Title

Explicit Context-Driven Development: From Vibe Coding to Verifiable Workflows in AI-Augmented Software Engineering

Abstract

AI-assisted programming tools can dramatically accelerate software development, but informal "vibe coding"—rapidly iterating with Large Language Models (LLMs) without explicit guidance—often leads to fragile architectures, context drift, and poor traceability. We argue that effective use of AI in software projects requires treating context and prompts as first-class, version-controlled artifacts. To this end, we propose Explicit Context-Driven Development (ECDD), a methodology that formalizes the interaction between human developers and AI agents. ECDD is built on three pillars: (i) Prompt-Driven AI Governance, (ii) Auditable Task & State Management, and (iii) a structured Five-Phase Lifecycle (Define, Plan, Elaborate, Scope, Implement). In this model, **executable prompts** guide the AI to interactively interview the user and generate **persistent context artifacts** (e.g., `project_definition.md`, `workpackage_list.md`, `copilot-instructions.md`). This decouples project state from the ephemeral chat window, ensuring that architectural decisions and coding standards are rigorously maintained and automatically injected into the AI's context throughout the software development lifecycle.

Keywords

Explicit Context-Driven Development (ECDD), AI-Augmented Software Engineering, Vibe Coding, Prompt Engineering, Context Management, Human-in-the-Loop, Software Development Lifecycle (SDLC), Verifiable Workflows.

I. INTRODUCTION

AI coding tools hold great promise but also introduce new risks. Unlike traditional development environments, modern large language models (LLMs) can generate entire features from high-level prompts. This makes it tempting for developers to "throw ideas at the AI" and iterate informally—a practice often described as *vibe coding*. While this style can be enjoyable for exploration and rapid prototyping, it tends to produce incoherent architectures, inconsistent conventions, and hidden defects once projects grow beyond small experiments.

In contrast, emerging best practices emphasize making project goals, constraints, and conventions explicit so that the AI assistant behaves like a disciplined collaborator rather than an improvising partner. Instead of relying on whatever context happens to be in the editor window or chat history, developers are encouraged to treat specifications and context as version-controlled artifacts—"version control for your thinking"—that can be inspected, diffed, and reviewed alongside code. In this view, prompts are not ad hoc instructions but part of the project's technical documentation.

We propose **Explicit Context-Driven Development (ECDD)** as a methodology that puts this principle at the center of AI-augmented software development. In ECDD, project context is not just static documentation but is actively managed through a structured **Five-Phase Lifecycle**: *Define, Plan, Elaborate, Scope, and Implement.*

In this methodology: 1. **Prompt-Driven AI Governance** is enforced by executable prompts (e.g., `define.prompt.md`, `scope.prompt.md`) that interactively interview the user to generate canonical project definitions (`project_definition.md`) and AI instructions (`copilot-instructions.md`). 2. **Auditable Task Management** is handled through persistent roadmap files (`workpackage_list.md`) and detailed specifications (`workpackage_WP-XXX.md`) rather than ephemeral chat to-do lists. 3. **State Management** is maintained via an append-only log (`log.md`) that tracks the evolution of the codebase, ensuring that the AI "remembers" previous implementation steps.

The aim is to preserve human oversight—"Copilot, not autopilot"—while still obtaining the speed benefits of AI assistance.

This paper makes three contributions. First, we define ECDD as a concrete methodology for AI-augmented development, grounded in the idea of explicit, version-controlled context artifacts. Second, we describe the reusable **Five-Phase Workflow** and the corresponding prompt templates that operationalize ECDD. Third, we demonstrate how this approach decouples the project state from the specific AI tool, making the workflow IDE-agnostic (compatible with GitHub Copilot, Cursor, Windsurf, etc.).

The remainder of the paper is organized as follows. Section II introduces Explicit Context-Driven Development and its design principles at a high level. Section III details how each ECDD pillar can be instantiated as a concrete workflow with canonical prompt templates and logging structures. Section IV discusses lessons learned, limitations, and how ECDD relates to existing work on AI-assisted development workflows. Section V concludes and sketches future research directions.

II. EXPLICIT CONTEXT-DRIVEN DEVELOPMENT (ECDD)

ECDD is a methodology for AI-augmented software development that treats prompts and contextual information as first-class artifacts. Rather than relying on whatever code happens to be visible in the editor or whatever has recently been discussed in chat, ECDD insists that the essential context for a project be written down, structured, and version-controlled. This context includes project goals, architectural decisions, domain models, coding standards, and risk boundaries for AI usage. By making all of this explicit and durable, ECDD aims to turn AI assistance from an improvisational helper into a predictable component of the engineering process.

