


Ensuring the Maintainability and Supportability of "Vibe-Coded" Software Systems: A Framework for Bridging Intuition and Engineering Rigor

Stephane H Maes¹ 

May 6, 2025

Abstract

This paper addresses the emerging concept of "vibe-coding"—the translation of intuitive feelings or high-level intentions directly into software code. While potentially accelerating development or enabling novel forms of creation, such code inherently risks being opaque, poorly understood, and difficult to maintain.

We propose the Intent-Driven Explicable Architecture (IDEA) framework, a novel approach designed to ensure that vibe-coded software remains maintainable, understandable, and supportable throughout its lifecycle. IDEA integrates techniques for formalizing intuitive inputs, constraining code generation using established software engineering principles, automatically generating explanations linking code to intent, and incorporating rigorous human-in-the-loop validation. We argue that by structuring the translation process and embedding traceability and explicability, IDEA mitigates the primary risks associated with intuition-driven code generation, paving the way for its responsible exploration.

1. Introduction

This paper presents a proposal and analysis of vibe coding complementary to [1].

1.1. Defining "Vibes Coding": Conceptualization and Potential Mechanisms

Software development traditionally relies on detailed specifications and requirements derived through structured analysis. However, emerging paradigms explore more direct pathways from human intent to functional code. "Vibes coding" represents one such paradigm, conceptualized as the process where developers articulate high-level, often subjective or affective goals, colloquially known as the "vibe", which are subsequently translated into executable software artifacts, i.e., code. This translation is often envisioned as being mediated by sophisticated Artificial Intelligence (AI) or Machine Learning (ML) systems, capable of interpreting ambiguous, natural language descriptions or even non-verbal cues and synthesizing corresponding code.

¹ shmaes.physics@gmail.com

The potential appeal of vibes coding lies in its promise to accelerate prototyping, capture nuanced aspects of user experience that are difficult to specify formally, and potentially lower the barrier to entry for individuals without traditional programming expertise. It aligns with a desire to make software creation more intuitive and directly reflective of human feeling, aesthetic or intent. Instead of meticulously defining algorithms and data structures, a developer might specify a desire for an interface that feels calming, or a data processing pipeline that operates with a sense of urgency. It contrasts sharply with, or adds to, conventional requirements, which emphasizes precision, clarity, and testability through formal or semi-formal specifications, and formal acceptance criteria. A core challenge, therefore, resides in the inherent ambiguity and subjectivity of the vibe itself, demanding a robust translation mechanism that can bridge the gap between intuitive feeling and logical code execution. That is in addition to correctly understanding what is to be coded [23]², and doing it correctly in ways that are maintainable and supportable [1], and avoiding troubles due to all forms of hallucinations [2,3].

1.2. The Maintainability Imperative in Software Engineering

Software maintenance, discussed in details in [1,13], encompassing activities like error correction/bug fixes, adaptation to new environments, and implementation of new features, improvements or enhancements, constitutes a significant portion of the total software lifecycle cost. Industry studies consistently show that maintenance can consume upwards of 50-80% of overall software expenditure. Consequently, software maintainability, the ease with which software can be modified, is a critical quality attribute [16]. Poor maintainability leads to increased technical debt, system fragility, higher costs, and slower evolution.

Maintainability is typically characterized by several sub-attributes, including understandability³ (ease of comprehending the code's purpose, structure, and behavior) [14,20], modifiability (ease of making changes), testability (ease of verifying changes), and analyzability (ease of diagnosing deficiencies or identifying parts to modify)[17]. These characteristics are not merely desirable; they are essential for the long-term viability and success of any non-trivial software system, especially when a product. Neglecting maintainability during development inevitably results in systems that are expensive to adapt, risky to change, and ultimately unsustainable. Any novel software development paradigm, including vibes coding, must therefore be evaluated not only on its generative capabilities but also on its compatibility with these fundamental engineering principles [1]. And so far, vibe coding does not systematically produce understandable code, unless if proper processes are followed [1]. These are typically slowing down vibe coding, and hard to follow by non-programmers. That is a major concern, but also a reason why, for the near foreseeable future, the job of developers remains safe, contrary to what is said by the press or too naïve executives [1].

1.3. The Unique Challenge: Why Vibe-Coded Software Resists Maintainability

² Note: While [a23] is focused on preferences, the techniques are relevant to generating explanations aligned with human understanding.

³ As discussed in [a1], this is one of the main concerns with vide coding, or AI coding in general: the code rapidly evolve in ways that cease to be understood by its coder.

While potentially powerful, the very nature of vibes coding introduces significant obstacles to achieving maintainable software. The primary difficulty stems from the large semantic gap between the subjective, often ill-defined "vibe" and the precise, logical structure required for executable code. This gap makes direct translation inherently challenging and prone to interpretation errors.

Furthermore, the reliance on complex generative systems, such as large-scale AI models, introduces the risk of producing code that is opaque or operates as a "black box" (See [1] and references therein). Such systems might generate functionally correct code, which is nonetheless convoluted, non-idiomatic, or difficult for human developers to comprehend. It happens often [1]. The intricate internal logic or vast parameter space of these models can make it hard to understand *why* a particular code structure was chosen, even if it fulfills the functional aspects derived from the vibe.

This leads directly to a critical problem: the original coder, who specified the vibe, and/or subsequent maintainers, may lack a fundamental understanding of the generated code's logic and rationale. Without this understanding, tasks central to maintenance, like debugging, modification, performance tuning, security hardening, become exceedingly difficult and error-prone [13-17,20]. Tracing the high-level, intuitive requirements, i.e., the vibes, to specific implementation details becomes problematic, hindering impact analysis and verification. The lack of clear lineage from intent to code undermines the foundation upon which software maintenance relies. The core issue transcends simple code generation; it becomes a challenge of knowledge translation and preservation across a highly abstract boundary, from human intuition to formal logic. The subjective nature of the input necessitates powerful translation mechanisms, but these very mechanisms risk obscuring the connection between the initial intent and the final artifact, thereby jeopardizing understandability and, consequently, maintainability, and associated support [1].

