BonsAIDE: An Extended Vision for Human-AI Interaction in IDEs

DAVID MORENO-LUMBRERAS, Universidad Rey Juan Carlos, Spain
RAULA GAIKOVINA KULA, The University of Osaka, Japan
CHRISTOPH TREUDE, Singapore Management University, Singapore

AI-driven coding assistants are transforming software development, yet their full potential in Integrated Development Environments (IDEs) remains underutilized. A key challenge is their tendency to hallucinate, producing plausible but incorrect code and leading developers down unintended paths. Current static file-based IDEs also lack support for tracking the provenance of AI-generated code or integrating version control in ways that match the dynamic and iterative nature of AI-assisted workflows. Consequently, developers lack tools to systematically manage, refine, and validate Generative AI (GenAI) code, making correctness, maintainability, and trust difficult to ensure. Inspired by the art of Japanese Bonsai gardening—emphasizing balance, structure, and pruning—we propose a new paradigm: an IDE where AI is free to generate, and developers guide evolution by pruning and shaping alternatives. We present BonsAIDE, a prototype that supports branching, comparison, and pruning of AI-generated code. In an initial study with ten participants, we observed: (1) diverse exploration strategies across identical tasks; (2) high tool acceptance with low perceived difficulty; (3) benefits of branching and pruning, including clutter reduction and parallel exploration; and (4) concrete feedback on desired features such as side-by-side diffs and improved navigation. These findings motivate future research on provenance, prompt-aware navigation, and scalable human–AI interaction.

ACM Reference Format:
David Moreno-Lumbreras, Raula Gaikovina Kula, and Christoph Treude. 2026. BonsAIDE: An Extended Vision for Human-AI Interaction in IDEs. 1, 1 (June 2026), 25 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

> *"The art of bonsai is a journey of discovery, a path that leads to a profound understanding of the delicate balance between nature and human creativity.[1]"*

Modern IDEs were designed for human-written code—where edits are controlled, revisions are structured, and workflows are predictable. AI-generated code, by contrast, emerges unpredictably, lacks provenance, and often requires iterative refinement. This creates challenges for IDEs that lack tools to track code origins, verify accuracy, or support seamless integration. Developers face risks such as AI hallucinations—plausible but incorrect code—and must spend significant time manually verifying and reconciling changes, increasing the chance of subtle errors. As generative AI becomes more central to development, the focus is shifting from the code itself to the intent behind its generation—a change that demands new tools and workflows to ensure reliability and accountability.

[1] https://bonsaibotanica.com/the-science-behind-bonsai-natures-living-art/

Authors' addresses: David Moreno-Lumbreras, Universidad Rey Juan Carlos, Spain, david.morenolu@urjc.es; Raula Gaikovina Kula, The University of Osaka, Japan, raula-k@ist.osaka-u.ac.jp; Christoph Treude, Singapore Management University, Singapore, ctreude@smu.edu.sg.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2026 Association for Computing Machinery.
Manuscript submitted to ACM

Existing IDEs treat genAI code as static text, offering little support for its evolution. Tools like GitHub Copilot[2], OpenAI's ChatGPT[3], and Google's Gemini[4] position genAI as an assistant or agent that operates side-by-side with developers. However, such models are vulnerable to hallucinations [21], especially in reasoning chains or chain-of-thought processes [35], which remain opaque to users. A 2025 report indicates that closed genAI models often do not disclose their technical underpinnings[5], resulting in limited traceability and transparency. This lack of visibility undermines developer agency and trust—making it difficult to verify whether generated code aligns with intended design or logic. Without fine-grained control, developers must accept genAI suggestions without understanding the reasoning behind them, increasing the risk of integrating erroneous or suboptimal code. Since current IDEs were designed for human-written code, retrofitting genAI into them fails to address these core challenges. Instead, we propose a new paradigm—a Bonsai-inspired IDE—that treats genAI code as an evolving structure, enabling developers to guide, explore, and refine it dynamically. In this model, genAI code is not merely inserted but actively managed throughout the development lifecycle.

In this vision paper, we introduce a novel concept: reimagining how generative AI (genAI) can be fully harnessed while preserving developer agency. Rather than treating code as static, file-based artifacts, we propose viewing it as a dynamic collection of genAI code snippets—each linked to the prompt from which it was generated. Inspired by Code Bubbles [6], which shifts from file-based to fragment-based code structures. We envision a framework where genAI-generated code evolves through intentional, guided modifications—much like a Bonsai tree, carefully pruned and shaped to achieve balance and purpose. This model emphasizes flexibility and adaptability, enabling developers to refine, branch, and merge code alternatives in real time—always maintaining full control over the decision-making process. The IDE becomes a co-evolving platform that dynamically adapts to AI-generated changes, enabling seamless interaction, continuous refinement, and thoughtful decision-making. Ultimately, the goal is to create a system where genAI coding evolves hand-in-hand with developer intent—ensuring transparency, control, and alignment at every step.

