Vibe Coding Principles

Six Principles for Responsible AI-Assisted Programming

Gábor Kusper

Eszterházy Károly Catholic University, Eger, Hungary
kusper.gabor@uni-eszterhazy.hu

Abstract. Vibe coding has become a recognizable label for conversational, flow-oriented AI-assisted programming. It promises speed, experimentation, and a strong sense of creative momentum, yet it also weakens direct code comprehension and can hide assumptions, risks, and architectural debt. This article proposes a compact conceptual framework for disciplined vibe coding, developed by synthesizing recent literature, community gray literature, and classroom practice. The framework contains six principles: Responsible Vibe Coding, Tool-Aware Vibe Coding, Specification-First, Plan–Guide–Verify, Rule of Principle-Guided Prompts, and Definition of Done. Its novelty lies in relocating familiar software-engineering commitments to the prompt-based conversational workflow. A short prompt-based example and preliminary classroom evidence illustrate how the principles can guide AI-assisted programming. The framework is not presented as a fully validated method, but as a teachable vocabulary for preserving accountability, making intent explicit, supporting verification, and keeping later refactoring feasible.

Keywords: vibe coding, AI-assisted programming, software quality principles

1 Introduction

Vibe coding names a style of AI-assisted programming in which a developer steers a strong LLM (Large Language Model) through natural language, accepts generated code quickly, and prioritizes visible progress over line-by-line authorship. Karpathy's early formulation made the style memorable by describing a mode in which one can "forget that the code even exists" and keep moving through prompts, error messages, and accepted diffs [1]. He captures both the attraction and the danger of vibe coding. It lowers friction and sustains flow, but it can also normalize unreviewed changes, reduced comprehension, and workaround-driven development.

The educational meaning of this change is already visible. If a model can produce substantial code from natural language, then part of programming education moves upward toward problem formulation, prompt design, lightweight specification, and critical evaluation. Recent computing-education work therefore does not make code reading obsolete. Instead, it suggests that generated code must be interpreted, challenged, and owned by the learner or developer [2–4].

© The Author(s) 2026
S. Árvai-Homolya et al. (eds.), *Proceedings of the 2nd Innovation in Engineering Education Conference (IEE 2026)*, Atlantis Highlights in Social Sciences, Education and Humanities 58,
https://doi.org/10.2991/978-94-6239-823-8_10

This shift also echoes an older aspiration from declarative programming: progress comes from stating more clearly what the system should do while delegating more low-level realization to the machine. Vibe coding gives this aspiration a conversational and iterative form. The result is powerful, but unstable unless intent, review, and stopping criteria become explicit.

The article argues that vibe coding changes development in two opposite directions at once. Rapid visible success can prolong productive flow. The same momentum can also weaken responsibility, blur user intent, create false confidence, and make later change harder. In response, the paper proposes six Vibe Coding Principles. They are intentionally short and principle-level. Like older object-oriented design principles, they do not prescribe one route. They act as lighthouses: they illuminate routes that are more likely to remain understandable, extensible, and responsible under change [5–7].

The contribution is conceptual. It does not claim that the six principles are final, nor that they replace empirical validation. Its aim is more modest: to stabilize a fast-moving discussion in a form that is easier to teach, cite, refine, and operationalize.

The novelty of the framework is not that responsibility, specification, verification, design principles, or completion criteria are new software-engineering ideas. Rather, the novelty lies in relocating these ideas to the prompt-based and conversational surface of AI-assisted programming. In vibe coding, intent, design constraints, review expectations, and stopping criteria must be expressed before and during generation, not only after code exists. The six principles therefore reformulate familiar engineering commitments for a workflow in which natural language, tool context, and generated diffs become central development artifacts.

2 Related Work and Scope

Current work usually frames vibe coding as conversational, flow-oriented AI-assisted development in which the human and the model co-produce code through prompts, context, and iterative feedback. A key theme is intent mediation: more meaning is negotiated through natural language and tool context rather than through direct low-level implementation [8, 9]. This does not make the workflow fully autonomous. Even when delegation is extensive, the human remains the final owner of intent, judgment, and consequence [10].

Empirical and qualitative signals point to a recurring tension. Developers value vibe coding because it enables speed, co-creation, and a strong sense of flow; at the same time, they report fragile trust, weak comprehension, and uncertainty about when a result is good enough [11]. Studies of students and domain-specific learning designs show a similar mixture: AI can lower barriers and increase experimentation, but it also demands new forms of explanation, reflection, and assessment [12–14].

Software-engineering literature adds a second lens. Design principles, patterns, tests, contracts, and reviews emerged because growing systems become difficult to change. Vibe coding does not remove those problems. It can even accelerate them,

because code grows quickly while the developer's mental model grows more slowly. Work on vibe modeling, control-oriented development, contracts, project memory, and context-before-code reports therefore converges on the same practical need: stronger anchors for intent, context, verification, and maintainability [15–18].