1.4. Overview of the Proposed IDEA Framework

To address these profound challenges, this paper introduces the Intent-Driven Explicable Architecture (IDEA) framework⁴. IDEA is a multi-stage process designed to systematically integrate maintainability and understandability considerations into the vibe coding paradigm. It aims to structure the translation from intuition to code in a way that preserves intent, enforces engineering discipline, and facilitates human comprehension. The framework comprises four key stages:

1. Vibe Formalization: Capturing and refining the initial intuitive input into a more structured, albeit still high-level, representation.
2. Constrained Generation: Synthesizing code from the formalized vibe, guided by architectural principles, design patterns, and maintainability heuristics.
3. Explication Engine: Automatically generating documentation, justifications, and visualizations that link the generated code back to the original intent and design choices.
4. Human-in-the-Loop Validation: Incorporating rigorous human review and feedback to verify the generated artifacts and refine the overall process.

The contribution of this paper is a structured methodology, embodied by the IDEA framework, along with associated techniques intended to make vibe coding a more viable and responsible approach from a software engineering perspective. By embedding traceability, explicability, and established engineering practices within the

⁴ We will let the reader compare it to VIBE4M [a1]. But they both follow the same principle with the same goals.

intuition-driven development process, IDEA seeks to mitigate the inherent risks and enable the creation of vibe-coded software that is both innovative and sustainable: the code can be tested maintained and supported.

2. Formalizing Intuition: Capturing and Translating the Vibe (Stage 1 of IDEA)

The ambiguity inherent in a vibe is a significant source of difficulty in generating understandable and maintainable code [23]. Directly feeding vague notions like "make it feel trustworthy" into a code generator is unlikely to produce predictable or reliable results. The first stage of the IDEA framework, Vibe Formalization, focuses on bridging this initial semantic gap by eliciting, refining, and representing the intuitive requirements in a more structured manner.

2.1. Techniques for Eliciting and Representing Intuitive Requirements

Effective formalization begins with improved elicitation. Simple free-form text prompts, while easy to provide, often lack the necessary detail and nuance. More sophisticated techniques are required to help users articulate and explore their intuitive goals [12]. These might include:

- **Structured Interviews:** Guided conversations designed to probe the affective, experiential, and qualitative aspects of the desired software. Questions could focus on analogies, metaphors, desired emotional responses in users, or key scenarios illustrating the vibe.
- **Mood Boards and Visual Collages:** Using images, colors, textures, and keywords to create a visual representation of the intended feeling or aesthetic. This can be particularly useful for UI/UX design vibes.
- **User Journey Mapping (Affective Focus):** Mapping user interactions not just by function but by the desired emotional or experiential state at each step.
- **Scenario-Based Elicitation:** Developing concrete scenarios that exemplify the desired vibe in action, helping to ground abstract feelings in specific functional contexts.
- **Biofeedback Interfaces (Exploratory):** Potentially using physiological data (e.g., camera input, and other multimodal input including for example galvanic skin response, heart rate variability) as an auxiliary input channel to capture implicit affective states, although this remains highly speculative and requires significant research [4-7].

The goal of these techniques is not to eliminate subjectivity but to refine and disambiguate the vibes, encouraging the user to move from a vague feeling towards a richer, more detailed, albeit still high-level, description of their intent. This process draws inspiration from requirements elicitation practices used in user experience design and human-computer interaction, adapted for the specific challenge of capturing "vibes" destined for code generation.

2.2. Intermediate Representations: Bridging Subjectivity and Code Logic

Once elicited, the refined vibe needs to be captured in a format suitable for machine processing without losing its essential meaning. This necessitates an Intermediate Representation (IR). The IR acts as a crucial artifact within the IDEA framework, serving as the formalized contract between the user's intuition and the code generation process. Its quality and expressiveness directly impact the feasibility of generating maintainable and understandable code later [16,23].

The IR must balance expressiveness (capturing subjective qualities, from multimodal input) with formality (enabling automated processing). Potential formats for such an IR include:

- Enriched Feature Models: Standard feature models augmented with tags, or attributes, representing non-functional qualities, desired affects, or experiential goals associated with features.
- Goal-Oriented Requirement Languages (Adapted): Extending existing goal modeling languages (like GRL or i*) to incorporate primitives for representing affective states, user experience qualities, or aesthetic properties alongside functional goals.
- Domain-Specific Languages (DSLs): Creating specialized languages tailored to specific domains where vibe coding might be particularly relevant (e.g., game design, interactive art installations, specific types of user interfaces). A DSL could offer constructs specifically designed to express common vibes within that domain.
- Structured Annotations on Prototypes: Allowing users to annotate wireframes, mockups, or prototypes with specific "vibe" descriptors linked to UI elements or interaction flows.

Designing effective IRs is a significant challenge. The IR must be capable of representing concepts like "playfulness," "serenity," "professionalism," or "urgency" in a way that can be systematically mapped to concrete software characteristics. It forms the backbone for traceability, ensuring that the initial intent is not lost during the translation to code. A well-defined IR transforms the highly ambiguous input into a structured specification, making the subsequent stages of constrained generation and explication feasible.