The Bonsai-inspired IDE integrates design principles that mirror the aesthetics and philosophy of bonsai cultivation to enhance the developer experience. *Asymmetry* is embodied in the Code Generation Graph and Prompt-Driven Development View, where genAI-generated code evolves in dynamic, non-linear ways—mirroring organic growth. *Simplicity* is achieved through the Intent-Based Project Structure and AI-Assisted Code Morphing, which streamline interactions and reduce cognitive load. *Proportion* is maintained in the presentation of Dynamic Code States and Hierarchical Generation Layers, offering a balanced view of code across development stages. Finally, *Depth* is realized through the Interactive Code Evolution Timeline, Regeneration Networks, and AI Code Sandboxing—enabling developers to trace, assess, and refine the evolution of their code over time. We realize this vision in *BonsAIDE*, a VSCode-based prototype that supports branching, comparison, and pruning of genAI-generated alternatives. To validate its effectiveness, we conduct a user study and release the prototype alongside a replication package (Sections 4 and 5.1).

Finally, we outline a research agenda focused on key challenges in building a sustainable, developer-agentic ecosystem for genAI-assisted coding. Together, these challenges form a foundation for a new paradigm in AI-assisted development, where code evolves not as a static artifact, but as a living, traceable, and intentionally shaped system under the guidance of the developer.

[2] https://github.com/features/copilot
[3] https://openai.com/chatgpt/overview/
[4] https://gemini.google.com/
[5] https://responsibleinnovation.org/wp-content/uploads/2025/01/ARI-Report-Transparency-in-Frontier-AI.pdf

2 BACKGROUND AND RELATED WORK

In this section, we present background and related literature on the IDE, current usage of GenAI technologies like LLMs and agent interactions with humans, and finally initial ideas on how an AI-driven IDE works.

2.1 The Non-AI driven IDE (Eclipse and VSCode)

Traditional Integrated Development Environments (IDEs) such as Eclipse[6] and Visual Studio Code (VSCode)[7] have revolutionized the way developers approach coding. They provide developers with powerful tools that facilitate modular coding practices, from writing code at various levels (classes, methods, variables) to refactoring, building, and testing. Their user-friendly interfaces further enhance developer productivity by offering customization and intuitive navigation, making them indispensable tools in the modern software development landscape. Eclipse, known for its extensive plugin support, offers a modular approach to development, allowing developers to integrate various tools such as debuggers, version control systems, and automated testing frameworks. VSCode, built on top of the popular Node.js platform, also caters to this modular coding paradigm by enabling the use of extensions that enhance functionality.

Developers leverage these IDEs for their ability to handle complex projects through features like code editing with syntax highlighting and auto-completion. They can refactor code by restructuring modules or classes, ensuring a clean and maintainable codebase. The build process is streamlined, often automated through tools integrated into the IDE, that compile and link different parts of the code together seamlessly. Additionally, testing frameworks within these environments allow developers to run test cases efficiently, ensuring robust functionality. The user interface (UI) of traditional IDEs like Eclipse and VSCode is designed with developer productivity in mind. These interfaces typically consist of a sidebar for file navigation, a top menu bar for accessing commands, and an editor area where code is written and edited. Customization options allow developers to tailor the UI to their preferences, improving ease of use. VSCode's interface, for instance, is known for its flexibility, with a market of thousands of extensions available to add functionality. Eclipse also offers extensive customization; in some workflows, VSCode may feel more straightforward depending on the setup. Both IDEs prioritize a clean and intuitive layout that simplifies navigation and access to tools. The modular coding approach supported by these IDEs is further enhanced by their UI design, which organizes code into logical sections, making it easier for developers to manage and manipulate different parts of their projects. This structure not only promotes efficient coding, but also aids in debugging and refining the codebase as needed[8].

2.2 Human-AI Interaction in Software Development

The growing integration of AI into software engineering has motivated extensive research on how humans and AI should collaborate effectively in development environments. Foundational work on design principles [3] provides guidelines such as timely feedback, transparency, and error prevention, which remain critical as LLM-based coding assistants are embedded in IDEs. Building on this foundation, Sergeyuk et al. [31] highlight in their review that the in-IDE Human-AI experience is shaped by task-specific interface design, productivity impacts, and the quality of interaction in terms of trust and readability. Along similar lines, Nghiem et al. [30] present insights for next-generation AI assistants, stressing the importance of clear expectations, seamless integration with IDE capabilities, and responsible data collection practices.

