ChatGPT

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

Murat Karakaya

Abstract

AI-assisted programming tools like GitHub Copilot can dramatically accelerate software development, but informal “vibe coding” – riffing on a rough idea without clear guidance – often leads to unpredictable results. We present **Explicit Context-Driven Development (ECDD)**, a structured methodology that makes context explicit and tasks auditable in AI-augmented development. ECDD is built on three pillars: Prompt-Driven AI Governance (version-controlled project instructions and coding standards), Auditable Task & State Management (prompts and logs that track work packages), and a Plan–Execute–Verify loop for each feature or refactoring. We demonstrate ECDD through a detailed case study of building the “Tango Song Explainer” web app (a Firebase/React app using an LLM and Gemini Search API). The case study covers setting up custom instructions and prompts, planning features with LLMs, coding via GitHub Coding Agents and Copilot, and continuous integration/deployment. Key contributions include (1) formalizing ECDD as a practical framework for AI-driven workflows, (2) illustrating prompt engineering in version control, and (3) providing a concrete “from idea to deploy” example. We conclude that disciplined, prompt-based processes are essential for treating AI as a reliable copilot rather than an unpredictable autopilot.

Introduction

AI coding tools hold great promise but also new risks. Unlike traditional development, advanced LLMs can generate entire features from high-level prompts, which tempts developers to “throw ideas at the AI” and hope for the best – a practice one source calls **“YOLO vibe coding”** [1]. While fun for exploration, this approach often yields incoherent code or hidden bugs when scaled to production. Modern best practices stress the opposite: explicitly encoding project goals, constraints, and conventions so the AI assistant can act as a well-informed collaborator [2] [3]. In other words, treat specifications and context as version-controlled artifacts (“version control for your thinking” [4]) rather than implicit guesswork.

We introduce **Explicit Context-Driven Development (ECDD)**: a methodology that puts context upfront and makes all AI-driven steps auditable. In ECDD, project context (architecture, API contracts, coding rules) is stored in dedicated prompt files (e.g. in `.github/copilot-instructions.md` and `.prompt.md` files) so that the AI always has the full picture. Work is broken into clear tasks, tracked in 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), and verify via tests or reviews. This aligns with the industry notion of “spec-driven development” for AI [1] [4]. ECDD ensures the human developer retains control (“Copilot, not autopilot” [2]) and produces reproducible, reviewable outcomes.

This paper presents ECDD's principles and demonstrates them via a complete **Tango Song Explainer** case study. The methodology section defines the three ECDD pillars: Prompt-Driven AI Governance, Auditable Task/State Management, and the Plan–Execute–Verify loop. The case study walks through setting up a Git repo, writing project/instructions prompts, using LLMs to plan and implement features, and deploying to Firebase with CI/CD. We include tables contrasting implicit vs explicit context and LLM usage modes (planning vs coding), as well as sample prompt/code snippets. Finally, we discuss limitations (qualitative focus, lack of quantitative evaluation) and future work comparing ECDD to tools like SpecKit, OpenSpec, and BMAD [4] [5]. Our contribution is a practical, IEEE-style guide that shows how engineering discipline can tame AI coding for real-world projects.

Methodology

Prompt-Driven AI Governance

In ECDD the project's "culture" and rules are explicitly encoded in version control. A central file (e.g. `.github/copilot-instructions.md`) contains the **project overview, architecture, API contracts, tech stack, and coding standards** [6] [7]. By convention, GitHub Copilot's custom instructions use this file as a "persistent prompt" for all chat requests [6]. For example, a project copilot-instructions file might list React/Tailwind front-end, Firebase backend, UI design rules, and naming conventions [7]. This ensures every AI suggestion aligns with the project's conventions. Additional governance files (e.g. `config/.editorconfig`, lint rules) complement these prompts.

