension,” *ICSE Proceedings*, 2025.
- [4] K. Patel, “Continuous Documentation Systems in Agile Development,” *Software Engineering Journal*, 2024.
- [5] B. Wang and H. Lee, “Semantic Code Diffing for AI-Based Code Review,” *ACM SIGSOFT*, 2025.
- [6] J. Doe et al., “Dev Agents: Autonomous DevOps Systems,” *AI for Software Engineering Workshop*, 2025.
- [7] R. Verma, “Vulnerability Detection in CI/CD Pipelines Using LLMs,” *DevSecOps Journal*, 2024.
- [8] H. Tan et al., “Deep Learning for Commit Message Generation,” *IEEE Access*, 2023.
- [9] OpenAI Research, “Large Language Models for Code Understanding,” 2023.
- [10] M. Smith et al., “GitHub Webhook-Based Event Processing Systems,” *Cloud Computing Conference*, 2022.
- [11] Y. Zhang et al., “AI-Assisted Software Engineering Survey,” *IEEE Computing Surveys*, 2025.