From Vibe Coding to Verifiable Workflows: A Structured Methodology for AI-Augmented Software Development

1st Murat Karakaya
Department of Software Engineering
TED University

Ankara, Türkiye
murat.karakaya@tedu.edu.tr

***Abstract*— AI-assisted programming tools can dramatically accelerate software development, but informal vibe coding—rapidly iterating with a large language model (LLM) without explicit guidance—often leads to fragile architectures, inconsistent conventions, and poor traceability at scale. We argue that effective use of AI in software projects requires treating context and prompts as first-class artefacts rather than transient chat messages. To this end, we propose Explicit Context-Driven Development (ECDD), a methodology for AI-augmented software development built on three pillars: (i) prompt-driven AI governance, in which project-wide goals, architecture, and coding standards are captured in version-controlled prompt files; (ii) auditable task and state management, where work packages and outcomes are tracked through structured backlog and status prompts; and (iii) a Plan–Execute–Verify loop that couples LLM-based planning with supervised implementation and explicit verification. Instead of presenting a single monolithic case study, we specify reusable workflows and canonical prompt templates for each pillar, with a small Tango Song Explainer web application used only as a running example. These workflows turn prompts, plans, and logs into inspectable development artefacts that can be reviewed and evolved alongside code. We conclude by discussing limitations, trade-offs, and opportunities for tool support and empirical evaluation of ECDD in real teams.**

Keywords*— *AI-assisted programming, software development workflows, prompt engineering, verifiable AI 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 emphasise 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 artefacts — "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 centre of AI-augmented software development. In ECDD, project context (architecture, domain concepts, API contracts, coding standards) is captured in dedicated prompt files that are checked into version control. Work is broken down into explicit tasks, tracked via structured to-do and status logs, and each task follows a Plan–Execute–Verify loop: plan the approach with an LLM, execute by coding (often via a coding agent or Copilot-style assistant), and verify through tests or review. 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. Second, we describe reusable workflows and prompt templates for each of ECDD's three pillars: prompt-driven AI governance, auditable task and state

management, and the Plan–Execute–Verify loop. Instead of a single monolithic case study, we provide pillar-specific patterns that can be adapted across projects, with a web app serving as a running example. Third, we discuss practical limitations and trade-offs observed in applying ECDD in practice, and outline directions for future work, including quantitative evaluations and comparisons to existing specification- and workflow-oriented tools.

The remainder of the paper is organised 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 artefacts. 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

At its core, ECDD assumes that every AI interaction in a project sits within an explicit workflow. The workflow describes (a) what the AI is supposed to do, (b) which artefacts define the current context, and (c) how the outcome will be verified. In practice, this means that LLM prompts are not just typed into a chat box; they are captured in files with clear roles (e.g., project instructions, planning prompts, implementation prompts, verification prompts) and maintained alongside the codebase. Developers can then review changes to these prompts as part of code review, treating prompt evolution as part of the system's design history.

ECDD is organized into three pillars:

- Prompt-Driven AI Governance. Project-wide instructions and conventions are captured in persistent prompt files (for example, a copilot-instructions file and additional .prompt.md files for architecture, coding standards, and risk boundaries). These files form a governance layer that constrains how AI tools behave across the project.
- Auditable Task and State Management. Work is decomposed into tasks and tracked using structured prompts and logs (such as todo and status prompt files). These artefacts record what was planned, what was done, and why, enabling later auditing of AI-generated changes.
- Plan–Execute–Verify Loop. Each non-trivial change follows a three-step loop: plan the change with an LLM, execute the plan using AI-assisted coding tools, and verify the result via tests, review, or additional analysing prompts. Plans and verification results are themselves stored in version control, closing the loop.

TABLE I. ECDD PILLARS AND CORE ARTEFACTS

Pillar	Artefacts
Prompt-Driven AI Governance	• copilot-instructions.md (project-wide instructions for AI assistants) • architecture.prompt.md (system overview and constraints) • coding-standards.prompt.md (style and quality rules)
Auditable Task & State Management	• todo.prompt.md (structured backlog of work packages) • status.prompt.md (append-only status and decision log)
Plan–Execute–Verify Loop	• Planning prompts (feature plans and risk analysis) • Execution prompts (implementation guidance) • Verification prompts (review and test prompts)

B. From Vibe Coding to ECDD in Practice

Informal vibe coding typically starts from a blank editor or chat window: the developer describes a feature in natural language, receives code from an LLM, and iterates ad hoc. Context is mostly implicit, carried in the current files and chat history, and decisions are rarely recorded systematically. This can work for very small experiments, but as the codebase grows it becomes increasingly difficult to explain why certain designs were chosen, how changes relate to each other, or which prompts produced which parts of the system.