[6] https://eclipseide.org/
[7] https://code.visualstudio.com/
[8] https://code.visualstudio.com/docs

Beyond interface design, another important aspect of human–AI interaction is how developers engage with LLM outputs in evaluative roles. Rather than simply accepting suggestions, humans often need to judge, validate, or annotate AI-generated code. Ahmed et al. [1] explore this by examining whether LLMs can replace manual annotation of software artifacts, while Crupi et al. [12] study the role of LLMs as "judges" of code generation and summarization, effectively shifting part of the evaluation workload away from the developer. Zhao et al. [37] extend this direction with CodeJudge-Eval, a benchmark designed to probe how models themselves assess code correctness, highlighting that human oversight remains essential as even state-of-the-art LLMs struggle with subtle errors. These studies reveal that interaction is not limited to generation, but also involves collaborative judgment between humans and AI systems. At the same time, Evtikhiev et al. [13] argue that widely used metrics such as BLEU are insufficient to capture code quality, and Gao et al. [16] emphasize broader challenges—including hallucinations and lack of transparency—that shape how developers must critically interact with AI suggestions.

These challenges are reflected in the adoption of practical tools such as GitHub Copilot, which moves beyond passive auto-completion to offer real-time, conversational assistance inside the IDE. Copilot provides inline suggestions, documentation, and refactoring support, effectively acting as a pair programmer. According to GitHub[9], 85% of developers reported more confidence in their code quality, and 88% noted that Copilot Chat helped them maintain focus and reduce frustration. While not fully autonomous, these assistants illustrate how human–AI interaction in IDEs is shifting towards a more collaborative model.

Finally, interaction is not limited to one-to-one human–AI collaboration. Frameworks such as CAMEL [25] envision communicative multi-agent systems where LLMs reason collectively, opening possibilities for richer collaboration but also raising new questions about developer control and agency. This body of work shows both the potential and the risks of integrating AI into software development. While guidelines, benchmarks, and frameworks provide important foundations, there remains a gap in enabling developers to dynamically manage, compare, and refine AI-generated code. Our vision of the BONSAI follows these approaches where the human has a more informative interaction with the AI.

2.3 AI Agents and their integration into the IDE

The concept of autonomous agents has long been studied in artificial intelligence. Franklin and Graesser [15] provided one of the earliest taxonomies to distinguish agents from traditional programs, and subsequent surveys such as Albrecht and Stone [2] explored how agents can model the behavior of others, highlighting both progress and open challenges. With the advent of LLMs, these ideas have been revitalized, enabling agent-like capabilities such as planning, reasoning, and negotiation. He et al. [18] review LLM-based multi-agent systems for software engineering, showing their potential to improve robustness and scalability in complex tasks, while also identifying gaps in agent collaboration and reliability.

Recent work has proposed concrete frameworks where agents collaborate on development tasks. Cai et al. [7] demonstrate graphical interfaces that lower the barrier to interacting with LLMs, while Chen et al. [10] introduce CodeR, which coordinates agents through task graphs for issue resolution. Similarly, AgentVerse [11] explores emergent behaviors in multi-agent collaboration, and Hu et al. [20] propose self-evolving collaboration networks for software projects. Other perspectives emphasize the role of intelligent agents in static analysis [14] or in reasoning and coordination through frameworks like Flows [22]. Together, these contributions illustrate how agents can move beyond passive assistants to active participants in software engineering workflows.

[9]https://github.blog/news-insights/research/research-quantifying-github-copilots-impact-on-code-quality/

The agent perspective also intersects with how IDEs themselves are envisioned. Gonzalez-Barahona [17] links LLMs with extended reality as part of next-generation development environments, while Marron [27] argues for a shift from integrated to intelligent IDEs that orchestrate AI agents and automation rather than serving merely as editors. Complementary visions include advanced debugging support, as illustrated by the Visual Debugger [24], which rethinks how developers interact with runtime information. Beyond SE, even research challenges like the Hanabi cooperative game [4] have served as testbeds for multi-agent reasoning, offering insights relevant to AI coordination in development contexts. The literature points to a convergence of multi-agent systems and IDE innovation. While agents provide autonomy, reasoning, and collaboration, IDEs offer the context where these capabilities can be harnessed. Our Bonsai-inspired approach builds on this trajectory, proposing an IDE where agents and developers interact in a structured, visual environment that emphasizes exploration, pruning, and refinement of AI-generated code.

3 THE VISION: A BONSAI-INSPIRED IDE

"The art of bonsai is a living sculpture, shaped by the hands of the artist and the forces of nature." – Unknown

The vision of an AI-native IDE is inspired by the cultivation of a Bonsai tree – transforming software development from a static, file-based process into a dynamic, evolving system where AI generation and developer-guided refinement work in tandem.

3.1 Features

3.1.1 Intent-driven Navigation Beyond Files. Like a Bonsai tree that grows under careful and intentional guidance, this AI-native IDE enables developers to navigate their codebase dynamically – moving beyond rigid file structures to an intent-driven system based on semantic queries. Developers can explore their code dynamically, using intent-based prompts and semantic queries to retrieve, modify, and refine generated code. This metaphor of a Bonsai tree emphasizes flexibility and growth, as developers prune and shape their codebase through interactions that are context-driven, rather than confined to the traditional file and directory structure.