The key is to **move from implicit to explicit context** [2]. Implicit context means the AI only sees the surrounding code in the editor, which may not convey overall goals or constraints. Explicit context is provided by the developer via written prompts and documentation. Table 1 contrasts these:

- **Implicit Context**: the AI infers intent from local code, open files, and environment [8]. It requires no extra work but has limited view. Suggestions may conflict with unstated conventions.
- **Explicit Context**: the developer writes descriptions of the project scope, architecture, and rules in prompt files. The AI's outputs are then grounded in this context. For example, the .prompt.md and copilot-instructions.md files define the "elevator pitch", tech stack, and style guidelines [7] [6].

This prompt-based governance effectively creates a "context layer" above the code [9]. It elevates the human to architect-orchestrator: instead of reacting to Copilot's unsolicited suggestions, the developer steers it with clear instructions. ECDD requires that any time a new library or framework is introduced, or a change is made to design principles, the instructions file is updated. As one source notes, this version-controlled context file becomes a single source of truth that keeps human and AI aligned [9].

Auditable Task & State Management

ECDD treats development work as a set of discrete, trackable tasks. We use **prompt files as tickets and logs**. For example, a `todo.prompt.md` might list the current Work Packages (features or fixes) for an MVP, and a `status.prompt.md` logs completed work. In the Tango example, the initial `todo.prompt.md` could have entries like "Build search bar UI" and "Implement Firebase query". After an agent completes a task, it appends a structured update to `status.prompt.md` [10]. Figure 1 shows an excerpt of the prescribed status format, which includes work-package name, timestamp, actions taken, modified files, and a comparison to the todo list [10].

Figure 1: Plan–Execute–Verify loop (cycles of planning tasks with an LLM, executing via a coding agent or Copilot, and verifying with tests).

This log is **auditable**: every change is described in plain language, and the file is committed to Git. Anyone (or any auditing tool) can see the progress and that AI suggestions were reviewed. We also create a GitHub issue or branch for each task, tying the code commits to these prompt tickets. For example, we might open issue "UI: Add song list sidebar" and a branch `feature/song-list`. When the coding agent finishes and a PR is submitted, the `status.prompt.md` entry will note "Implemented feature X", list the changed files, and mark that task as done [10].

This disciplined logging guards against silent autopilot. If the AI agent tries to implement a hidden task, it won't appear in `todo.prompt.md`, so its changes will be obvious in `status.prompt.md`. Importantly, *only actual completed actions are logged*, and entries must match the structured template exactly [10]. This prevents both fabricated progress and overlooked work. Auditing is as simple as reviewing these Markdown logs and the Git diff for each update.

Plan–Execute–Verify Loop

Each work package in ECDD follows a **Plan–Execute–Verify** cycle, analogous to the Agile concept of spec-driven implementation. We first **Plan** with a powerful LLM (e.g. GPT-4), producing a clear blueprint in a prompt file. For a new feature, this might be a prompt like: "Act as a Software Architect planning a new feature. Generate a detailed skeleton plan for adding links to the 'Notable Recordings' section. Save output to `add-links.prompt.md` with headings like 'Feature Goal' and 'Implementation Steps'." An example plan file would outline goals, affected components, stepwise tasks, and draft acceptance criteria [11]. The AI often autonomously refines this plan: it might add a section "Current Workflow & Feature Understanding" to document the existing LLM-parse-Firestore pipeline and highlight exactly which code paths need changes [12].

Once the plan is validated (we may even run a separate "Verify plan" prompt where another LLM checks completeness) [13], we move to **Execute**. We choose either an *Agent Mode* or *Edit Mode* for coding [14]. For larger multi-file features, agent mode is efficient: we give Copilot a prompt to create a new branch and implement the plan. For example: "Branch *feature/add-song-search*. Implement the steps in `project.prompt.md` and `todo.prompt.md` for the song search UI." The AI agent then types code, makes commits, and asks for feedback. The developer supervises in real time, giving feedback prompts or "undo" commands to steer corrections. We emphasize **context window management**: keep prompts concise by loading only needed files (#file mentions) and split long tasks into multiple agent runs.