Under ECDD, the same developer begins by establishing a minimal set of governance prompts (copilot-instructions, architecture.prompt, coding-standards.prompt) that encode project goals, constraints, and conventions. New work then flows through three explicit steps. First, the developer defines or updates work packages in todo.prompt, making the planned changes and their acceptance criteria visible to both humans and AI tools. Second, for each work package, a short plan is produced via a planning prompt and executed with the help of a coding assistant, following the Plan–Execute–Verify loop. Third, the outcome is summarised in status.prompt, linking the change back to the relevant work packages and plans. Throughout this process, humans remain in the loop: they approve and refine governance prompts, choose which plans to execute, and decide whether verification results are acceptable.

The result is not a fundamentally different set of capabilities—developers can still "ask the AI to build a feature"—but a different way of structuring those interactions. Instead of ephemeral chat sessions, ECDD turns prompts, plans, and logs into durable artefacts that can be reviewed, diffed, and reused, providing a clearer alternative to unstructured vibe coding.

In the rest of the paper we instantiate these pillars as concrete workflows. For each pillar we specify canonical prompt templates, expected input–output structures, and the surrounding developer actions. Although our examples reference a toy application, the workflows are designed to be

reusable across projects of different domains and technology stacks, with full prompt sets provided in an online repository.

III. ECDD Pillars in Practice

While ECDD's three pillars capture high-level principles, teams need concrete workflows to apply them consistently in day-to-day development. In this section we instantiate each pillar as a set of artefacts (prompt files and logs) and describe how developers and LLMs interact around them. For each pillar we also provide canonical prompt templates that we have found useful in practice. Due to page limits, a paper can usually show only excerpts; in our own projects we maintain the full prompt sets as version-controlled files in the repository and share them in an accompanying online archive.

Table I summarizes how the three pillars map to concrete artefacts. Prompt files are stored in the repository (typically under .github/ or a prompts/ directory) and are treated as first-class project assets: they are reviewed, versioned, and evolved alongside the source code.

In the remainder of this section we outline the workflow for each pillar and provide representative prompt templates. The examples assume a web-application project, but the patterns are intended to be reusable across domains.

A. *Prompt-Driven AI Governance*

The goal of prompt-driven AI governance is to align AI tools with the project's intent and constraints. Instead of letting Copilot or a coding agent infer everything from scattered files, we maintain a small set of governance prompts:

- copilot-instructions.md defines how AI assistants should behave in this repository.
- architecture.prompt.md captures the high-level system shape: components, data flows, external services, and non-functional requirements.
- coding-standards.prompt.md codifies naming, formatting, testing expectations, and quality rules.

Workflow.

1. Bootstrap. At project creation time, the team drafts minimal versions of these files based on the initial goals and tech stack.
2. Evolve. Whenever a significant architectural or process decision is made (new framework, new security rule, change in error-handling strategy), the relevant prompt file is updated in the same commit or pull request.
3. Use. Developers keep these files open or reference them when interacting with AI tools; agents that read the repository treat them as part of the context. Code reviewers check not only code changes but also updates to governance prompts.

The prompts are written to the AI, using natural language and structured sections. Below we show canonical templates that can be adapted per project. Due to space constraints, we only show excerpts of these templates; the full and evolving versions are available in our online repository [URL].

Aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa

IV. Canonical template: `copilot-instructions.md`

This file is the primary control surface for AI assistants integrated into the IDE or repository.

```
# Copilot Instructions for This
Repository

You are an AI pair programmer and code
generation assistant working on this
repository.

Your primary goals are:
- Help the team implement and maintain
this project safely and consistently.
- Respect the architecture, domain
model, and coding standards defined in
this repository.
- Prefer clarity and correctness over
brevity or cleverness.

## Project Context

- Project name: <PROJECT_NAME>
- High-level description: <ONE-PARAGRAPH
SUMMARY OF WHAT THE SYSTEM DOES>
- Main tech stack:
  - Frontend: <e.g., React + TypeScript
+ <UI framework>>
  - Backend: <e.g., Node.js / Python /
Firebase / etc.>
  - Data storage: <e.g., PostgreSQL,
Firestore, etc.>
- Target users and key scenarios:
  - <Scenario 1>
  - <Scenario 2>
  - <Scenario 3>

Whenever you propose changes, make sure
they fit this context.

## Architecture and Boundaries

- Follow the architecture described in
`architecture.prompt.md`.
- Do not introduce new services,
frameworks, or major dependencies
without the user explicitly asking for
it.
- Keep features small and incremental;
avoid "big bang" refactors unless
explicitly requested.
- Preserve public APIs and contracts
unless the user explicitly agrees to
change them.

If you are unsure about an architectural
decision, ask a clarifying question and
suggest options instead of guessing.

## Coding Standards
```

```
Follow the coding standards described in
`coding-standards.prompt.md`. In
particular:

- Use the established naming conventions
for files, components, functions, and
variables.
- Match the existing code style in this
repository (formatting, imports, error
handling).
- Always include or update tests when
you add or change non-trivial behaviour.

If you see code that violates these
standards, gently suggest improvements
when appropriate.

## Interaction Guidelines

- Explain your reasoning briefly when
proposing non-trivial changes.
- When the user asks for a feature,
clarify edge cases and constraints
before generating large amounts of code.
- Prefer modifying existing functions
over duplicating logic.
- When editing, keep diffs minimal and
well-scoped.

## Safety and Privacy

- Do not invent APIs, endpoints, or data
fields that are not present in the code
or explicitly described by the user.
- Do not hard-code secrets, tokens, or
credentials. Use environment variables
or configuration files as appropriate.
- If a requested change might introduce
security, privacy, or performance risks,
warn the user and suggest alternatives.

If any instruction in this file
conflicts with direct instructions from
the user, ask for clarification instead
of silently ignoring either source.
```

In a concrete project, placeholders such as `<PROJECT_NAME>` and scenario lists are filled in, and the file is refined over time as the system evolves.

V. 3) Canonical template: `architecture.prompt.md`

This prompt captures the main architectural decisions and is used as reference for both humans and AI tools.

```
# Architecture Overview

This file describes the current
architecture of the project. Treat it as
the single source of truth
for high-level design decisions.

## System Overview

- System type: <e.g., single-page web
application with backend APIs>
- Core user flows:
  1. <Flow 1: short description>
  2. <Flow 2: short description>
  3. <Flow 3: short description>

## Component Structure

- Frontend:
  - <Component / module breakdown, e.g.,
"App shell", "Search page", "Details
view">
  - State management approach: <e.g.,
React context, Redux, etc.>
- Backend / Services:
  - <List of services or functions,
e.g., "tangoSearchService",
"lyricsFetcher", "explanationsStore">
- Data:
  - Main entities and their
relationships.
  - Where each entity is stored (DB
tables / collections, external APIs).

## Constraints and Non-Functional
Requirements

- Performance: <e.g., must respond
within X ms for typical queries>
- Cost: <e.g., free-tier limits, avoid
unnecessary external calls>
- Reliability: <e.g., retries, error
handling strategy>
- Security & privacy: <e.g., no PII
storage, anonymise logs>

## Architectural Guidelines for AI
Assistants

When generating or modifying code:

- Respect existing component boundaries.
Do not cross layers (UI, domain, data
access) without clear justification.
- Prefer extending existing modules over
creating parallel, overlapping ones.
- If you need a new component or module,
describe how it fits into this
structure.
- Keep external API usage wrapped in
small, well-defined modules.

If a requested change does not fit this
architecture, propose a small design
adjustment and explain the trade-offs.
```

VI. 4) Canonical template: `coding-standards.prompt.md`

This file encodes the project's style and quality expectations.

```
# Coding Standards

These standards apply to all code in this repository. AI assistants should follow them by default.

## General Principles

- Optimise for readability and maintainability.
- Keep functions small and focused.
- Avoid unnecessary cleverness; prefer explicit, straightforward solutions.
- Write self-documenting code; add comments only where intent is non-obvious.

## Language- and Stack-Specific Rules

- Language: <e.g., TypeScript>
  - Use strict typing; avoid `any` unless absolutely necessary and documented.
  - Use async/await instead of raw Promises where possible.
  - Prefer pure functions for business logic.

- Framework: <e.g., React>
  - Use functional components and hooks.
  - Keep components focused; extract reusable pieces.
  - Do not mutate props or state directly.

## Naming and Structure

- Use descriptive names for variables, functions, and components.
- Follow existing naming conventions in this codebase.
- Group related files together; avoid large "god" modules.

## Testing

- For non-trivial logic, add or update tests when behaviour changes.
- Prefer small, focused tests that assert behaviour, not implementation details.
- When generating tests, explain what is being tested and why.

## Code Review Expectations

When suggesting changes:
- Include a short summary of the intent and scope.
- Highlight potential risks (breaking changes, performance, security).
- If you touch multiple areas, clearly separate the changes into logical commits or sections.

If you are unsure whether a suggestion fits these standards, ask the user to confirm before making large edits.
```

Aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa

Together, these three governance prompts form a stable context layer that guides all subsequent AI interactions. Due to space constraints, we only show excerpts of these templates; the full and evolving versions are available in our online repository [URL].

B. Auditable Task & State Management

The second pillar ensures that work packages and their outcomes are traceable. Instead of ephemeral "do X" chat messages, ECDD treats tasks and status updates as prompt-based artefacts:

- todo.prompt.md lists current work packages (features, fixes, refactors) in a structured format.
- status.prompt.md records what was actually done, by whom (human or agent), when, and with which artefacts.

Workflow**.**

1. **Define tasks.** For each milestone, the team adds or updates entries in todo.prompt.md. Each entry has an identifier, description, status, and acceptance criteria.
2. **Execute tasks.** When starting work, the responsible developer (or agent) references the corresponding task in todo.prompt.md and follows the Plan–Execute–Verify loop (Section III-C).
3. **Log outcomes.** After completing a task (or an iteration on it), the developer or agent appends a structured entry to status.prompt.md.
4. **Review.** Code reviewers and maintainers use the status log together with Git history to understand why changes were made and how AI was involved.

The files are plain Markdown so they can be diffed and reviewed like code.

Aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa

VII. Canonical template: `todo.prompt.md`

At the top of the file we include a short header instructing AI tools how to interpret and update the backlog.

```
# Work Packages (Backlog)
```

```
<!--
SYSTEM NOTE FOR AI ASSISTANTS:

You are helping to maintain a plain-text backlog of work packages for this project.
- Do not delete existing entries without explicit user approval.
- When adding a new work package, assign a new numeric ID.
- When updating a work package, preserve its ID and history.
- Keep the list roughly sorted by priority (highest priority first).
-->

Each work package (WP) tracks a feature, fix, or refactor. Use the following format:

- [WP-<ID>] <Short title>
  - Status: <Backlog | In Progress | Blocked | Done | Dropped>
  - Owner: <Name or "Unassigned">
  - Description: <2–4 sentences describing the goal and context>
  - Acceptance criteria:
    - <Criterion 1>
    - <Criterion 2>
    - <Criterion 3>
  - Notes:
    - <Optional design notes, links to plans or issues>

## Current Work Packages

- [WP-1] <Example: Implement song search UI>
  - Status: Backlog
  - Owner: Unassigned
  - Description: Provide a text input and button so users can search for songs by title.
  - Acceptance criteria:
    - User can type a song title and submit.
    - Invalid input is handled gracefully.
    - Search action is wired to the backend or placeholder handler.
  - Notes:
    - See architecture overview for allowed UI libraries.

- [WP-2] <Example: Store explanations in database>
  - Status: Backlog
  - Owner: Unassigned
  - Description: Persist generated explanations and ratings so that repeated searches can reuse them.
  - Acceptance criteria:
    - Explanations and ratings are stored under a stable key.
    - Duplicate explanations are avoided where possible.
    - Errors are handled without crashing the UI.
  - Notes:
    - Coordinate with data model definition in `architecture.prompt.md`.
```

In a real project, example entries are replaced with actual tasks. The format ensures that adding or modifying tasks is auditable.