2.3. Semantic Mapping and Intent Modeling

The final part of Vibe Formalization involves establishing connections between the elements captured in the IR and potential software constructs, architectural choices, design patterns, or quality attributes. This requires building a semantic model or knowledge base that understands these mappings [23]. For example, the model might associate:

- A secure vibe IR element with the selection of specific authentication protocols, encryption libraries, or input validation patterns.
- A responsive and fluid vibe with asynchronous processing, specific UI animation frameworks, or performance optimization techniques.
- A minimalist aesthetic vibe with particular layout principles, color palettes, or the avoidance of excessive ornamentation in the UI.

- A robust and reliable vibe with fault tolerance patterns (e.g., circuit breakers, retries) or extensive logging mechanisms.

This mapping process could leverage ontologies, knowledge graphs, or rule-based systems that encapsulate domain knowledge and software engineering best practices. The semantic model essentially translates the formalized intent (in the IR) into actionable guidance for the code generation stage. It defines how the desired qualities are expected to manifest in the software's architecture and implementation. The accuracy and richness of this semantic mapping are critical for ensuring that the generated code not only functions correctly but also genuinely reflects the original vibe in a meaningful way.

3. Generating Maintainable Code from Intuitive Inputs (Stage 2 of IDEA)

Having formalized the intuitive requirements into an Intermediate Representation (IR) and established semantic mappings, the second stage of the IDEA framework focuses on the code generation process itself. A core tenet of IDEA is that this generation must be constrained and guided by established software engineering principles to ensure the resulting code is maintainable. Unfettered generation, even if functionally aligned with the vibe, risks producing opaque and unmanageable systems [1].

3.1. Architectural Constraints and Pattern Enforcement during Generation

The generation process should not operate in a vacuum; it must adhere to sound architectural principles. IDEA proposes mechanisms to enforce architectural constraints derived from the IR and the semantic mapping stage, as well as general software engineering best practices. This means the AI or other generative system is not free to produce any code that satisfies the functional interpretation of the vibe, but must do so within predefined structural boundaries.

Examples of such constraints include:

- Enforcing Layered Architectures: Ensuring separation of concerns (e.g., presentation, business logic, data access).
- Mandating Specific Architectural Styles: Requiring adherence to styles like microservices, event-driven architecture, or model-view-controller (MVC), if deemed appropriate based on the vibe or application domain.
- Applying Relevant Design Patterns: Automatically incorporating established design patterns that promote flexibility, reusability, and understandability [18,21-23]. For instance, a vibe emphasizing adaptability might trigger the use of Strategy or Observer patterns, while a vibe focused on object creation might leverage Factory or Builder patterns. The semantic mapping (Stage 1) should inform the selection of appropriate patterns.

By embedding these constraints, the framework guides the generator towards producing code with known structural properties conducive to maintenance [13]. This proactive structuring helps prevent the emergence of

monolithic, tangled codebases often associated with unguided generation, directly addressing maintainability concerns related to modifiability and analyzability.

3.2. Injecting Readability and Modularity Principles

Maintainability extends beyond high-level architecture to the quality of the code itself. Code that is difficult to read or tightly coupled is inherently difficult to maintain, regardless of the overarching architecture [1]. Therefore, the generation process within IDEA must explicitly prioritize code readability and modularity.

Techniques to achieve this include:

- **Style Guide Enforcement:** Automatically formatting the generated code according to predefined style conventions (e.g., naming conventions, indentation, commenting guidelines).
- **Automated Refactoring:** Applying refactoring techniques (e.g., extract method, introduce parameter object) either during or immediately after generation to improve code structure.
- **Complexity Management:** Using metrics [19] like cyclomatic complexity or cognitive complexity to monitor and control the complexity of generated functions or methods, potentially guiding the generator to produce simpler, more understandable units of code.
- **Modularity Metrics:** Employing metrics related to coupling (dependencies between modules) and cohesion (relatedness of elements within a module) as objectives or constraints during generation, aiming for low coupling and high cohesion.
- **Abstraction Level Control:** Ensuring that generated code utilizes appropriate levels of abstraction, hiding implementation details behind well-defined interfaces.

Optimizing for these micro-level quality attributes is crucial because AI generators, particularly those based on large language models, might prioritize functional correctness or token efficiency over human readability and established coding idioms. Explicitly incorporating readability and modularity as objectives or constraints within the generation process counteracts this tendency, ensuring the output is amenable to human understanding and modification.

3.3. The Role of AI/ML in Structured Code Synthesis

AI and ML models are central to the vision of translating abstract vibes into concrete code [8,21,22]. Large Language Models (LLMs) trained on vast code corpora have demonstrated impressive capabilities in code generation, translation, and completion [10,11]. However, for the IDEA framework, the key is not just generative power but controllability and predictability.

While general-purpose LLMs can be used, techniques that allow for more explicit control over the output are preferred:

- Neuro-Symbolic AI: Combining neural networks (for pattern recognition and generation) with symbolic reasoning (for enforcing rules and constraints) could offer a way to generate code that adheres strictly to architectural and stylistic requirements.
- Grammar-Guided Generation: Using formal grammars (representing the target programming language and potentially architectural constraints) to guide the generation process, ensuring syntactically correct and structurally sound output.
- Reinforcement Learning with Maintainability Rewards: Training generative models using reinforcement learning where the reward function incorporates not only functional correctness but also metrics related to maintainability (e.g., low complexity, adherence to patterns, style guide compliance) [10,11,19]. [11] also shows the need to train on secure code.
- Retrieval-Augmented Generation: Combining generative models with retrieval systems that can pull relevant, high-quality code snippets or patterns from a curated library, promoting reuse and adherence to best practices.
- Translation engines as in [8].

The goal is to leverage AI's ability to bridge the vibe-to-code gap while mitigating the risks associated with unconstrained, black-box generation. The AI acts as a powerful synthesis engine, but one operating within the guardrails established by the formalized intent (IR), semantic mappings, architectural constraints, and maintainability heuristics defined in the IDEA framework.