During execute, we commit code frequently. The prompt plan and status log help here: after a subset of tasks are done, we update the status file and possibly the issue description. We also keep `copilot-instructions.md` up to date (for example, if we add a new Firebase data model, we record it). In practice, one developer note is: *"First implement the frontend, then backend – the other way is very hard"* [15]. Another tip is to do heavy LLM planning first, then hand off to a cheaper model or coding agent for implementation [15].

Finally, **Verify** means testing and review. We run manual smoke tests and automated suites (e.g. Playwright tests for the UI). If tests fail, we let a coding agent attempt fixes under constraints: for example, prompt "I have applied these refactorings. Run all tests and fix only the *test files* if they fail, do not change application code [16]." We carefully preserve one person's oversight: "the human-in-the-loop remains responsible for quality" [17]. We use pull requests for code review. Only after passing tests and review do we merge and mark the status complete.

This Plan–Execute–Verify loop is continuous. For refactoring existing code, the "Plan" step might be lighter: we describe the current behavior and desired change, then let a coding agent apply edits in Chat- or Edit-mode. Throughout, we leverage features like Copilot's `#file` and `#codebase` mentions to give precise context. By contrast to random prompting, this structured approach ensures each code change is traceable to a clear plan and verification step.

Case Study: The Tango Song Explainer App

To illustrate ECDD in action, we built a toy web app: **Tango Song Explainer**. The app lets a user enter a tango song name, then explains its meaning using web search and LLMs, storing results in Firebase for reuse. We walk through key steps (see Fig. 2 for a git workflow diagram and Fig. 3 for the ECDD prompt flow).

1. **Repository Setup**: We started by creating a new Git repo (`tango-songs`). In its root, we made a `.github/` folder. We created `copilot-instructions.md` with high-level context: "This is a React Single-Page App hosted on Firebase Hosting. Users search for tango songs, which are looked up via Google Gemini Search API. Explanations are stored in Firestore along with user ratings. Use Material-UI for a clean design. Follow our coding standards (single quotes in JS, semicolons, etc.)." (The PDF best-practices suggests leaving it blank initially then filling after planning [18], but we prepped it from day one.)

2. **Project Definition Prompt**: We wrote `project.prompt.md`:

```
I want to build a simple web app:
- Frontend: React app (create-react-app or Next.js), modern clean UI
(CSS frameworks allowed).
- Backend: Firebase Firestore for storage, Firebase Auth optional for
user rating.
- Features:
  1. Song search UI (text input, button).
  2. If song not in DB, call Gemini Search API to get lyrics/context.
  3. Use an LLM (e.g. ChatGPT/Gemini) to analyze lyrics and output an
explanation.
  4. Store explanation in Firestore under the song name.
  5. On success, show explanation to user.
  6. Provide a 1–5 star rating input; save rating.
Keep design minimal but responsive. Project must use only Firebase free-
tier features (no paid functions).
```

   This prompt was iteratively improved via Copilot Chat (the handbook example shows doing "2A … check project.prompt.md for missing data" etc [19]). The final prompt served as an authoritative spec for the app's scope and tech.

3. **Initial Task List**: We created `todo.prompt.md` with the initial MVP tasks: UI mockup, search integration, data model, example test. For example:

```
## To-Do (v1):
- Build React UI with a search bar and results area.
- Implement Firebase Firestore integration (add song explanation).
- Integrate Gemini Search LLM call for new songs.
- Display explanation and store in DB, then reload next time without
LLM.
- Add star-rating widget (save to DB).
- Write initial unit/integration tests.
```

We committed this to Git.