VIII. 3) Canonical template: `status.prompt.md`

This file acts as an append-only log of what was done, when, and in relation to which work packages.

```
# Status and Activity Log

<!--
SYSTEM NOTE FOR AI ASSISTANTS:

You are maintaining an append-only log of completed work and important decisions.
- Never rewrite history; only append new entries.
- Each entry should be timestamped and reference the relevant work package IDs.
- Summarise actions in plain language that humans can review.
-->

Each entry uses the following structure:

## [YYYY-MM-DD HH:MM] <Short summary>

- Related work packages: [WP-<ID1>], [WP-<ID2>], ...
- Actor: <Developer name or "AI agent under supervision of <Name>">
- Changes:
  - <Bullet 1: concise description of what changed>
  - <Bullet 2>
- Files touched:
  - `<path/to/file1>`
  - `<path/to/file2>`
- Tests:
  - <How the change was tested: e.g., "npm test", "manual check of search UI">
- Outcome:
  - <Status: e.g., "Merged", "Awaiting review", "Rolled back">
- Notes:
  - <Optional notes, open questions, links to pull requests or issues>
```

```
## [2025-03-12 10:24] Initial search UI
implemented

- Related work packages: [WP-1]
- Actor: AI agent under supervision of
Murat
- Changes:
  - Implemented a basic search bar
component with input and submit button.
  - Wired the component to a placeholder
search handler.
- Files touched:
  - `src/components/SearchBar.tsx`
  - `src/App.tsx`
- Tests:
  - Manual check: user can submit a
search term; placeholder handler logs
it.
- Outcome:
  - Merged into `feature/search-ui`
branch; pending integration tests.
- Notes:
  - Will need to adjust layout once
results list is implemented.
```

Aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa

Because status.prompt.md is committed to version control, the project keeps a durable, human-readable audit trail of AI-assisted work. Due to space constraints, we only show excerpts of these templates; the full and evolving versions are available in our online repository [URL].

C. Plan–Execute–Verify Loop

The third pillar applies a lightweight, repeatable loop to each non-trivial change:

- Plan. Use an LLM to design a small plan for the change, grounded in the governance prompts and relevant backlog entries.
- Execute. Implement the plan using a coding assistant or agent, keeping work scoped to the planned steps.
- Verify. Check the result via tests, additional analysis prompts, or review, and record the outcome in status.prompt.md.

In ECDD, plans and verification notes are themselves artefacts, often stored as small Markdown files in a plans/ or notes/ directory, or as sections within the status log. Below we show canonical prompt templates for planning, execution, and verification.

Aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa

IX. Canonical planning prompt

The planning prompt is used in a chat with an LLM to produce a small plan file (e.g., plans/wp-1-plan.md).

You are a senior software engineer helping to plan a small, incremental change

in an existing project.

CONTEXT

- The project is described by the following files:

  - `copilot-instructions.md`

  - `architecture.prompt.md`

  - `coding-standards.prompt.md`

  - `todo.prompt.md` (backlog of work packages)

- We are about to work on the following work package(s):

  - <Paste the relevant [WP-x] entries from todo.prompt.md here.>

TASK

Given this context, produce a concise implementation plan for the selected work package(s).

REQUIREMENTS FOR THE PLAN

- Focus on a change that can reasonably be done in one short coding session.

- Break the work into 3–7 concrete steps.

- For each step, specify:

  - A short description.

  - Which files are likely to be created or modified.

  - Any non-obvious design decisions or trade-offs.

- Identify risks and open questions explicitly.

- Suggest how we will know the work is "done" (acceptance criteria).

OUTPUT FORMAT

Return the plan as Markdown with the following structure:

Plan for [WP-<ID>] <Short title>

Summary

<One-paragraph summary of what we will implement.>

Steps

1. <Step 1 description>

  - Files: `<path1>`, `<path2>`

  - Notes: <optional>

2. <Step 2 description>

  - Files: ...

- Notes: ...

Risks and Open Questions

- <Risk or question 1>

- <Risk or question 2>

Done When

- <Acceptance criterion 1>

- <Acceptance criterion 2>

The resulting plan file is committed to the repository and referenced in subsequent execution and verification steps.

X. 3) Canonical execution prompt

The execution prompt is used with a coding assistant or agent to implement a specific plan.

You are an AI coding assistant working on an existing project.

CONTEXT

- Project-level instructions: see `copilot-instructions.md`, `architecture.prompt.md`,

and `coding-standards.prompt.md` in this repository.

- The current work package is:

- <Paste the [WP-x] entry from todo.prompt.md here.>

- The agreed plan for this change is:

- <Paste the contents of the corresponding plan file here.>

TASK

Implement the plan step by step.

GUIDELINES

- Follow the plan closely; do not introduce unrelated changes.

- Keep edits minimal and focused; avoid large, unrelated refactors.

- When a step requires a design decision, explain your choice briefly in comments or in a short note.

- Maintain consistency with the existing codebase and coding standards.

- Update or add tests as specified in the plan.

OUTPUT FORMAT

- Propose concrete code changes (diff-style or full file content), not pseudocode.