4. The Explication Engine: Automated Documentation and Justification (Stage 3 of IDEA)

A critical failure mode of vibe coding is the loss of rationale—the "why" behind the generated code. Even if the code is well-structured and functionally correct, maintainers struggle if they cannot understand the connection between the original intent and the resulting implementation. The third stage of the IDEA framework, the Explication Engine, directly addresses this by automatically generating documentation and justifications that illuminate the vibe-to-code transformation process. This stage leverages the structured information flow established in the previous stages (Vibe Formalization and Constrained Generation).

4.1. Generating Code Annotations Linked to Initial Intent

The Explication Engine aims to embed traceability directly within the codebase. It should automatically generate code comments or annotations that explicitly link specific code blocks, functions, classes, or modules back to the corresponding elements in the Intermediate Representation (IR).

For example, a block of code implementing an encryption routine might be annotated with a comment like:

```
// Implements: IR Component 'User Data Confidentiality' (derived from Vibe: 'Feeling of safety')
```



```
// Constraint Applied: Use AES-256 encryption as per security policy.
```

Similarly, a UI component responsible for a specific animation might be annotated:

```
// Implements: IR Component 'Playful Transition Effect' (derived from Vibe: 'Engaging and fun UI')
```

```
// Pattern Applied: Fade-in/Slide-up animation sequence.
```

These annotations provide fine-grained traceability, allowing developers to immediately see which part of the original intent a specific piece of code is meant to fulfill. This significantly aids program comprehension and impact analysis during maintenance.

4.2. Producing Human-Readable Descriptions of Generated Logic

Beyond inline annotations, the Explication Engine should generate higher-level, natural language descriptions of the generated code and its underlying rationale. This directly tackles the understandability challenge by providing explanations for maintainers. These descriptions could take several forms:

- **Module Summaries:** For each generated module or major component, a summary describing its purpose, key responsibilities, and how it contributes to fulfilling specific IR elements.
- **Architectural Decision Records (ADRs):** Automatically generated records explaining significant architectural choices made during generation (e.g., why a microservices architecture was chosen, why a specific database technology was selected), linking these choices back to the vibe/IR and applied constraints.
- **Justification Narratives:** Explanations detailing *why* the generator produced a particular implementation. This might involve referencing the semantic mappings used, the design patterns applied, and the constraints that guided the synthesis. For example: "To achieve the 'quick response' vibe (IR: 'Low Latency Interaction'), asynchronous processing was implemented using a message queue for background tasks, following the Asynchronous Request-Reply pattern."

These generated explanations serve as essential documentation, helping developers understand not just *what* the code does, but *why* it does it that way, thereby improving trust and facilitating maintenance.

4.3. Visualizing the Vibe-to-Code Transformation Path

Understanding complex systems often benefits from visual aids. The Explication Engine could include tools or generate artifacts that visualize the entire transformation pipeline within the IDEA framework:

- **Traceability Graphs:** Visual graphs showing the links from initial vibe elicitation artifacts (e.g., mood boards, interview notes) to IR components, then to semantic mappings, applied constraints, architectural choices, generated modules/classes, and finally specific code snippets.
- **Interactive Dashboards:** Web-based dashboards enabling developers to explore the relationships between vibes, IR elements, code components, and generated explanations dynamically. Users could click on an IR element to see all related code and justifications, or click on a code module to trace it back to the original intent.

Such visualizations provide a holistic overview, making the complex journey from subjective intuition to concrete code more transparent and navigable for developers tasked with understanding or modifying the system.

4.4. Proposed Table: Vibe-to-Code Traceability and Justification Matrix

A key output of the Explication Engine is a structured summary of the traceability and justification links. This can be effectively represented in a tabular format, serving as a central reference artifact for maintainers.

Vibe Element / IR Component	Mapped Architectural/Design Choice	Generated Code Snippet(s) / Module(s)	Generated Justification	Verification Status
Vibe: "Feeling of safety"	Authentication Module (OAuth 2.0)	auth.py, TokenService	Implements secure user login via industry-standard OAuth 2.0 to ensure data access control, fulfilling safety requirement.	Reviewed, Tested
IR: User Data Confidentiality	AES-256 Encryption Utility	crypto.utils.encrypt_data	Encrypts sensitive user data at rest using AES-256, adhering to security constraint derived from 'safety' vibe.	Reviewed, Tested
Vibe: "Need for quick response"	Asynchronous Task Queue (Celery)	tasks.py, OrderProcessor	Offloads long-running order processing to background queue to maintain UI responsiveness, fulfilling 'quick response'.	Reviewed, Tested
IR: Low Latency Interaction	API Gateway Caching	api_gateway_config.yaml	Caches frequently accessed read-only data at the	Reviewed

			API gateway edge to minimize latency for user queries.	
Vibe: "Playful UI transition"	Specific Animation Library (React Spring)	UserProfile.jsx	Uses spring physics animations for profile card transitions, creating an engaging and 'playful' user experience.	Reviewed
IR: Engaging Visual Feedback	Micro-interaction on Button Click	Button.css, Button.jsx	Provides subtle scaling and color change on button press, enhancing perceived interactivity as per 'playful' vibe.	Not Reviewed

Table 1: Vibe-to-Code Traceability and Justification Matrix (Example)

This matrix in table 1 directly addresses the core problem of the disconnect between the initial subjective vibe and the final concrete code, which leads to poor understanding and subsequent maintainability challenges. To bridge this gap, explicit traceability and justification are paramount. The table provides a structured format capturing the entire chain:

Vibe -> (IR) -> Design Choice -> Code Artifact -> Justification -> Verification Status.