A more critical strand of work warns that AI-assisted coding and vibe coding should not be read only as productivity stories. Security-oriented studies of AI code generation show that generated code can contain serious weaknesses: Pearce et al. found that a substantial portion of Copilot-generated programs in security-relevant scenarios were vulnerable [19]. In computing education, Prather et al. report that GenAI can widen the gap between students who can use AI strategically and students whose metacognitive difficulties are amplified by AI assistance [20]. More recent vibe-coding-specific work makes a similar point from inside the emerging practice itself. Chou et al. describe vibe coding as a spectrum ranging from almost uninspected reliance on AI output to more careful examination and adaptation, with debugging and refinement often becoming a partly stochastic process [21]. Waseem et al. formulate the same concern as a flow–debt trade-off: rapid conversational generation may support prototyping and MVP development, but it can also accumulate technical debt through architectural inconsistencies, security vulnerabilities, missing design rationale, and maintenance overhead [22]. These critical accounts do not invalidate vibe coding, but they support the need for explicit responsibility, verification, tool awareness, design guidance, and stopping criteria.

Community and tool documentation also matter because the field is developing in public. Vendor documentation already turns some informal practices into tool-level mechanisms: project memory, persistent instruction files, visible context, and rule files are becoming part of everyday AI-assisted programming environments [23, 24]. This supports **Tool-Aware Vibe Coding**: disciplined use depends not only on stronger models, but also on stable context and review surfaces.

At the same time, practitioner discussion warns that unstructured vibe coding can obscure responsibility and increase maintenance risk, so the proposed principles should be read as a compact, provisional stabilization of recurring public recommendations rather than as the closure of the debate [25].

Another important point is terminological drift. The same underlying intuition appears under several names: "spec-driven" work, "staying the architect," "single source of truth," "context rot," or "small safe steps." A stabilizing article should not preserve every temporary label. It should identify the more durable commitments beneath them. That is why the present framework sometimes generalizes beyond the wording that appears in individual threads.

The same logic explains why this article records search expressions rather than a catalog of forum URLs. A static paper is a poor place to preserve a long list of unstable links into a fast-moving social discussion. Search expressions are more durable. They help later readers rediscover the debate in its then-current form, even if the original ranking of posts, exact thread titles, or platform-level visibility has already changed.

The main gap is now clear. The literature already identifies the risks of vibe coding, and the community has already started to formulate local remedies, but the field still lacks a compact normative vocabulary that can connect these scattered insights to software engineering language. This paper addresses that gap by proposing a small principle set that is broad enough to summarize the current state, yet simple enough to be teachable and discussable.

This paper is therefore a synthesis paper. It combines peer-reviewed and preprint literature, earlier software-engineering principles, and fast-moving community practice. Its burden of proof is analytical rather than statistical. The questions are: what tensions require principle-level guidance, how can these tensions be consolidated, and how can the resulting framework remain compact enough for practical use?

3 What Vibe Coding Changes

In this paper, *disciplined vibe coding* means prompt-based AI-assisted programming in which fast generation is constrained by explicit specification, human ownership, review, verification, and task-level completion criteria.

The most important change is not only faster code production, but redistributed cognition. The developer increasingly acts as an orchestrator: framing the task, constraining the solution space, evaluating model output, and deciding whether the generated result is acceptable. Skill is therefore expressed through task framing, prompt design, review discipline, tool use, and verification choices.

This orchestration can feel lighter than implementation. That feeling is part of the attraction of vibe coding, especially in exploratory work and education. A short prompt can produce a visible feature or a plausible fix in seconds. Quick success strengthens motivation, motivation invites further experimentation, and the next success often arrives quickly again. A useful framework should therefore not treat flow as a trivial bonus. Flow is one of the main goods that disciplined vibe coding should preserve.

The problem is that short-term flow can be mistaken for long-term health. A project may feel smooth today while silently becoming harder to understand and modify tomorrow. The main risks can be stated in practical terms:

- **Decreasing sense of responsibility:** Fast generation can make it easier to accept generated code without understanding, reviewing, or owning its consequences.
- **Mitigates the risk of problems caused by inappropriate AI-tool choice and token-budget constraints:** Code quality depends on the chosen model, its training data, context window, and available token budget. Lower-cost configurations can increase error risk, while stronger or longer-context configurations may improve results but raise cost and latency.
- **Loss of user intent:** When goals, constraints, assumptions, or non-functional requirements are not explicit, the generated code may implement an assumed task instead of the intended one.

- **False confidence:** Quick visible progress, plausible code, or passing local checks can feel like sufficient evidence before the result has been properly reviewed and verified.
- **Ad hoc prompting and erosion of engineering principles:** Repeated prompt-by-prompt changes can bypass design principles, patterns, tests, and maintainability constraints.
- **Aimless exploration:** Without an explicit Definition of Done, prompting can continue through unrelated additions, endless refinements, and over-vibing.

When a serious change request arrives, these omissions return as maintainability pressure. The aim of principles is therefore realistic. They do not promise that a growing system will never need refactoring. They try to prolong productive flow, delay the complexity wall, and keep later refactoring possible rather than catastrophic.