A. Three Pillars of ECDD

ECDD is organized into three pillars that structure the collaboration between human and AI:

1. **Prompt-Driven AI Governance**: Project-wide instructions and conventions are captured in persistent prompt files and generated context artifacts. Specifically, the `copilot-instructions.md` file acts as a governance layer, automatically injecting architectural constraints and coding standards into every AI interaction. This file is not written manually but is generated by the **Scope** phase to ensure it aligns with the project definition.
2. **Auditable Task and State Management**: Work is decomposed into tasks and tracked using structured plain-text artifacts. The `workpackage_list.md` serves as the roadmap, while `log.md` provides an append-only history of implementation steps. These artifacts record what was planned, what was done, and why, enabling later auditing of AI-generated changes and preventing the AI from repeating past mistakes.
3. **The Five-Phase Lifecycle**: Unlike simple "Plan-Execute" loops, ECDD defines a comprehensive lifecycle that guides a feature from concept to code.
    - **Define**: Establish scope and architecture (`project_definition.md`).
    - **Plan**: Decompose work into a roadmap (`workpackage_list.md`).
    - **Elaborate**: Create detailed specifications for a single task (`workpackage_WP-XXX.md`).
    - **Scope**: Generate the AI context instructions (`copilot-instructions.md`).
    - **Implement**: Execute the code changes with a granular todo list (`todos_WP-XXX.md`) and update the log.

This structured lifecycle ensures that the AI has the necessary context at each stage—high-level goals during planning, and low-level technical specs during implementation.

III. ECDD PILLARS IN PRACTICE

In this section, we detail how the three pillars of ECDD are instantiated as a concrete workflow. We describe the specific prompts and templates used in each phase to transform "vibe coding" into a verifiable engineering process. Due to space constraints, we provide high-level descriptions of the artifacts; the full repository containing all executable prompts and templates is available at https://github.com/kmkarakaya/ECDD.

A. Prompt-Driven AI Governance

The foundation of ECDD is the **Define** and **Scope** phases, which establish the governance layer. Instead of manually writing a static `CONTRIBUTING.md`, the developer runs the `define.prompt.md`.

1. **Interactive Definition**: The `define` prompt acts as an expert Product Owner. It interviews the user, asking targeted questions about the project's goals, tech stack, and constraints. This prevents the "blank page problem" and ensures critical architectural decisions are made explicitly.
2. **Artifact Generation**: The output is a structured `project_definition.md` file. This file becomes the "source of truth" for all subsequent AI interactions.
3. **Context Scoping**: Once the definition is stable, the `scope.prompt.md` reads it and generates `.github/copilot-instructions.md`. This file translates the high-level definition into specific behavioral instructions for the AI coding assistant (e.g., "Always use TypeScript interfaces," "Follow the repository pattern").

By automating the creation of these governance artifacts, ECDD ensures that the AI's "system 2" thinking (planning and architecture) is captured and enforced during "system 1" execution (coding).

B. Auditable Task & State Management

To combat context drift, ECDD replaces ephemeral chat history with persistent roadmap artifacts during the **Plan** and **Elaborate** phases.

1. **Roadmap Planning**: The `plan.prompt.md` reads the project definition and decomposes it into a list of Work Packages (WPs) in `workpackage_list.md`. Each WP has a clear ID, priority, and dependency list.
2. **Detailed Elaboration**: Before writing any code, the developer selects a specific WP (e.g., WP-001) and runs `elaborate.prompt.md`. The AI expands the high-level task into a detailed specification (`workpackage_WP-001.md`), complete with acceptance criteria, data models, and API contracts.
3. **Human Review**: Crucially, these artifacts are reviewed *before* implementation. The developer can edit the specification to correct misunderstandings or refine the architecture, ensuring the AI is aligned with human intent before a single line of code is written.

C. The Plan-Execute-Verify Loop (Implementation)

The **Implement** phase closes the loop, turning specifications into code while maintaining a clear audit trail.

1. **Granular Planning**: The `implement.prompt.md` reads the detailed WP

specification and generates a granular checklist in `todos_WP-XXX.md`. This breaks the feature down into atomic steps (e.g., “Create database schema,” “Implement API endpoint,” “Write unit test”).
2. **Iterative Execution**: The AI (or developer) executes these steps one by one. Unlike “vibe coding,” where the AI might generate a massive, unverified code block, ECDD enforces a step-by-step approach.
3. **State Logging**: As work progresses, the `log.md` file is updated with an append-only record of changes, decisions, and file modifications. This log serves as the project’s long-term memory, allowing the AI to resume work context-aware even after the chat session ends.

This rigorous process ensures that every line of code is traceable to a specific requirement and that the project’s architectural integrity is maintained over time.

IV. DISCUSSION AND LIMITATIONS

ECDD is intentionally conservative in how it uses AI: it assumes that human developers remain responsible for understanding the system and that prompts and plans are part of the system’s design, not just transient instructions. This stance brings several benefits compared to informal “vibe coding.” First, explicit governance prompts anchor the behavior of AI assistants in the project’s actual goals and constraints rather than in whatever files happen to be visible. Second, auditable task and state logs make it easier to reconstruct why a change was made and how AI contributed to it, which is particularly important when debugging regressions or performing post-mortems. Third, the Five-Phase Lifecycle encourages small, reviewable increments instead of large, opaque bursts of AI-generated code.