3.1.2 Parallel Code path Branching for AI Exploration. Parallel AI exploration and branching in the AI-native IDE resemble the growth of a Bonsai tree, where multiple code paths evolve simultaneously and can be selectively refined. Rather than following a single, linear iteration, developers can branch out and explore alternative implementations in parallel, much like a Bonsai artist deciding how each branch will grow and evolve. The system maintains contextual awareness across branches, allowing developers to selectively merge, refine, or prune AI-generated code paths. This process reflects the Bonsai artist's ability to shape multiple branches simultaneously, maintaining harmony and balance in the overall structure of the tree while adapting it to new growth opportunities.

3.2 Incorporating Bonsai Aesthetic Principles

The art of bonsai styling employs various techniques to mould the tree's growth and shape. Pruning, wiring, and pinching are used to achieve balance, proportion, and harmony. The aim is to replicate the presence of a mature tree in nature, imbuing it with character and presence. The IDE design is guided by four key Bonsai styling principles, drawing on established sources on its aesthetics and methodology [8, 9, 33].

(1) **Asymmetry:** Asymmetry is a core principle in Bonsai styling, where trees are shaped to reflect natural, organic growth rather than rigid symmetry.

(a) *Code Generation Graph:* The IDE can display a graph that does not follow a rigid hierarchical structure but rather mimics the organic growth of a tree, where nodes branch out in a natural, non-symmetrical way. Linkages between generated code snippets and the prompts are dynamically visualized, rather than following strict parent-child hierarchies like traditional file trees. This approach highlights the flexibility and continuously evolving nature of AI-generated code.

(b) *Prompt-Driven Development View:* Similar to asymmetry, prompts and modifications could be arranged in a more organic flow. The timeline or canvas would not necessarily be linear but would evolve dynamically as developers add new inputs or make changes, reflecting a more natural progression that is not bound by strict order.

(2) **Simplicity:** Simplicity is another crucial principle, as the beauty of bonsai lies in its simplicity, with each element carefully chosen to contribute to the overall composition.

(a) *Intent-Based Project Structure:* Rather than overloading users with unnecessary information or complexity, the project structure focuses on developer intent, much like how bonsai pruning strips away unnecessary elements to showcase simplicity. The interface should be intuitive and minimalistic, focusing on the purpose of each AI interaction or constraint while hiding the complexity of the underlying code generation process.

(b) *AI-Assisted Code Morphing:* Instead of a cluttered set of tools or settings, the code morphing process should be simple, focusing solely on the key constraints that affect code readability, efficiency, or style. The interface should allow users to seamlessly adjust these elements, maintaining a clean design that emphasizes ease of use.

(3) **Proportion:** Proportion plays a key role, ensuring that the size of the tree, the pot and any accompanying elements are in harmony, creating a sense of balance and visual appeal.

(a) *Dynamic Code States:* The concept of dynamic code states needs to have proportionate interactions between versions of code. Switching between AI-generated alternatives should feel natural, with developers able to focus on relevant states without feeling overwhelmed by unnecessary versions. Each state – raw AI generation, refinement, and finalized code – should align with the project's development stage, providing sufficient context without overwhelming the user.

(b) *Hierarchical Generation Layers:* The layers of code (base, refinement, and final version) should be represented proportionally in the IDE, with clear delimitations between each stage. Users should be able to easily toggle between these layers to get the right view of their work at any given time, without one layer feeling more dominant than the others.

(4) **Depth:** Finally, depth is incorporated by arranging branches and foliage in layers, giving the illusion of a larger tree and enhancing the overall aesthetic.

(a) *Interactive Code Evolution Timeline:* The timeline should provide depth by allowing users to view a history of their project's evolution. It could show layers of changes over time, allowing users to backtrack and see the progress from the original AI-generated code to the most refined version. This timeline could be visualized as a layered path, offering depth and context, as developers can see how each iteration influenced the code.

(b) *Regeneration Networks:* These networks can add depth by visualizing how different prompts and modifications lead to divergent paths, giving developers a clear view of how one prompt or change can alter the entire project. This view would allow them to explore the depth of possibilities and refine their code with an understanding of the larger picture. From a technical perspective, Regeneration Networks model regeneration as an explicit dependency graph over code states. Nodes correspond to concrete code artifacts produced by an LLM, while edges encode regeneration relationships driven by prompt modifications, task changes, or constraint updates.