4 The Six Vibe Coding Principles

The six principles are presented as semantic commitments with prompt-level consequences. The semantic level states the enduring idea. The prompt level translates it into day-to-day AI-assisted work.

Table 1 summarizes the mapping and shows uneven coverage of the proposed principles within the collected recommendations.

Principle	Core claim	Primarily mitigates
1. Responsible Vibe Coding	AI can redistribute labor, but responsibility remains with the human developer.	Decreasing sense of responsibility
2. Tool-Aware Vibe Coding	Good results depend on model, mode, available context, and token budget.	Problems caused by inappropriate AI-tool choice and token-budget constraints
3. Specification-First	Generation should start from explicit goals, constraints, and assumptions.	Loss of user intent
4. Plan–Guide–Verify	Work in small cycles: plan intent, guide generation, verify result.	False confidence
5. Rule of Principle-Guided Prompts	Carry principles, patterns, and practices into the prompt layer.	Ad hoc prompting and erosion of engineering principles
6. Definition of Done	Use explicit task-level completion criteria.	Aimless exploration

Table 1. Overview of the six Vibe Coding Principles.

4.1 Community gray literature and synthesis method

This subsection explains how the six principles were synthesized from community gray literature, then the following subsections motivate each of them individually.

A large part of the current discussion on vibe coding principles lives in forums, tool communities, and expert posts. These sources are useful, but their URLs are unstable and the discussion moves quickly. For that reason, this paper does not try to archive threads one by one. Instead, it follows a rediscovery logic. Over a rolling three-month window, the discussion was repeatedly re-entered through queries such as

- `vibe coding best practices`,
- `vibe coding spec-driven development`,
- `vibe coding context rot`, and
- `staying the architect vibe coding`.

The recurring recommendations were collected, compared with the author's own experiences, expanded into 12 community recommendations, then merged and pruned into the six principles proposed in this paper.

The 12 collected vibe-coding community recommendations and their mappings to the six proposed final principles are as follows. The mapping is many-to-many: primary mappings indicate the main consolidation path, while secondary mappings indicate supporting relationships.

1. **Principle of Explicit Intent: "If the intent is not explicit, the code is accidental."** The prompt must clearly state: 1. what the code should do, 2. what the code must **not** do, 3. what assumptions it makes. In vibe coding, hallucinations often arise from *implicit assumptions*.
   Mapped primarily to principle **3. Specification-First: Generation should start from explicit goals, constraints, and assumptions**; secondarily to **6. Definition of Done: Use explicit task-level completion criteria**.
2. **Principle of Input-Space Awareness: "Working for one input is not working."** The generated code must **not be considered correct** until: 1. the relevant input space has been mapped, 2. edge cases have been identified. This principle **supports testing**, but is not reducible to it.
   Mapped to: **3. Specification-First** and **4. Plan–Guide–Verify: Work in small cycles: plan intent, guide generation, verify result**; secondarily to **6. Definition of Done**.
3. **Principle of Test as a Confidence Measure: "Tests do not prove correctness, they reduce uncertainty."** A unit test is: 1. **not a proof**, 2. but a **mechanism for reducing uncertainty**. This is an important rhetorical shift: it positions testing as engineering risk management.
   Mapped to: **4. Plan–Guide–Verify**; secondarily to **6. Definition of Done**.
4. **Principle of Prompt-Level SOLID: "Be SOLID before the code exists."** The **SOLID principles already appear in the prompt**: 1. Single Responsibility → what the code *does not* do. 2. Open/Closed → what changes it must withstand. 3. Dependency Inversion → what abstractions it should use. Novelty: **design principles → prompt principles**.

Mapped to: **5. Rule of Principle-Guided Prompts: Carry principles, patterns, and practices into the prompt layer**; secondarily to **3. Specification-First**.

5. **Principle of Minimal Generation: "The greenest code is the code you did not generate."** AI usage has a **cost**: 1. computational resources (energy, water, . . .), 2. time, 3. mental overhead. Provocative, but important: **not all code should be generated**, even if it can be.
   Mapped to: **6. Definition of Done**; secondarily to **1. Responsible Vibe Coding: AI can redistribute labor, but responsibility remains with the human developer**.
6. **Principle of Traceability: "If you cannot explain why the code exists, you should not trust it."** The developer must be able to: 1. trace individual parts of the code back to the prompts, 2. justify the structure of the solution. Reddit phrase: *"I don't know why this works."*
   Mapped to: **2. Tool-Aware Vibe Coding: Good results depend on model, mode, available context, and token budget** and **1. Responsible Vibe Coding**; secondarily to **5. Rule of Principle-Guided Prompts**.
7. **Principle of Progressive Commitment: "Do not fully trust the first generation."** During vibe coding: 1. first, *low-commitment* code is produced, 2. then it gradually becomes "production-ready." This goes against the "copy–paste–ship" mentality.
   Mapped to: **4. Plan–Guide–Verify**; secondarily to **2. Tool-Aware Vibe Coding** and **6. Definition of Done**.
8. **Principle of Human-in-the-Loop Responsibility: "The AI generates, the human owns."** Responsibility **cannot be delegated**: 1. for bugs, 2. for security issues, 3. for unexpected behavior. Legally trivial, but often forgotten in engineering practice.
   Mapped to: **1. Responsible Vibe Coding**.
9. **Principle of Behavioral Validation over Surface Correctness: "Readable code can still be wrong."** In vibe coding, code is often: 1. beautiful, 2. idiomatic, 3. but behaviorally incorrect. This justifies: A. testing, B. critical thinking, C. an input-focused approach.
   Mapped to: **4. Plan–Guide–Verify**; secondarily to **3. Specification-First** and **6. Definition of Done**.
10. **Principle of Feedback Loop Tightness: "Fast feedback beats clever prompts."** A short cycle: generate → test → reflect → refine overrides increasingly long and clever prompts. Reddit phrase: *"a prompt that is too long = loss of control."*
    Mapped to: **4. Plan–Guide–Verify**; secondarily to **2. Tool-Aware Vibe Coding**.
11. **Principle of Degradation Awareness: "AI-generated code degrades silently."** During refactoring and extension: 1. AI-generated structure can degrade quickly, 2. if it is not protected by tests, design constraints, review, and stable tool context. This is a long-term maintainability principle.
    Mapped to: **2. Tool-Aware Vibe Coding** and **5. Rule of Principle-Guided Prompts**; secondarily to **4. Plan–Guide–Verify**.