4. **Planning with LLM**: We gave the plan prompt to GPT-4 (or Copilot in Plan mode): "Plan the above features step by step. Output as bullet points." The AI produced a multi-step plan, splitting frontend and backend tasks and referencing Firestore schema. We saved this as `plan.feature1.prompt.md`. It even suggested verifying user input and error handling – we added these to the plan and updated `todo.prompt.md` accordingly.

5. **Coding (Execute)**: For the first feature (UI with search bar), we created a branch `feature/search-ui`. We opened a chat with Copilot and prompted it: "Create the React components for a search bar and results display. Use Material-UI. Save code in `src/SearchBar.js` and `src/App.js`." Copilot began writing JSX code. We reviewed and corrected in realtime (for example, adjusting CSS or adding missing imports). We committed the initial UI.

Next, we tackled the backend logic. On branch `feature/firestore`, we wrote a prompt: "Add Firebase Firestore integration: if a song is submitted, check the `songs` collection; if not found, call Gemini API, store result." Using Copilot's Chat with `#file` context gave Copilot our component code; it inserted Firestore rules. After some trials, the basic lookup/save code was in place.

We then merged these branches to `develop`. At each merge, we updated `status.prompt.md` with what was implemented (e.g. "Implemented UI search bar. Files: SearchBar.js, App.js. Task 1 completed").

1. **Verification**: We manually tested by running `npm start`. Entering a song (e.g. "La Cumparsita") triggered the Gemini API call and showed a dummy explanation. We wrote an end-to-end Playwright test (the handbook mentions using MCP servers for Playwright [20]). One test failed, so we gave Copilot an instruction in a new chat:

   ```
   I ran the test and got an error. The component is not importing
   Firestore correctly. Fix only the import lines in the test file and
   code.
   ```

   Copilot suggested adding `import { getFirestore } from 'firebase/firestore'`. We applied the fix and tests passed.

2. **CI/CD Setup**: Once the feature was stable, we wrote a GitHub Actions workflow (`.github/workflows/deploy.yml`) to deploy to Firebase. We configured secrets (FIREBASE_TOKEN, PROJECT_ID) as per Firebase guidance [21]. The LLM helped write the action YAML from scratch when prompted "Configure Firebase CLI deploy on push to main." We tested by merging to `main`, which triggered a GitHub Actions run and published the site to Firebase Hosting.

Through all steps, the **context layer** (instructions and prompts) was continuously updated. For example, when we added the Gemini Search, we amended `copilot-instructions.md` to note the API usage. We also recorded decisions: e.g. in `status.prompt.md` we noted “Switched from ChatGPT to Gemini API for better song info retrieval.” This makes the project’s decision history explicit.

The end result was a functioning app with version-controlled AI workflows: every feature had a documented plan, every code change was linked to a prompt/instruction file, and tests ensured correctness. Table 2 compares using implicit (typical Copilot typing) vs explicit (ECDD) workflows in this case.

Discussion and Limitations

ECDD emphasizes **process over spectacle**. In our case study, we did not measure speed or output quality quantitatively, focusing instead on showing that a disciplined approach *is feasible*. One limitation is that writing detailed prompts and logs takes effort; a quantitative analysis of overhead vs benefit remains future work. We also assumed a certain level of AI reliability (e.g. Copilot’s ability to follow structured plans) which may vary between models or providers. Smaller or less experienced teams might find the setup intensive.

Another limitation is the lack of formal metrics. We do not, for example, compare defect rates or development time between ECDD and ad-hoc coding. Our evaluation is qualitative, based on the logic that clarity and auditability inherently improve maintainability [9]. Empirical validation (e.g. user studies or project benchmarks) would strengthen the case, but is beyond this paper’s scope.

Additionally, ECDD relies on versioning prompt files, which some toolchains might not yet natively support. We treated prompt files as code files (tracked in Git), but continuous integration around them (linting prompts, merging conflicts) needs more tooling. Our approach was partly manual. As AI-assistance tools mature, we expect better integration (for example, automated checks of prompt completeness).