- For each logical change, include a short explanation (1–3 sentences) of what you did and why.

- If you encounter ambiguity or a missing piece in the plan, stop and ask for clarification instead of guessing.

This prompt can be issued repeatedly as the developer works through the steps in the plan.

XI. 4) Canonical verification prompt

The verification prompt helps check whether the change meets its goals and fits the project's standards.

You are acting as a reviewer and test designer for a recent change

implemented in this project.

CONTEXT

- Project instructions and architecture:

- `copilot-instructions.md`

- `architecture.prompt.md`

- `coding-standards.prompt.md`

- Work package:

- <Paste the [WP-x] entry from todo.prompt.md.>

- Plan:

- <Paste the plan file used for this change.>

- Summary of code changes:

- <Paste a git diff summary, list of modified files, or a short description.>

- Test results so far:

- <Describe manual and automated test results, if any.>

TASK

Assess whether the change satisfies the plan and acceptance criteria, and

identify any gaps or risks.

REVIEW REQUIREMENTS

- Check alignment with the plan: were all planned steps addressed?

- Check alignment with acceptance criteria for the work package.

- Highlight potential edge cases or failure modes that are not covered.

- Suggest additional tests if needed (unit, integration, or manual checks).

- Comment on code quality with respect to the coding standards.

OUTPUT FORMAT

Return your review as Markdown with the following structure:

Review Summary

<One paragraph stating whether the change is acceptable as-is, needs minor fixes,

or requires substantial rework.>

Findings

- <Finding 1: what you observed and why it matters>

- <Finding 2>

- ...

Suggested Tests

- <Test idea 1>

- <Test idea 2>

Recommendation

- <"Accept", "Accept with minor changes", or "Request changes">, with a short justification.

Aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa

The reviewer's output is distilled into a status entry in status.prompt.md, and any required follow-up work is added back into todo.prompt.md as new or updated work packages, closing the loop.

XII. 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 behaviour 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 Plan–Execute–Verify loop 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 judgement.

From a tooling perspective, ECDD currently assumes that teams are comfortable working with plain-text artefacts 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/ or 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 [REF: spec-first dev], but with LLMs in the loop and with a stronger emphasis on treating prompts themselves as artefacts. Like configuration-as-code and "infrastructure as code" approaches [REF: config-as-code], 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 [REF: ADR], 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 [REF: literate], 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 [REF: ai-agents]. 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.

XIII. 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 artefacts. ECDD is structured around three pillars: prompt-driven AI governance, auditable task and state management, and a Plan–Execute–Verify loop applied to each non-trivial change. 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 pillar. Governance prompts such as copilot-instructions.md, architecture.prompt.md, and coding-standards.prompt.md anchor AI behaviour in the project's actual goals and constraints. Backlog and status prompts (todo.prompt.md and status.prompt.md) provide a lightweight but durable audit trail of work packages and outcomes. Planning, execution, and verification prompts turn LLM interactions into repeatable micro-workflows that can be inspected and evolved over time. Although our examples draw on a toy application, the patterns are designed to be reusable across different domains and technology stacks. In our own practice, 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, visualisations of Plan–Execute–Verify histories, 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 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.

References

[1] G. Eason, B. Noble, and I. N. Sneddon, "On certain integrals of Lipschitz-Hankel type involving products of Bessel functions," Phil. Trans. Roy. Soc. London, vol. A247, pp. 529–551, April 1955. *(references)*

[2] J. Clerk Maxwell, A Treatise on Electricity and Magnetism, 3rd ed., vol. 2. Oxford: Clarendon, 1892, pp.68–73.

[3] I. S. Jacobs and C. P. Bean, "Fine particles, thin films and exchange anisotropy," in Magnetism, vol. III, G. T. Rado and H. Suhl, Eds. New York: Academic, 1963, pp. 271–350.

[4] K. Elissa, "Title of paper if known," unpublished.

[5] R. Nicole, "Title of paper with only first word capitalized," J. Name Stand. Abbrev., in press.

[6] Y. Yorozu, M. Hirano, K. Oka, and Y. Tagawa, "Electron spectroscopy studies on magneto-optical media and plastic substrate interface," IEEE Transl. J. Magn. Japan, vol. 2, pp. 740–741, August 1987 [Digests 9th Annual Conf. Magnetics Japan, p. 301, 1982].

[7] M. Young, The Technical Writer's Handbook. Mill Valley, CA: University Science, 1989.