At the same time, ECDD introduces overhead. Writing and maintaining governance prompts, plans, and logs takes time, especially at the beginning of a project when requirements are fluid. Teams that are used to highly informal workflows may initially perceive ECDD as “too much process” for small experiments. In our own use, we found that the overhead is most defensible when at least one of the following conditions holds: the project is expected to live longer than a few weeks, multiple developers or agents will contribute to the same codebase, or the domain carries non-trivial risk (such as privacy, safety, or regulatory concerns). For tiny throwaway prototypes, a lighter-weight subset of the methodology may be more appropriate.

Another trade-off is the risk of “prompt bloat.” If every architectural nuance and process rule is encoded into increasingly long prompt files, AI assistants and humans alike may struggle to extract the relevant parts. ECDD therefore works best when governance prompts are treated as curated documents rather than dumping grounds: they should be periodically refactored, pruned, and restructured, much like code. In practice, this means removing obsolete sections, splitting large documents into focused ones, and keeping examples aligned with the current state of the system.

There is also a risk of false confidence. The existence of explicit plans, logs, and tests does not by itself guarantee correctness or robustness; it only makes reasoning and auditing easier. Poorly written plans, misaligned governance prompts, or superficial tests can still lead to subtle failures, especially in domains where AI-generated content interacts with complex external systems. ECDD mitigates this only indirectly, by making issues more visible and reviewable; it does not replace careful engineering judgment.

From a tooling perspective, ECDD currently assumes that teams are comfortable working with plain-text artifacts in version control and that they can integrate these with their preferred AI tools. In our experience, simple repository conventions (such as placing prompts under `.github/prompts/`) are sufficient for many setups, but richer integrations—e.g., IDE plugins that surface relevant prompt files automatically—would make adoption smoother. Exploring how ECDD interacts with existing DevOps pipelines, issue trackers, and documentation systems is an important direction for future tooling work.

A. Relation to Existing Work

ECDD sits at the intersection of several threads of work on software development practice. From one angle, it can be seen as a pragmatic descendant of specification-driven and model-driven development, but with LLMs in the loop and with a stronger emphasis on treating prompts themselves as artifacts. Like configuration-as-code and "infrastructure as code" approaches, it insists that important project knowledge live in version-controlled text files rather than in wikis, tickets, or team folklore.

Compared to traditional style guides and architecture decision records (ADRs), ECDD's governance prompts are more tightly coupled to AI tools: they are written to be consumed both by humans and by LLMs, and they are referenced explicitly in planning, execution, and verification prompts. This dual audience influences their structure and language, pushing teams toward more precise but still natural-language descriptions of goals, constraints, and patterns. There are also parallels with literate programming and notebook-based workflows, in that ECDD attempts to keep "what we are doing" and "why we are doing it" close to the code, but in a form that AI tools can act on directly.

Existing proposals for AI-assisted development pipelines often focus on end-to-end agents that plan, code, and test with minimal human intervention. ECDD deliberately chooses a different point in the design space: it assumes human-in-the-loop control and asks how to make the human-AI collaboration itself more structured, observable, and reviewable. In that sense, it is closer to "copilot" visions of AI assistance than to fully autonomous agents, while still borrowing useful ideas from agentic architectures such as explicit planning and tool orchestration.

V. CONCLUSION AND FUTURE WORK

We have introduced Explicit Context-Driven Development (ECDD), a methodology for AI-augmented software development that treats prompts and contextual information as first-class, version-controlled artifacts. ECDD is structured around three pillars: Prompt-Driven AI Governance, Auditable Task & State Management, and a rigorous Five-Phase Lifecycle (Define, Plan, Elaborate, Scope, Implement). Together, these pillars aim to retain the speed and flexibility of AI-assisted coding while improving traceability, consistency, and human oversight compared to informal "vibe coding."

To make ECDD actionable, we described concrete workflows and canonical prompt templates for each phase. Governance is enforced through the interactive creation of `project_definition.md` and `copilot-instructions.md`. Task management is handled via `workpackage_list.md` and detailed specifications. The implementation loop turns LLM interactions into repeatable micro-workflows that can be inspected and evolved over time. Although our examples draw on general software patterns, the templates are designed to be reusable across different domains and technology stacks. We maintain the full prompt sets and example repositories online so that others can study, adapt, and extend them.

There are several avenues for future work. First, we plan to conduct more systematic evaluations of ECDD in real teams, including controlled studies comparing ECDD-style workflows to baseline AI usage patterns in terms of defect rates, development velocity, and onboarding time. Second, we aim to explore richer tool support: for example, IDE extensions that automatically surface relevant governance prompts, visualizations of the roadmap progress, and integrations with issue trackers and CI pipelines. Third, we would like to investigate how ECDD interacts with other safety and assurance techniques, such as static analysis, formal specifications, and red-teaming of AI-generated changes, particularly in safety-critical or regulated domains.

Finally, as LLMs and tooling evolve, ECDD itself will need to adapt. More capable models with larger context windows may reduce some of the friction in understanding large codebases, but they will not eliminate the need for explicit context, shared conventions, and auditable processes. We see ECDD not as a fixed recipe but as a framing: a reminder that prompts, plans, and logs are part of the software system and deserve the same care, review, and iteration as the code they help produce.