12. **Principle of Educational Honesty: "If you cannot implement it without AI, you do not understand it yet."** Provocative, but strong: AI only temporarily conceals the lack of it.
Mapped to: **1. Responsible Vibe Coding**.

Table 2 summarizes the mapping and shows uneven community support across the proposed principles:
Principle 4 – Plan–Guide–Verify: 5 primary + 1 secondary;
Principle 1 – Responsible Vibe Coding: 3 primary + 1 secondary;
Principle 2 – Tool-Aware Vibe Coding: 2 primary + 2 secondary;
Principle 3 – Specification-First: 2 primary + 2 secondary;
Principle 5 – Rule of Principle-Guided Prompts: 2 primary + 1 secondary;
Principle 6 – Definition of Done: 1 primary + 5 secondary.

Recommendation	Primary Mapping	Secondary Mapping
1	3	6
2	3; 4	6
3	4	6
4	5	3
5	6	1
6	1; 2	5
7	4	2; 6
8	1	no secondary
9	4	3; 6
10	4	2
11	2; 5	4
12	1	no secondary

Table 2. Summary of the mapping from 12 recommendations to 6 final principles.

In the following subsections, each principle is presented at the semantic and prompt levels, followed by a short explanation and the main risk it mitigates.

4.2 Responsible Vibe Coding

Semantic level: Vibe coding does not transfer responsibility: the human developer remains responsible for correctness, safety, consequences, and the responsible use of tools.

Prompt level: Prompt for options, surface risks and trade-offs, and keep human approval for decisions that matter.

This principle keeps accountability on the human side. AI may produce more code, but it does not become the legal, ethical, or professional owner of the result. In education, students should still be able to explain what they submit. In

professional settings, teams remain answerable for functionality, security, privacy, performance, and maintainability.

Mitigates the risk of decreasing sense of responsibility.

4.3 Tool-Aware Vibe Coding

Semantic level: Good vibe coding depends on strong tools and on awareness of tool context.

Prompt level: Understand the model, mode, visible files, memory, agentic powers, context window, token budget, and known limitations before trusting the output.

A plain chat window, an IDE assistant, and a project-wide composer create different working conditions. Discipline must match the tool. If the tool supports persistent rules, project memory, structured diffs, or test execution, good practice should use them. If it does not, the missing structure must be recreated through notes, checklists, or more careful prompting. Tool awareness also includes resource awareness: model size, training background, context window, and available token budget influence both code quality and cost. A weaker or lower-cost setup may increase the risk of errors, while a stronger or longer-context setup may improve results but also increase cost and latency.

Mitigates the risk of problems caused by inappropriate AI-tool choice and token-budget constraints.

4.4 Specification-First

Semantic level: Vibe coding does not mean specification-free coding; explicit specification matters more than ever.

Prompt level: Base every meaningful prompt on an explicit specification, and update the specification before updating the code.

Fast generation magnifies vague intent. When goals, constraints, and assumptions stay implicit, the model fills the gaps and the rationale behind the code disappears. The specification may be lightweight: a short change request (CR), contract, acceptance description, or requirement specification. What matters is that intent survives beyond the most recent prompt.

Mitigates the risk of loss of user intent.

4.5 Plan–Guide–Verify

Semantic level: Work in small cycles: plan the intent, guide the generation, and verify the result.

Prompt level: Use separate prompts or steps for planning, code generation, and verification; do not collapse them into blind acceptance.

Planning externalizes intent. Guiding translates that intent into prompts and constraints. Verification asks whether the result actually satisfies the task. Verification may be lightweight or strong, but some verification step should stand

between generation and acceptance. This principle protects flow by interrupting it only where unchecked momentum is most likely to create later pain.

Mitigates the risk of false confidence.