It moves beyond a simple feature list to show *how* a specific feeling or intention led to a specific implementation and *why* that translation is considered valid by the framework. This artifact serves as a crucial reference for developers, maintainers, and testers, directly supporting understandability (by explaining the 'why'), modifiability (by clarifying the impact of changes), and testability (by linking tests back to the original intent). It operationalizes the output of the Explication Engine, making the generated knowledge accessible and actionable.

4.5. AI Tools Explaining (Any) Code

Although, we are skeptical of the reliability and universality of such tools, but we try to remain open minded, and such tools certainly can only help. [9] is an example of the fantastic (marketing) promises, which may help greatly

with the maintainability and understandability challenges identified here and in [1]. When it works, it would help transfer knowledge and understand the code.

We would appreciate feedback on the value and usability of a tool like [9], from readers who would have used them with complex enterprise software-type of code.

5. Human-in-the-Loop: Verification, Validation, and Refinement (Stage 4 of IDEA)

While the IDEA framework aims to automate and structure the generation and explanation of vibe-coded software, human oversight remains indispensable. Automation can introduce errors, misinterpret nuances, or produce suboptimal solutions. The fourth stage of IDEA, Human-in-the-Loop (HITL) Validation, emphasizes the critical role of human developers in reviewing, verifying, and refining the outputs of the automated stages, ensuring quality, trustworthiness, and alignment with the true intent.

5.1. Protocols for Code Review and Acceptance of Generated Code

Generated code, even when accompanied by explanations, must undergo rigorous review by human developers. This review process needs to be adapted for the context of AI-generated artifacts. Key aspects, in addition to the considerations of [1], include:

- **Reviewing Code and Explanations Together:** Developers should assess not only the functional correctness and quality (readability, modularity) of the code, but also the accuracy, clarity, and completeness of the generated explanations and justifications. Is the explanation consistent with the code? Does it accurately reflect the linkage back to the IR and vibe?
- **Assessing Fidelity to Intent:** The review must evaluate whether the generated code, guided by the IR and constraints, truly captures the essence of the original vibe. This requires developers to understand the initial vibe (potentially through the elicitation artifacts and IR), and judge the appropriateness of the implementation.
- **Verifying Constraint Adherence:** Reviewers must confirm that the generated code respects the architectural constraints, design patterns, and style guidelines mandated during the Constrained Generation stage.
- **Utilizing Static Analysis Tools:** Supplementing human review with automated static analysis tools to check for potential bugs, security vulnerabilities, and deviations from coding standards.
- **Structured Acceptance Criteria:** Defining clear criteria for accepting generated code, covering functionality, maintainability attributes, explanation quality, and fidelity to intent.

This structured review process ensures that the generated artifacts meet quality standards and builds confidence in the system. It emphasizes that AI generation is a tool to assist developers, not replace their critical judgment and responsibility for the final product. Effective human-AI collaboration requires calibrating trust based on verification.

5.2. Feedback Mechanisms for Improving Generation Accuracy and Maintainability

The HITL review stage is not merely a quality gate. It is also a vital source of feedback for improving the entire IDEA framework. Mechanisms should be in place to capture and utilize this feedback systematically:

- **Identifying Generation Weaknesses:** When reviewers find issues (e.g., confusing code, inaccurate explanations, misapplied patterns, poor vibe-to-code mapping), this feedback should be logged and analyzed.
- **Fine-tuning the AI Generator [10,11]:** Feedback can be used to retrain or fine-tune the AI code generation models, potentially using techniques like reinforcement learning from human feedback (RLHF) to improve output quality, constraint adherence, and explanation generation.
- **Refining the IR and Semantic Mappings:** If reviews consistently reveal misinterpretations of the vibe, or inappropriate mappings to technical constructs, the IR schema or the semantic knowledge base may need refinement.
- **Adjusting Constraints:** If certain architectural constraints or pattern enforcements lead to awkward, or inefficient code, the constraint models might need adjustment.
- **Improving Elicitation Techniques:** Persistent mismatches between the vibe and the final product might indicate that the initial elicitation techniques need improvement to capture intent more effectively.

This feedback loop creates a continuous improvement cycle, allowing the IDEA framework and its underlying components (AI models, knowledge bases, constraint engines) to learn and adapt over time, leading to better generation quality and more maintainable outcomes.

5.3. Collaborative Models for Understanding and Evolving "Vibe-Coded" Systems

Software development is typically a team activity. The IDEA framework must support collaborative understanding and evolution of vibe-coded systems. The generated artifacts—particularly the IR, the explanations, visualizations, and the traceability matrix—serve as shared points of reference for the team.

- **Shared Understanding:** These artifacts provide a common ground for developers to discuss the system's design, rationale, and connection to the original intent, facilitating onboarding of new team members and collective ownership.
- **Guided Evolution:** When the vibe needs to evolve, or new requirements emerge, the framework provides a structured process. Changes might start with updating the vibe description and the IR, followed by regenerating or refactoring affected parts of the system, guided by the framework's constraints and explanation capabilities. The traceability matrix helps identify the impact of proposed changes.

- **Human-AI Teaming:** The framework fosters a collaborative model where developers work alongside the AI system. Developers focus on specifying intent, defining constraints, reviewing outputs, and providing feedback, while the AI handles the detailed code synthesis and explanation generation.

This collaborative model implies a fundamental shift in the developer's role. Rather than focusing primarily on line-by-line implementations, the developer becomes more of a curator of intent (ensuring the vibe is well-captured in the IR), a verifier of generated artifacts (assessing code, and explanations, against quality standards and intent, generating documentation and metadata/tags), and a refiner of the generation process (providing feedback to improve the system). This higher-level engagement leverages human strengths in understanding context, nuance, and qualitative goals, complementing the AI's strengths in rapid synthesis and pattern application. It also provides job security [1].