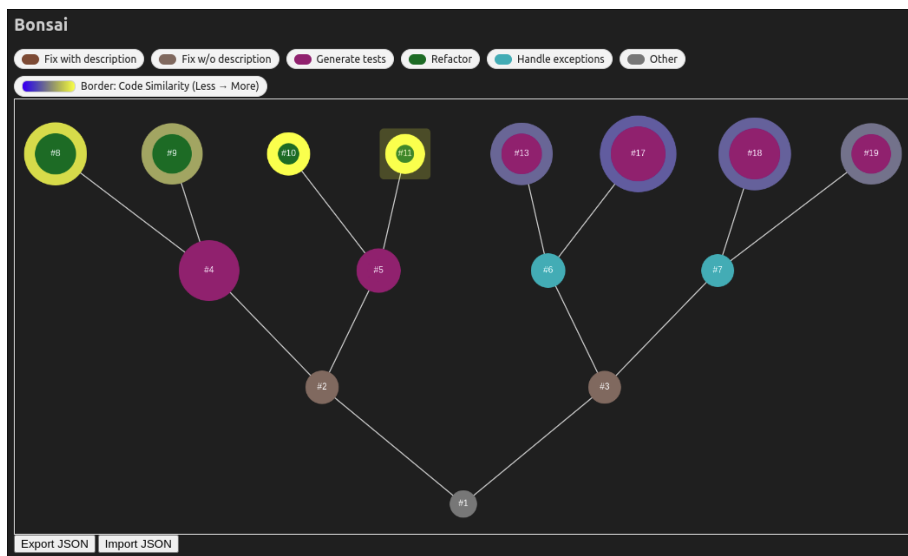

Fig. 1. Example of the BonsIDE with legend of visual encodings: fill = activity; border (on leaf selection) = similarity; size = token budget.

This structure allows BonsAIDE to represent not only successive versions, but alternative regeneration paths that coexist simultaneously. Unlike linear histories, the network captures which code variants share common ancestry and which diverge due to distinct prompts or constraints, enabling targeted comparison and selective pruning of AI-generated alternatives.

4 A VISUAL PROTOTYPE: BONSAIDE

In this section, we introduce a prototype of the bonsai-inspired IDE, the empirical design and results of the user study.

4.1 Brief Overview

As shown in Figure 1, BonsAIDE reframes bug fixing as a graph-based exploration—shaping and pruning an evolving set of alternatives rather than editing a single artifact. Inspired by bonsai aesthetics, the interface visualizes each edit as a node or branch in a tree, capturing alternative solutions and their provenance. As a VSCode plugin [10], the active file initializes as the root node. Subsequent actions generate new nodes representing divergent solution paths. The interface uses minimal visual encodings: node fill denotes the last activity; leaf borders convey similarity via a cool-to-warm gradient; node size reflects model token budget. A details panel exposes generated code (patches, tests, refactorings, exceptions), model reasoning (when available), and metrics such as tokens/LOC and similarity to the anchor. Sessions are serialized as JSON, enabling full auditability and reproducibility.

4.2 Operations on the Bonsai

We implement three core software engineering activities on the bonsai: Fix the problem (mandatory first step), Generate tests, Refactor, and Handle exceptions. These activities are selected based on their applicability to code snippets at this

[10]https://marketplace.visualstudio.com/items?itemName=dlumbrer.bonsai-code

Fig. 2. Details for the selected node: Code, Reasoning, Similarity, and Code Metrics.

level of abstraction and are grounded in common developer practices. The activity set in BonsAIDE is deliberately small and representative of recurring tasks reported by Meyer et al. [28]. It remains orthogonal and composable within BonsAIDE's branch-oriented interaction model. Each activity accepts a numeric input N, spawning N alternative solutions—making divergence explicit, structured, and comparable.

A typical session begins in the editor. The user launches BonsAIDE, and the active file appears as the first node on the canvas. The workflow follows a *fix-first* pattern: the user selects *Fix the problem* and, before committing, specifies a numeric input N to generate N alternative branches. These parallel branches preserve competing ideas, allowing the user to navigate, compare, and evaluate outcomes without overwriting code.

The canvas provides inline guidance:

- Node fill indicates the last activity (e.g., fix, test, refactor, exception handling), showing at a glance what each node represents.
- When a leaf is selected, border colors use a cool-to-warm gradient to highlight similar candidates, signaling those close to the current anchor and worth inspecting.
- Node size approximates the token budget of the underlying model interaction, offering a visual cue for cost and scope.

These encodings—shown in the legend—minimize the need for external annotations and support lightweight, intuitive navigation (Fig. 1). Selecting a node reveals the generated artifact (Fig. 2):

Fig. 3. Trim: (top) before pruning — dense canvas; (bottom) after pruning — focused working set.

- A patch for **Fix**, test files for **Generate tests**, structural edits for **Refactor**, and guards/handlers for **Handle exceptions**.
- When available, reasoning explains key design choices—such as off-by-one bounds or pivot decisions.
- Metrics include token/LOC usage and, for leaf anchors, a similarity score that aligns with the visual border cues.

Each activity generates a prompt sent to the backend model. BonsAIDE automatically constructs this prompt from the current context: task intent and constraints, the selected code state, any optional user-provided description, and instructions to return a minimal, well-scoped change with a brief rationale. The prompt evolves across steps to preserve relevant context and tighten the request—following patterns observed in real-world prompt engineering [32]. This ensures consistent, auditable prompts without requiring manual prompt engineering, and all generated artifacts are attached to their originating node for later review.