4.6 Rule of Principle-Guided Prompts

Semantic level: Use design principles (the first P), best practices (the second P), and design patterns (the third P) as constructive guidance rather than as dogma, i.e., stay on the bright side of PPP.

Prompt level: Ask the AI to preserve design principles such as SOLID, apply suitable patterns such as strategy, and follow practices such as test-driven development (TDD) where appropriate.

Once natural language becomes the main interface to development, engineering discipline must re-enter through the prompt layer. Prompts should therefore be treated as design artifacts, not disposable chat messages. Good prompting protects modularity, extension points, tests, and change resilience, not only for immediate local success.

Mitigates the risk of ad hoc prompting and erosion of engineering principles.

4.7 Definition of Done

Semantic level: Turn functional and non-functional requirements into explicit task-level Definitions of Done that govern completion.

Prompt level: Attach small DoDs to meaningful tasks and use them as completion criteria during verification.

Without explicit completion criteria, vibe coding can continue through endless small improvements and tangent-like prompt loops. A DoD helps the developer stop when the task is genuinely complete and notice when non-functional requirements have been omitted. In shorter form: do not over-vibe, done is done.

Mitigates the risk of aimless exploration.

4.8 A Short Prompt-Based Example

The principles can be illustrated with a small programming task. Suppose a student has to solve some change-request-based (CR-based) tasks described in [26], where the first task can be found at https://github.com/gkusper/progtech-cr0. The task belongs to a CR-based exercise series in which each change request is accompanied by tests, and the next CR should not be started until the current one is solved. In this way we try to simulate frequent changes of a real project.

A weak one-shot vibe-coding prompt might be:

> Generate the production code for CR0: https://github.com/gkusper/progtech-cr0, and make all tests pass.

This prompt is short and convenient, but it hides most of the relevant work. It does not ask the AI to separate understanding from coding, it does not make assumptions explicit, it does not require an implementation plan, and it does not define what should happen if a test fails. A more disciplined interaction should express the cycle behind **Plan–Guide–Verify**. For example, the student could use the following sequence.

> **Prompt 1 – Plan.** Read the CR0 task description and the provided tests: https://github.com/gkusper/progtech-cr0. Do not write code yet. First summarize the required behavior of the checkout program in your own words. List the affected classes or files, the assumptions you make, and the exact Definition of Done (DoD). The task is done only when all CR0 tests pass and no unrelated feature is added.
>
> **Prompt 2 – Guide and generate.** Now implement the smallest change that satisfies CR0 and DoD. Keep the solution simple and readable. Do not change public method names required by the tests. Preserve one clear responsibility per class where possible. After generating the code, briefly explain how the implementation satisfies the requirement and which tests should pass.
>
> **Prompt 3 – Verify and repair.** I ran the CR0 tests. One test still fails: the expected checkout total is different from the actual total. Do not rewrite the whole solution. Identify the most likely cause, propose a minimal fix, and explain which assumption was wrong.
>
> **Prompt 4 – Final check.** All CR0 tests now pass. Review the final solution against the original requirement and DoD. Point out any remaining risk, unnecessary complexity, or refactoring opportunity before I accept the code.

This example shows why one prompt is not enough to express disciplined vibe coding. **Specification-First** appears in the first prompt, where the requirement, assumptions, affected files, and Definition of Done are made explicit before coding. **Plan–Guide–Verify** appears in the separation between planning, guided generation, test-based repair, and final review. **Definition of Done** appears as an explicit stopping rule: the task is complete only when the stated CR0 requirement is implemented, all CR0 tests pass, and no unrelated feature is added. **Rule of Principle-Guided Prompts** appears in the request for a simple, readable solution with stable public method names and clear responsibilities: Preserve one clear responsibility per class where possible. **Tool-Aware Vibe Coding** is less visible here, because the AI tool and its capabilities are not described explicitly. Still, the prompts assume that the tool can access the source code from the repository URL, and the student relies on this capability. Finally, **Responsible Vibe Coding** remains necessary because the student must still run the tests, inspect the generated code, and decide whether the solution can be accepted.

5 Operationalization and Discussion

The principles are not a flat list. Responsible Vibe Coding and Tool-Aware Vibe Coding frame the whole workflow. The first keeps accountability anchored on the human side; the second reminds us that models, IDEs, memory files, and agent modes shape what discipline is realistic. The remaining four principles form the process core. Specification-First gives the work an explicit anchor. Plan–Guide–Verify turns that anchor into an iterative cycle. Principle-guided prompts reconnect the cycle with older engineering wisdom. Definition of Done protects the cycle from aimless exploration.

This structure also clarifies the PPP vocabulary: Principles, Patterns, and Practices. Vibe coding is best understood as an emerging practice, but it becomes a good practice only when illuminated by principles and supported by patterns. Prompting is therefore not a replacement for software engineering. It is the new surface on which software-engineering commitments must be expressed.