6. Ensuring Long-Term Supportability

Maintainability achieved during initial development must be preserved throughout the software's lifecycle. The IDEA framework includes considerations for long-term supportability, addressing testing, versioning, and knowledge management specific to the challenges of intuition-driven code.

6.1. Tailored Testing Strategies for Intuition-Driven Code

Testing vibe-coded software requires approaches that complement traditional functional testing:

- **Vibe-Oriented Usability Testing:** Designing usability tests specifically to evaluate whether the final application actually *evokes* the intended vibe or user experience described in the initial requirements and IR. This might involve qualitative user feedback, surveys measuring affective response, or observational studies. Conventional usability testing remains a must.
- **Acceptance Testing Based on IR:** Defining acceptance criteria directly from the Intermediate Representation. Tests should verify that the system exhibits the characteristics and qualities specified in the IR components. Conventional QA remains a must [1].
- **Explanation-Driven Testing:** Using the generated explanations to guide test case design, ensuring that the implemented logic aligns with the documented rationale.
- **Traceability-Based Regression Testing:** When changes are made, leveraging the Vibe-to-Code Traceability Matrix (Table 1) to identify potentially affected code sections and related IR elements. Regression tests should specifically verify that the intended vibe associated with unchanged parts of the system remains intact and that modifications correctly implement the revised intent.
- **Metamorphic Testing:** Where formal specifications are lacking (due to the vibe's nature), metamorphic testing can be useful. This involves defining expected relationships between different inputs and outputs (metamorphic relations) that should hold if the underlying vibe is correctly implemented.

These tailored strategies ensure that testing evaluates not only functional correctness but also the successful translation and implementation of the subjective, intuitive requirements central to vibes coding.

6.2. Versioning and Traceability of "Vibes" and Code

Effective evolution requires understanding the history of changes. For vibe-coded systems managed by IDEA, version control must extend beyond just the source code. It is crucial to version and link:

- Vibe Descriptions and Elicitation Artifacts: Tracking changes in the high-level intent over time.
- Intermediate Representations (IRs): Versioning the formalized intent captured in the IR.
- Semantic Mappings and Knowledge Bases: Recording changes in how IR elements are mapped to technical constructs.
- Constraint Definitions: Versioning the architectural rules, pattern libraries, and style guides used during generation.
- Generated Code: Standard source code versioning.
- Generated Explanations and Traceability Matrices: Versioning the documentation and traceability links corresponding to each code version.

In agreement with [1], maintaining this comprehensive version history with explicit links between artifacts (e.g., this code version was generated from this IR version using these constraints and produced these explanations) is essential for understanding system evolution, debugging issues introduced over time, and safely regenerating or refactoring parts of the system based on updated vibes or requirements.

6.3. Knowledge Management for Maintaining Vibe-Coded Systems

The rationale and context embedded in the vibe-to-code process are valuable knowledge assets that must be managed effectively to support long-term maintenance, especially as team members change [13]. This involves:

- Centralized Artifact Repository: Storing all IDEA artifacts (vibe descriptions, IRs, semantic models, constraint definitions, generated code, explanations, traceability matrices, review records, test results) in an accessible and organized repository.
- Preserving Review Discussions: Capturing the discussions, decisions, and rationale from the human-in-the-loop review stages.
- Onboarding Support: Using the structured artifacts (especially the IR, explanations, visualizations, and traceability matrix) to help new team members quickly understand the system's genesis, architecture, and the link between intent and implementation, mitigating the risk of knowledge loss.
- Knowledge Base Maintenance: Regularly updating the semantic mappings and constraint models based on feedback, new technologies, and evolving best practices.

Effective knowledge management ensures that the understanding generated and captured by the IDEA framework remains accessible and useful throughout the system's lifespan, directly combating the understandability challenges that often plague complex or unconventionally generated software.

7. Discussion

The proposed IDEA framework offers a structured approach to harness the potential of vibes coding while upholding software engineering principles. However, its implementation and adoption entail various benefits, drawbacks, limitations, and ethical considerations that warrant discussion.

7.1. Potential Benefits and Drawbacks of the IDEA Framework

Benefits:

- **Alignment with High-Level Goals:** By starting from the vibe, the framework potentially enables development that is more closely aligned with nuanced user experience goals or subjective requirements often lost in translation during traditional requirements engineering.
- **Accelerated Prototyping (Potentially):** If the generation and explication engines are efficient, the framework could speed up the creation of functional prototypes that embody specific experiential qualities.
- **Enhanced Traceability and Rationale:** A key strength is the explicit focus on capturing and documenting the link between intent (vibe, IR) and implementation (code), including the rationale, which directly improves understandability.
- **Encapsulation of Best Practices:** Embedding architectural constraints and design patterns into the generation process can promote adherence to good engineering practices.
- **Forced Articulation of Rationale:** The framework compels the system (and potentially the developers guiding it) to make design rationale explicit, which is often implicit or lost in traditional development.

Drawbacks:

- **Framework Complexity:** The IDEA framework itself introduces significant process overhead and requires sophisticated tooling (IR processors, constrained AI generators, explication engines, visualization tools).
- **Reliance on Advanced Technology:** Its feasibility depends heavily on the maturity and capability of underlying technologies, particularly in controllable AI code generation and robust natural language understanding for vibe interpretation and explanation.
- **Risk of Misinterpretation:** Despite formalization efforts (Stage 1), accurately capturing and translating a subjective "vibe" remains challenging. Misinterpretations can lead to generated code that functionally works but fails to meet the underlying intent.