As exploration progresses, users apply Trim to maintain focus (Fig. 3). Pruning removes a node and its descendants from the canvas without modifying source files, reducing the working set to only relevant elements while preserving the full decision graph for audit. Before pruning, multiple branches compete for attention; after pruning, the canvas reveals a smaller, more coherent neighborhood centered on the most promising path.

5 EMPIRICAL METHODOLOGY

To evaluate our visual IDE design, we develop a controlled experiment, with a sample set of users. We evaluate BonsAIDE on short, self-contained bug-fixing tasks derived from the QuixBugs benchmark [26]. The results, procedure, questionnaires, and analysis are available in the replication package [11]

5.0.1 Participant Demographics. We recruited 10 participants from academia and industry, all of whom volunteered to participate. A short demographics form was used to capture years of programming, Python, and IDE experience; frequency of AI assistant use; and self-reported color vision—no personal identifiers were collected. Primary outcomes include task completion (measured by the final exported node) and time on task. From the exported decision graphs, we compute descriptive statistics such as node and edge counts, maximum depth, branching frequency per activity, and activity distribution. Per-node metrics—including tokens per LOC and similarity scores for leaf nodes—are summarized to characterize the size and proximity of candidate solutions. After-Task responses identify the factors influencing the final decision (e.g., tests, refactorings, similarity, token count, time), while the Feedback form provides a brief technology acceptance assessment. All analyses are descriptive and comparative across tasks, with open-ended responses qualitatively coded to uncover patterns in decision-making.

5.0.2 Participant Protocol. Each participant completed at least one, and up to two, short bug-fixing tasks in Python, drawn from the QuixBugs dataset [26]. Sessions were conducted in VS Code with BonsAIDE, following a consistent protocol: participants began with a fix-first approach, optionally branching to explore alternative solutions, using similarity borders to guide comparisons between code states, and leveraging a details panel to anchor evidence—including code, reasoning, and lightweight metrics. The session included Trim to manage clutter and a final export step to preserve the full decision graph.

Participants solved tasks on their own machines with access to BonsAIDE and an LLM backend via API key; their operating system and editor theme were not controlled. Before the task, participants received an information sheet and consented to participation, followed by a brief tutorial covering fix-first, branching, similarity borders, the details panel, Trim, and export. During the session, the screen was recorded (without audio), and participants thought aloud. The task began with fixing the problem, after which they could apply Generate tests, Refactor, or Handle exceptions in any order and frequency, including repeated applications and branching with a numeric input N to generate alternative candidates. The session concluded with an export to JSON, and participants completed an After-Task and a Feedback questionnaire.

BonsAIDE exports a structured decision graph containing nodes (code states), edges (derivations), activity labels, artifacts (patches, tests, refactorings, exception handling), optional reasoning snippets, and metrics such as tokens per LOC and similarity for leaf anchors. Time on task was measured via screen recording or start/end timers. No personal identifiers were collected; all data were stored and analyzed in anonymized form. Each software engineering activity [28] issued a context-aware prompt to the LLM—generated automatically from the current code state, user hypothesis, task intent, and constraints—designed to request minimal, well-scoped changes with a brief rationale. These prompts evolved across steps to preserve context and tighten scope, following prompt evolution patterns observed in software repositories [32], ensuring consistent, auditable interactions without requiring manual prompt engineering. Sessions without a usable export or screen recording were excluded; deviations from the fix-first protocol were recorded, and any persistent crashes or blockers were logged, with sessions terminated and rescheduled if needed.