A practical implementation can be lightweight. Responsible Vibe Coding suggests ownership notes and human sign-off checkpoints. Tool-Aware Vibe Coding suggests a short declaration of the tool context and its limits. Specification-First suggests compact change requests or contract blocks. Plan–Guide–Verify suggests preserving the plan, generated diff, and verification evidence. Principle-guided prompts suggest reusable templates. Definition of Done suggests small task-level checklists.

The framework also explains a maturity path. Beginners often notice tools first, because tools are the most visible source of immediate gains. The next discovery is that strong tools are not enough: explicit specification is the real anchor of productive work. More experienced developers then rediscover older engineering values, because prompts that ignore design principles create code that becomes painful to change. From this perspective, Tool-Aware Vibe Coding is the visible entry point, Specification-First is the universal stabilizer, and Principle-Guided Prompts are where deeper engineering discipline returns.

The framework has limits. It is conceptual, not yet a tested method. Different domains will need different operationalizations, and the sixth principle may still be renamed or refined. The paper also does not settle whether vibe coding is best classified as a paradigm, method, or practice. At the present stage, a more useful contribution is to make disciplined vibe coding discussable in a stable way.

5.1 Preliminary Classroom Evidence

An earlier classroom study, *We Are Not Afraid of the Wolf!*, did not measure the six principles directly, but it provides empirical motivation for the present framework [27]. In that study, AI-assisted programming performance was not interpreted as a replacement for programming knowledge; rather, the results suggested that students still needed to review, debug, and understand AI-generated code. This earlier result is consistent with the need for principles such as **Responsible Vibe Coding** and **Plan–Guide–Verify**.

As preliminary classroom evidence for the present framework, we refer to the dataset *Structured Vibe Coding – We Are Not Afraid of the Wolf! – 2026* [28], a follow-up data collection for [27]. This dataset was not collected exclusively for the present conceptual paper. It supports a broader research program on structured AI-assisted programming in education. Nevertheless, it contains both principle-oriented questionnaire items and indirect workflow indicators that are relevant for the present framework.

The primary comparison dataset contains 98 responses: 60 structured workflow cases, called notebook-oriented in the dataset, and 38 unstructured workflow cases. The classroom study was conducted in two university contexts, Hungary and Romania. However, the cleaned row-level dataset does not contain country or institution information; therefore, no country-level comparison is reported in this paper.

Table 3 shows how the six principles were operationalized in the questionnaire. For each principle, four 5-point Likert items were used. The reported mean is the average of the four corresponding item columns over the 98 responses.

The strongest principle-oriented support appears for **Definition of Done** and **Specification-First**. **Plan–Guide–Verify**, **Tool-Aware Vibe Coding**, and **Responsible Vibe Coding** are also above the neutral midpoint. **Rule of Principle-Guided Prompts** receives weaker direct support, which is plausible in a student sample because it requires more explicit software-engineering awareness of modularity, extensibility, and design practices.

The indirect workflow indicators in Table 4 also support the framework, especially the process-oriented principles. Participants in the structured workflow group completed more change requests (CRs) per prompt than the unstructured group. They also reported stronger use of tests as evidence and stronger use of passing tests as a quality gate. These results do not validate the framework in a strict empirical sense, but they suggest that structured vibe coding can make specification, verification, and completion criteria more observable in classroom practice.

Tool-related results should be interpreted cautiously. The most frequently used tools were Gemini ($n = 46$), ChatGPT ($n = 25$), and Claude ($n = 20$), followed by Other tools ($n = 4$), GitHub Copilot ($n = 2$), and Cursor ($n = 1$). Descriptively, the Claude group reported higher final change-request success and more completed change requests per prompt than the two larger tool groups. However, the study was not designed as a randomized comparison of tools, and the smaller tool groups make causal interpretation inappropriate. The safer conclusion is that tool choice and tool context do matter, which is consistent with **Tool-Aware Vibe Coding**.

Future validation should examine whether principle-guided vibe coding improves measurable development outcomes. Possible criteria include error rate, test success, maintainability, later rework, code comprehension, student explanation quality, and the calibration of developer confidence against actual correctness. Such validation could be carried out through controlled classroom tasks, expert review of generated solutions, or longitudinal studies of small AI-assisted

Principle	Columns	Mean	Questionnaire items used
1. Responsible Vibe Coding	BD–BG	3.57	(a) I remained responsible for the final solution, even when AI generated parts of the code. (b) I checked AI-generated code before accepting it. (c) I could explain the final solution in my own words. (d) If the AI produced an error, I considered it my task to detect and correct it.
2. Tool-Aware Vibe Coding	BH–BK	3.66	(a) I was aware of the limitations of the AI tool I used. (b) I knew what context the AI tool could or could not see. (c) I adapted my prompts to the capabilities of the AI tool. (d) I did not assume that the AI understood the full project context automatically.
3. Specification-First	BL–BO	3.92	(a) Before asking AI to generate code, I tried to clarify the requirement. (b) I based my prompts on the task description or change request. (c) I updated my understanding of the requirement before modifying the code. (d) A clear specification helped me avoid vague prompts.
4. Plan–Guide–Verify	BP–BS	3.67	(a) I planned what I wanted before asking AI to generate code. (b) I guided the AI with specific constraints or examples. (c) I verified the generated result before accepting it. (d) I used tests, checks, or examples as evidence that the solution worked.
5. Rule of Principle-Guided Prompts	BT–BW	3.14	(a) I asked the AI to keep the code modular. (b) I asked the AI to keep the solution easy to extend. (c) I asked the AI to follow good programming practices. (d) I asked the AI to generate or respect tests.
6. Definition of Done	BX–CA	4.00	(a) I had a clear idea of when the task was finished. (b) Passing the tests was part of my definition of done. (c) I checked whether the solution satisfied all stated requirements. (d) I avoided adding unnecessary extra features before completing the required ones.