- **Difficulty in Validating Subjective Qualities:** Objectively verifying whether the final software truly embodies the intended "vibe" is inherently difficult and requires novel testing and evaluation approaches (as discussed in 6.1).
- **Potential Creativity vs. Structure Tension:** The emphasis on constraints and structure, necessary for maintainability, might be perceived as conflicting with the creative freedom implied by the term "vibes coding." Balancing these requires careful calibration of the framework's constraints.
- **Effort Redistribution:** While potentially automating some coding tasks, the framework demands significant effort in vibe elicitation, IR definition, semantic mapping, review, and validation, meaning overall development effort might be redistributed rather than reduced.

7.2. Limitations and Areas for Future Research

The IDEA framework, as proposed, has several limitations that point towards necessary future research:

- **Scalability:** Applying the framework effectively to large, complex enterprise systems remains an open question. Managing intricate IRs, complex constraint interactions, and large-scale code generation and explanation presents significant scaling challenges.
- **Handling Conflicting Vibes:** The framework currently assumes a relatively coherent vibe. How to handle requirements that contain inherently conflicting vibes (e.g., "fast and playful" vs. "deliberate and secure") needs further investigation, potentially requiring negotiation mechanisms during Vibe Formalization.
- **IR Robustness and Generality:** Developing Intermediate Representations that are expressive enough to capture diverse vibes across different domains, yet formal enough for reliable processing, is a major research challenge.
- **Controllable and Explainable Generation:** Improving the controllability of AI code generators to reliably adhere to complex constraints and enhancing the quality, accuracy, and human-friendliness of automatically generated explanations are critical areas for advancement [24-26].
- **Empirical Validation:** The effectiveness of the IDEA framework in demonstrably improving maintainability metrics (e.g., understandability, modifiability, analyzability) compared to traditional methods or unconstrained vibes coding needs rigorous empirical validation through case studies and controlled experiments. Measuring the impact on long-term maintenance costs is also essential.
- **Tooling and Integration:** Building integrated development environments (IDEs) or toolchains that seamlessly support all stages of the IDEA framework is necessary for practical adoption.
- **Democratization:** Extending coding to non-programmers may be in jeopardy as we would require these non-programmer to develop a whole bunch of additional skills and follow strict processes, achievable by programmers, but challenging for others.

7.3. Ethical Considerations in Intuition-Driven Development

The ability to translate vibes directly into software introduces potential ethical concerns:

- **Manipulation:** Vibes related to persuasion, addiction, or urgency could be used to generate user experiences that are manipulative or exploit psychological vulnerabilities. The framework currently lacks mechanisms to detect or prevent malicious vibes.
- **Accountability:** If vibe-coded software causes harm (e.g., due to security flaws, biased behavior, or unexpected consequences), establishing accountability is complex. Is it the developer who provided the vibe, the AI that generated the code, the designers of the IDEA framework, or the organization deploying the software?
- **Bias Amplification:** AI models used for vibe interpretation or code generation can inherit and amplify biases present in their training data. A vibe intended as neutral could be translated into biased code or behavior if the underlying models encode societal biases related to gender, race, or other characteristics.
- **Deskilling and Job Roles:** While potentially empowering some, over-reliance on automated generation could devalue traditional programming skills and impact developer job roles, necessitating workforce adaptation.

Addressing these ethical dimensions requires ongoing discussion, the development of guidelines for responsible use, and potentially building safeguards or ethical checks into the framework itself.

8. Conclusions

This paper confronted the significant challenge of ensuring maintainability and supportability for software developed via vibe coding, the translation of intuitive intent into code. The inherent ambiguity of vibes and the potential opacity of generative processes pose direct threats to code understandability and long-term maintainability. To mitigate these risks, we proposed the Intent-Driven Explicable Architecture (IDEA) framework. IDEA provides a structured, multi-stage approach encompassing:

- (1) Vibe Formalization to capture and structure intent using Intermediate Representations;
- (2) Constrained Generation to synthesize code adhering to architectural principles and maintainability heuristics;
- (3) an Explication Engine to automatically generate traceability links, justifications, and documentation linking code back to intent; and
- (4) rigorous Human-in-the-Loop Validation for review, feedback, and refinement. By systematically addressing ambiguity, enforcing engineering discipline, ensuring traceability, and demanding human oversight, the IDEA framework aims to render vibe-coded software comprehensible, modifiable, and thus maintainable throughout its lifecycle.

The considerations also apply to other AI coding practices, beyond vibe coding as defined here and in [1].

Vibes coding, represents a fascinating direction in human-computer interaction and software creation. It reflects a desire to bridge the gap between human feeling and digital artifact more directly. However, as this paper argues, realizing this potential responsibly requires acknowledging and addressing the fundamental software engineering challenges involved, particularly maintainability.

Frameworks like IDEA, which emphasize structure, explanation, and human oversight, offer a potential pathway for exploring such intuitive programming paradigms without succumbing to the pitfalls of opaque, unmanageable

systems. The future likely involves not a replacement of traditional methods, but rather the development of sophisticated tools and methodologies that enable a tighter, more explicit, and more understandable collaboration between human intuition, design principles, and AI-driven automation. Further research and empirical validation are essential to refine these approaches and understand their practical implications, ensuring that the pursuit of intuitive programming leads to software systems that are not only innovative but also robust, sustainable, and trustworthy.

However, with VIBE4M {a1} or IDEA, as discussed here, democratization of coding may still encounter headwinds.