[11] https://doi.org/10.5281/zenodo.17237356

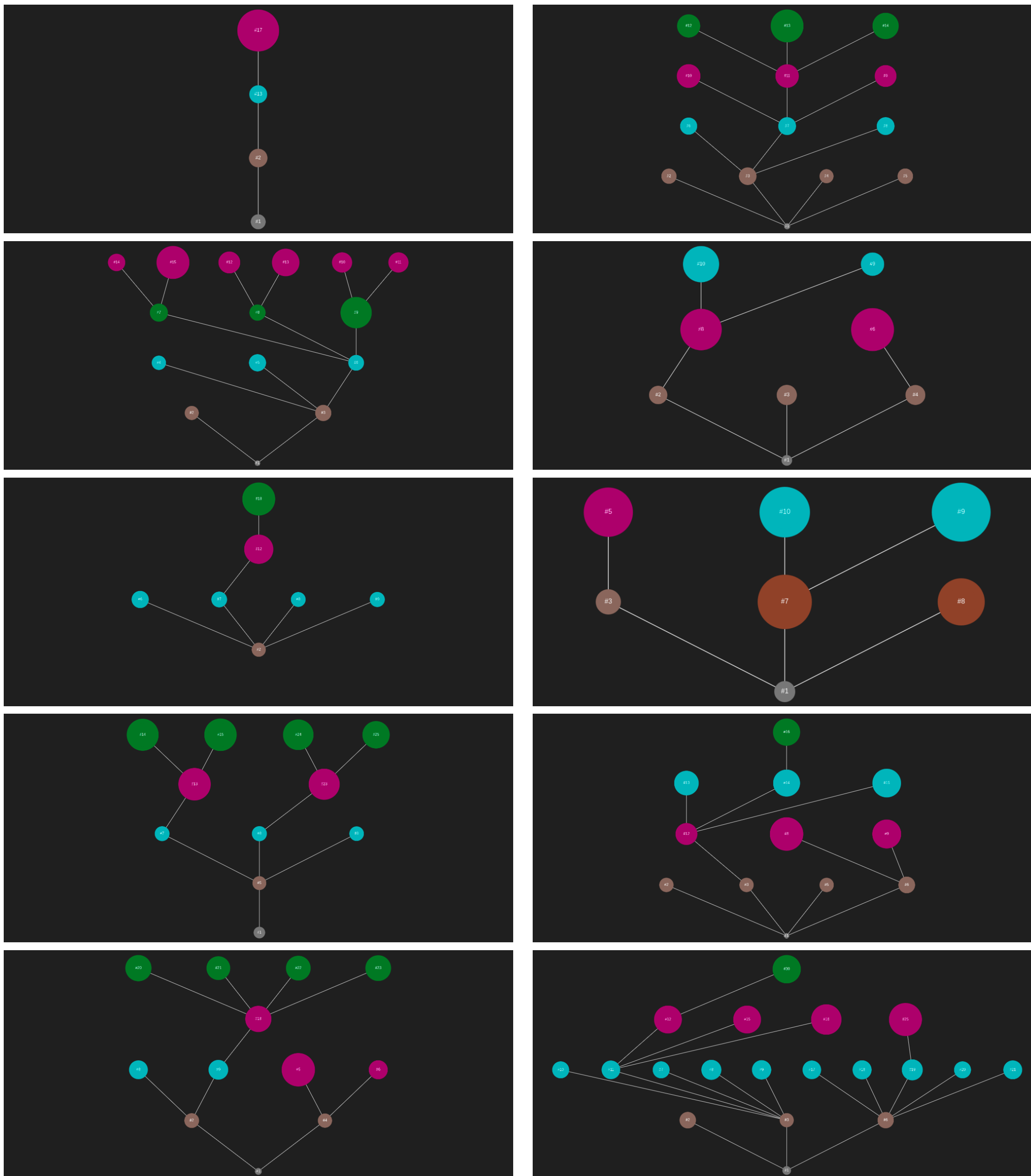

Fig. 4. BonsAIDE trees: the ten exported bonsai graphs.

5.0.3 *Data Collection Methodology.* We administered three forms: a pre-task Demographics, an After-Task form (per problem), and a post-session Feedback & Control survey. The Demographics form captures years of experience in programming, Python, and IDE use; frequency of AI assistant use; and self-reported color vision—no personal identifiers

were collected. The After-Task form records the final chosen node, the rationale for selection, the sequence and frequency of activities used (e.g., fix, test, refactor), and which factors influenced the decision—specifically tests, refactorings, similarity borders, token size, or time—along with whether Trim was applied and includes a reminder to export the decision graph. The Feedback form includes a compact technology acceptance assessment based on the UTAUT2 framework [34], evaluating core constructs such as performance expectancy (e.g., faster task completion, productivity), effort expectancy (learnability, clarity), social influence, facilitating conditions (availability of resources/knowledge), hedonic motivation (enjoyment), and behavioral intention to use AI tools. In addition, we included BonsAIDE-specific items on the perceived value of similarity borders, selection highlights, activity colors, and Trim, along with open-ended prompts exploring users' perceived benefits of viewing alternatives, pruning decisions, and visual feedback.

5.1 Results

In this section, we present the results after all participants successfully resolved the assigned bug using BonsAIDE, focusing on how they explored alternatives and interacted with the tool.

> *Observation 1 - Fixing problems with Bonsai IDE results in a multiple and diverse set of results and strategies*

5.1.1 Diverse BonsAIDE trees. Fig. 4 shows the ten exported bonsai graphs generated by the 10 participants. We can see that each participant produced a distinct exploration shape, even for the same task. Most trees contain multiple levels from the root (iterative steps), while one tree is almost linear with a single main branch. Trim was used in different ways: some participants pruned early and repeatedly to keep a compact working set; others let branches accumulate and used Trim later as a clean-up step. We also see selective pruning of branches with auxiliary actions (for instance, nodes with tests only), and cases where entire subtrees created in a single burst were removed. Every tree necessarily begins with *Fix the problem* (Fix-first protocol). After that first step, several patterns appear: short fan-outs after requesting N=2 or 3 alternatives; depth-first progress with a single Fix path and occasional additional Fix/Refactor steps; side branches created with *Generate tests*; and cases where *Handle exceptions* was applied early. In our sample, some participants did not use *Refactor*.