Table 3. Principle-oriented questionnaire items and direct support for the six principles in the 2026 structured vibe-coding dataset. Means are on a 1–5 Likert scale.

Indicator	All	Structured	Unstructured
Completed CRs per prompt	0.50	0.60	0.35
Tests helped decide whether the change was successful	4.36	4.53	4.08
Passing tests was treated as a quality gate	4.22	4.35	4.03
Structured vibe coding helped students work more effectively with AI	4.13	4.23	3.97
Preference for unstructured AI chat over structured workflow	2.83	2.55	3.26

Table 4. Selected indirect workflow indicators from the 2026 dataset. Likert items are on a 1–5 scale; for the last row, lower values indicate weaker preference for unstructured AI chat.

projects. The present paper therefore treats the six principles as a conceptual and preliminary classroom-supported framework, not as a fully validated method.

6 Conclusion

Vibe coding has moved beyond novelty. The current discourse already contains enthusiasm, criticism, local remedies, and early attempts at discipline. What it still lacks is a compact normative language that connects these insights to software-engineering practice. This paper proposed six Vibe Coding Principles as such a language.

The framework is intentionally modest. It does not claim that principles remove complexity, replace empirical work, or close the community debate. It offers a short set of lighthouses that help developers, teachers, and researchers preserve responsibility, specification, review, and stopping criteria in a tool-shaped mode of development. Its immediate purpose is practical: prolong productive flow and keep later refactoring possible.

References

1. A. Karpathy, "Vibe Coding," post on X (formerly Twitter), Feb. 2, 2025. Available: https://x.com/karpathy/status/1886192184808149383, Accessed on 30.03.2026.
2. P. Denny, J. Leinonen, J. Prather, A. Luxton-Reilly, T. Amarouche, B. A. Becker, and B. N. Reeves, "Prompt Problems: A New Programming Exercise for the Generative AI Era," in *Proceedings of the 55th ACM Technical Symposium on Computer Science Education*, Portland, OR, USA, 2024, pp. 296–302, doi: 10.1145/3626252.3630909.
3. C. Kerslake, P. Denny, D. H. Smith IV, J. Prather, J. Leinonen, A. Luxton-Reilly, and S. MacNeil, "Integrating Natural Language Prompting Tasks in Introductory Programming Courses," in *Proceedings of the 2024 ACM Virtual Global Computing Education Conference*, 2024, pp. 88–94, doi: 10.1145/3649165.3690125.
4. B. N. Reeves, J. Prather, P. Denny, J. Leinonen, S. MacNeil, A. Luxton-Reilly, S. M. Nicolajsen, and C. Brabrand, "Prompts First, Precision Later: Reviving the Vision of Natural Language Programming for Computing Education," in *Proceedings of the 25th International Conference on Computing Education Research (Koli Calling 2025)*, 2025, article no. 46, pp. 1–8, doi: 10.1145/3769994.3770039.
5. R. C. Martin, "The Open-Closed Principle," *C++ Report*, Jan. 1996. Available: https://objectmentor.com/resources/articles/ocp.pdf, Accessed on 03.07.2026.
6. R. C. Martin, *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall, 2002.
7. R. C. Martin, "Getting a SOLID Start," Object Mentor blog, Feb. 13, 2009. Available: https://blog.objectmentor.com/articles/2009/02/12/getting-a-solid-start, Accessed on 03.07.2026.
8. C. Meske, T. Hermanns, E. Von der Weiden, K.-U. Loser, and T. Berger, "Vibe Coding as a Reconfiguration of Intent Mediation in Software Development: Definition, Implications, and Research Agenda," *IEEE Access*, vol. 13, pp. 213242–213259, 2025, doi: 10.1109/ACCESS.2025.3645466.