Future Work

We plan to systematically compare ECDD with other structured AI workflows. GitHub’s Spec Kit [3], OpenSpec (a minimalist spec-driven CLI), and the BMAD Method [22] are three contemporaneous approaches that share the goal of adding “rigid structure” to AI coding. Preliminary investigation shows ECDD aligns with the idea of *version-controlled context* and *human-driven planning* like Spec Kit [4], but differs in simplicity and tooling (ECDD uses plain Markdown and Copilot vs Spec Kit’s Python CLI templates). OpenSpec is more lightweight yet similar in spirit, and BMAD’s multi-agent ecosystem is heavier-weight for enterprise projects [22]. We intend to build side-by-side case studies (e.g. adding a feature to a project) using these methods, comparing effort, clarity, and correctness. We will also explore automated tracking of prompt changes (e.g. using Git diff on `.prompt.md` files as part of code reviews).

Conclusion

AI in software development should be a **copilot, not an autopilot** [17]. The ECDD methodology shows how to make AI collaboration predictable and reviewable. By encoding project context, tasks, and plans as explicit, version-controlled prompts and logs, teams can ensure AI suggestions respect their architecture and standards. In our Tango app case study, this meant every feature had a clear spec file, a recorded task list, and a verification step, resulting in an auditable development history. We observed that well-structured prompts effectively become *live documentation* that steers the AI [9] .

In short, the magic of modern LLMs still requires human engineering discipline. A development workflow that harnesses prompt engineering, git habits, and testing is key. We encourage practitioners to adopt ECDD's principles: treat prompts as code, keep humans in the loop, and always *plan, execute, verify*.

References

[1] M. Karakaya, *Copilot Handbook for Developers*, unpublished developer guide, 2024. (Chapter 7: Best Practices) [7] [10]

[2] Den Delimarsky, "Diving Into Spec-Driven Development With GitHub Spec Kit," *Microsoft Developer Blog*, Sep. 15, 2025. Accessed Jan. 2025. [4]

[3] Tim Wang, "Spec-driven AI coding: Spec-kit, BMAD, AgentOS and Kiro," *Medium*, Oct. 18, 2025. [5]

[4] Z. Saadioui, "The Future of Code: A Phased Approach to Plan, Execute, & Verify with Your AI Agent," *Arsturn Blog*, Aug. 10, 2025. [1]

[5] Atlassian, "Gitflow Workflow," *Atlassian Git Tutorial*, 2023. [Online]. Available: https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow (accessed Apr. 2025).

[6] GitHub, "Configure custom instructions – repository-wide," *GitHub Docs*, 2024. [Online]. Available: https://docs.github.com/copilot/how-tos/configure-custom-instructions/add-repository-instructions (accessed Apr. 2025).

[7] GitHub, "GitHub Spec Kit," GitHub Repository, 2025. [Online]. Available: https://github.com/github/spec-kit (accessed Apr. 2025).

[8] GitHub, "GitHub Copilot – Copilot Not Autopilot," official documentation, 2024. [Online]. Available: https://docs.github.com/copilot (accessed Apr. 2025).

---

[1] How to Use AI Coding Agents: A Plan, Execute, Verify Guide
https://www.arsturn.com/blog/a-phased-approach-to-ai-coding-agents

[2] [6] [7] [8] [9] [10] [11] [12] [13] [14] [15] [16] [17] [18] [19] [20] [21] Copilot Handbook for Developers(1).PDF
file://file_00000000a450720eb207cf2a69cb5ba8

[3] [4] Diving Into Spec-Driven Development With GitHub Spec Kit - Microsoft for Developers
https://developer.microsoft.com/blog/spec-driven-development-spec-kit

[5] [22] Spec-driven AI coding: Spec-kit, BMAD, Agent OS and Kiro | by Tim Wang | Oct, 2025 | Medium
https://medium.com/@tim_wang/spec-kit-bmad-and-agent-os-e8536f6bf8a4