References

- [1]: Stephane H. Maes, (2025), "The Gotchas of AI Coding and Vibe Coding. It's All About Support And Maintenance", <https://doi.org/10.5281/zenodo.15343349>, <https://shmaes.wordpress.com/2025/04/28/the-gotchas-of-ai-coding-and-vibe-coding-its-all-about-support-and-maintenance/>, April 28, 2025.
- [2]: Stephane H. Maes, (2024), "Fixing Reference Hallucinations of LLMs", <https://doi.org/10.5281/zenodo.14543939>, <https://shmaes.wordpress.com/2024/11/29/fixing-reference-hallucinations-of-llms/>, November 29, 2024. (osf.io/u38w4/, [viXra:2412.0149v1](https://arxiv.org/abs/2412.0149v1)).
- [3]: Stephane H. Maes, (2024), "The Trouble with GenAI: LLMs are still not any close to AGI. They will never be", <https://zenodo.org/doi/10.5281/zenodo.14567206>, <https://shmaes.wordpress.com/2024/12/26/the-trouble-with-genai-llms-are-still-not-any-close-to-agi-they-will-never-be/>, December 25, 2024. (osf.io/qdaxm/, [viXra:2501.0015v1](https://arxiv.org/abs/2501.0015v1)).
- [4]: Stephane Herman Maes, Chalapathy Venkata Neti, (2005), "System and method for multi-modal focus detection, referential ambiguity resolution and mood classification using multi-modal input", US Patent 6964023, November 8, 2005.
- [5]: Stephane H Maes, TV Raman, (2000), "Multi-modal interaction in the age of information appliances", 2000 IEEE International Conference on Multimedia and Expo. ICME2000. Proceedings. Latest Advances in the Fast Changing World of Multimedia (Cat. No. 00TH8532), July 30, 2000.
- [6]: Stephane H Maes, TV Raman, (2000), "Multi-modal Web IBM Position", W3C/WAP Workshop, IBM Human Language Technologies, September, 2000.
- [7]: Stephane H Maes, (2001), "Multi-modal Architecture Recommendation and Multi-device Browsing", Proceedings of the 6th National Conference on Human-Computer Speech Communication (NCMMSC6), 2001, 2001.
- [8]: Stephane H. Maes, Karan Singh Chhina, Guillaume Dubuc, (2021), "Natural language translation-based orchestration workflow generation", US Patent 11120217.
- [9]: Julian Horsey, (2025), "Codebase Knowledge Builder AI Tool Turns Confusing Code Into Easy to Follow Tutorials", Geeky Gadgets, <https://www.geeky-gadgets.com/ai-agent-core-components-diagram-creator/>, May 2, 2025.
- [10]: Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. d. O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., Ryder, N., Pavlov, M., Power, A., Kaiser, L., Bavarian, M., Winter, C., T. M., H., Brooks, T., Chevalier-Boisvert, M., Ondoro, C., McGrew, B., Sutskever, I., Amodei, D., & Zaremba, W., (2021), "Evaluating Large Language Models Trained on Code", arXiv:2107.03374v2.

- [11]: Emma Woollacott, (2025), "Want to supercharge your vibe coding skills? Here are the best AI models developers can use to generate secure code. Claude 3.7 Sonnet is the best performer for vibe coding, while others produce very mixed results", IT.Pro, <https://www.itpro.com/software/development/vibe-coding-best-ai-models-secure-code-generation>, April 25, 2025.
- [12]: Jia Li, Ge Li, Yongmin Li, Zhi Jin, (2023), "Structured Chain-of-Thought Prompting for Code Generation", arXiv:2305.06599v3.
- [13]: Lientz, B. P., Swanson, E. B., (1980), "Software Maintenance Management", Addison-Wesley.
- [14]: von Mayrhauser, A., Vans, A. M., (1995), "Program comprehension during software maintenance and evolution", Computer, 28(8), 44-55.
- [15]: Bansal, G., Nushi, B., Kamar, E., Lasecki, W. S., Weld, D. S., & Horvitz, E., (2019), "Updates in Human-AI Teams: Understanding and Addressing the Performance/Compatibility Tradeoff", Proceedings of the AAAI Conference on Artificial Intelligence, 33, 2429-2437.
- [16]: ISO, (2024), "ISO/IEC 25002:2024, Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — Quality model overview and usage", <https://www.iso.org/standard/78175.html>.
- [17]: Lehman, M. M., & Belady, L. A., (1985), "Program Evolution: Processes of Software Change", Academic Press.
- [18]: Gamma, E., Helm, R., Johnson, R., & Vlissides, J., (1995), "Design Patterns: Elements of Reusable Object-Oriented Software", Addison-Wesley.
- [19]: Chidamber, S. R., & Kemerer, C. F., (1994), "A Metrics Suite for Object Oriented Design", IEEE Transactions on Software Engineering, 20(6), 476-493.
- [20]: Ko, A. J., Myers, B. A., Coblenz, M. J., Aung, H. H., (2006), "An exploratory study of how developers seek, relate, and use information during software maintenance tasks", IEEE Transactions on Software Engineering, 32(12), 971-987.
- [a21]: Nijkamp, E., Pang, B., Hayashi, H., Tu, L., Wang, H., Zhou, Y., Savarese, S., Xiong, C. (2023), "CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis", arXiv:2203.13474v5.
- [22]: Rajeev Alur, et al., (2013), "Syntax-Guided Synthesis", 2013 Formal Methods in Computer-Aided Design.
- [23]: Ziegler, D., Stiennon, N., Wu, J., Brown, T. B., Radford, A., Amodei, D., Christiano, P., & Irving, G., (2022), "Fine-Tuning Language Models from Human Preferences", arXiv:1909.08593v2.
- [24]: Wang, D., Yang, Q., Abdul, A., & Lim, B. Y., (2019), "Designing Theory-Driven User-Centric Explainable AI", Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems (CHI '19), Paper 613, 1–15.
- [a5]: Bussone, A., Stumpf, S., & O'Sullivan, D., (2015), "The Role of Explanations on Trust and Reliance in Clinical Decision Support Systems", Proceedings of the 20th International Conference on Intelligent User Interfaces (IUI'15), 160–169.
- [26]: Castelvetti, D., (2016), "Can we open the black box of AI?", Nature, 538(7623), 20-23.