9. A. Sarkar and I. Drosos, "Vibe Coding: Programming through Conversation with Artificial Intelligence," Proceedings of the 36th Annual Conference of the Psychology of Programming Interest Group (PPIG 2025), pp. 87–118, September 2025.
10. R. Sapkota, K. I. Roumeliotis, and M. Karkee, "Vibe Coding vs. Agentic Coding: Fundamentals and Practical Implications of Agentic AI," arXiv:2505.19443, 2025, doi: 10.48550/arXiv.2505.19443.
11. V. Pimenova, S. Fakhoury, C. Bird, M.-A. Storey, and M. Endres, "Good Vibrations? A Qualitative Study of Co-Creation, Communication, Flow, and Trust in Vibe Coding," arXiv:2509.12491, 2025, doi: 10.48550/arXiv.2509.12491.
12. F. Geng, A. Shah, H. Li, N. Mulla, S. Swanson, G. S. Raj, D. Zingaro, and L. Porter, "Exploring Student-AI Interactions in Vibe Coding," *Proceedings of the 28th Australasian Computing Education Conference*, 2026, pp. 45–54, doi: 10.1145/3786228.3786236.
13. D. J. Woo, K. Guo, and Y. Yu, "A Vibe Coding Learning Design to Enhance EFL Students' Talking To, Through, and About AI," arXiv:2509.08854, 2025, doi: 10.48550/arXiv.2509.08854.
14. M. Chow and O. Ng, "From Technology Adopters to Creators: Leveraging AI-Assisted Vibe Coding to Transform Clinical Teaching and Learning," *Medical Teacher*, vol. 47, no. 12, pp. 1927–1929, 2025, doi: 10.1080/0142159X.2025.2488353.
15. J. Cabot, "Vibe Modeling: Challenges and Opportunities," in *Advances in Conceptual Modeling*, Lecture Notes in Computer Science, vol. 16190, Springer, 2025, pp. 105–118, doi: 10.1007/978-3-032-08620-4_7.
16. R. Huang, A. Reyna, S. Lerner, H. Xia, and B. Hempel, "Professional Software Developers Don't Vibe, They Control: AI Agent Use for Coding in 2025," arXiv:2512.14012, 2025, doi: 10.48550/arXiv.2512.14012.
17. S. Wang, "VibeContract: The Missing Quality Assurance Piece in Vibe Coding," arXiv:2603.15691, 2026, doi: 10.48550/arXiv.2603.15691.
18. M. N. U. Shuvo, M. A. Islam, M. M. Hasan, M. Waseem, and P. Abrahamsson, "Context Before Code: An Experience Report on Vibe Coding in Practice," presented at the *1st International Workshop on Vibe Coding and Vibe Researching (VibeX 2026)*, co-located with EASE 2026, Glasgow, United Kingdom, 2026, preprint doi: 10.48550/arXiv.2603.11073.
19. H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, "Asleep at the keyboard? Assessing the security of GitHub Copilot's code contributions," in *Proceedings of the 2022 IEEE Symposium on Security and Privacy (SP)*, 2022, pp. 754–768, doi: 10.1109/SP46214.2022.9833571.
20. J. Prather, B. N. Reeves, J. Leinonen, S. MacNeil, A. S. Randrianasolo, B. A. Becker, B. Kimmel, J. Wright, and B. Briggs, "The Widening Gap: The Benefits and Harms of Generative AI for Novice Programmers," in *Proceedings of the 2024 ACM Conference on International Computing Education Research (ICER '24)*, 2024, volume 1, pp. 469–486, doi: 10.1145/3632620.3671116.
21. Y.-H. Chou, B. Jiang, Y. W. Chen, M. Weng, V. Jackson, T. Zimmermann, and J. A. Jones, "Building Software by Rolling the Dice: A Qualitative Study of Vibe Coding," to appear in *Proceedings of the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2026)*, 2026, preprint doi: 10.48550/arXiv.2512.22418.
22. M. Waseem, A. Ahmad, K.-K. Kemell, J. Rasku, S. Lahti, K. Mäkelä, and P. Abrahamsson, "Vibe Coding in Practice: Flow, Technical Debt, and Guidelines for Sustainable Use," arXiv:2512.11922, 2025, doi: 10.48550/arXiv.2512.11922.
23. Anthropic, "How Claude remembers your project," Claude Code Docs, 2026. Available: https://code.claude.com/docs/en/memory, Accessed on 30.03.2026.

24. OpenAI, "Custom instructions with AGENTS.md," OpenAI Developers, 2026. Available: https://developers.openai.com/codex/guides/agents-md, Accessed on 30.03.2026.
25. Todd Wardzinski, "The uncomfortable truth about vibe coding," Red Hat Developer, Feb. 17, 2026. Available: https://developers.redhat.com/articles/2026/02/17/uncomfortable-truth-about-vibe-coding, Accessed on 30.03.2026.
26. G. Kusper and C. Szabó, "Vibe Coding in Education," in *2025 International Conference on Emerging eLearning Technologies and Applications (ICETA)*, 2025, pp. 506–511, doi: 10.1109/ICETA67772.2025.11280285.
27. G. Kusper, Gy. I. Mátyás, and T. Balla, "We are not afraid of the wolf! – AI usage attitudes among Hungarian informatics students," *Annales Mathematicae et Informaticae*, vol. 61, pp. 186–201, 2025, doi: 10.33039/ami.2025.10.004.
28. G. Kusper and I. Drămnesc, "Structured Vibe Coding – We Are Not Afraid of the Wolf! – 2026," Dataset, Zenodo, 2026, doi: 10.5281/zenodo.21134392.

Open Access This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial 4.0 International License (http://creativecommons.org/licenses/by-nc/4.0/), which permits any noncommercial use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