> *Observation 2 - The technology acceptance model indicates a positive response with high ratings in usage and low rankings in task perception*

5.1.2 Technology and Bonsai acceptance. Fig. 5 depicts the *technology acceptance* results of the survey. Responses are overall positive: learnability and clarity score highest, enjoyment and available resources are also high, productivity and intention to use are clearly positive, and social influence is the lowest item within this block. For the *Bonsai-specific acceptance* block (Fig. 6), the results show low perceived task difficulty (a good outcome for the reversed item), very high visibility and color distinguishability, positive evaluations for similarity and Trim in narrowing candidates, and mid-scale ratings for the assistant's reasoning.

> *Observation 3 - We have three evaluations of the benefits, 1) multiple outcomes enables keeping an original version while trying parallel Code path branching, 2) no visible clutter during exploration, 3) Addition support is needed*

5.1.3 Perceived benefits. We summarize the perceived benefits into three perspectives. First, participants were asked about the benefits of seeing different solutions and branches produced via branching. Responses consistently describe branching as useful to compare alternatives side by side, to keep the original version while trying options, and to

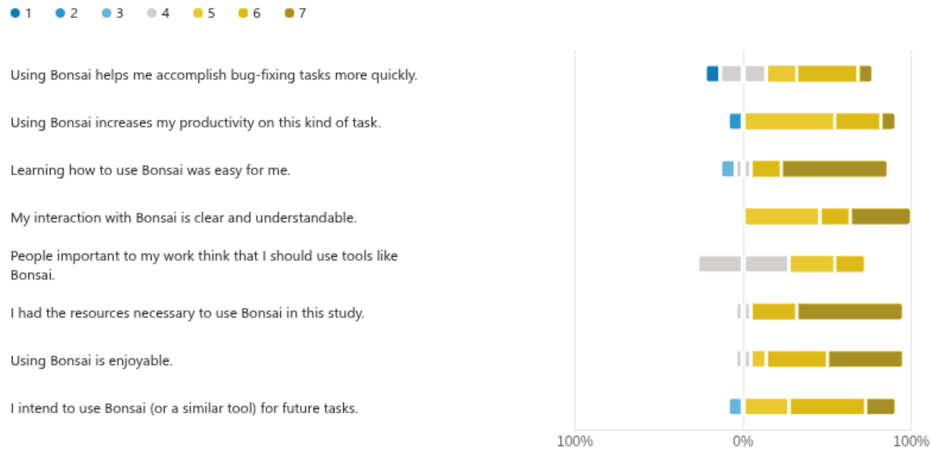

Fig. 5. Technology acceptance Linkert scale results (1 = Strongly disagree ... 7 = Strongly agree).

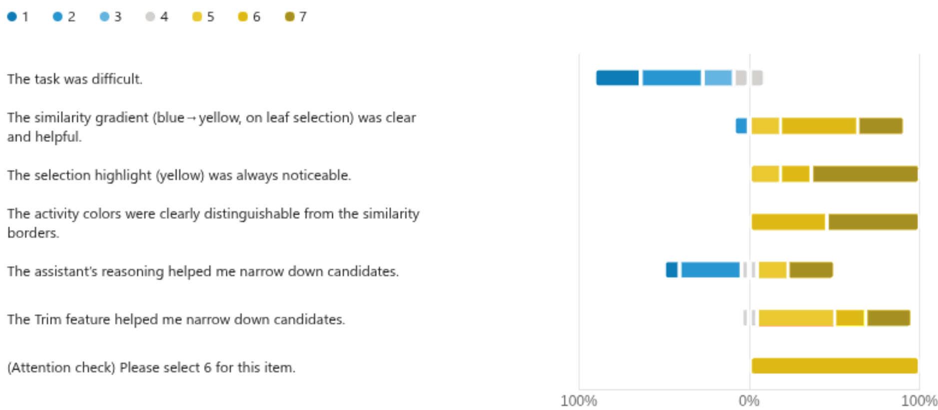

Fig. 6. Bonsai acceptance Linkert scale results (1 = Strongly disagree ... 7 = Strongly agree)

recover quickly if a direction did not work. Several participants noted that having multiple candidates visible helped them notice when distinct fixes converged toward a similar outcome, which in turn supported their final selection. Some also reported adding short notes to branches to keep track of the hypothesis behind each variant.

Second, participants were then asked about the Trim feature (right-click to prune subtrees). Reported benefits focus on reducing visual clutter during exploration, maintaining attention on a smaller working set, and removing branches that were no longer needed while preserving the rest of the graph for reference. A common use case was trimming branches that contained auxiliary actions (for example, nodes with tests only) after they had served their purpose; others described removing entire subtrees created during a burst of alternatives. A few participants indicated they did not use Trim in their run.

Third, participants were asked about the visualization elements (similarity borders on leaf selection, activity colors, node size and simple metrics). Many described borders and colors as quick cues to navigate the tree and understand the provenance of each node. Node size and basic metrics were mentioned as helpful to gauge the relative scope of candidates at a glance. Several responses also pointed to areas where additional support would be welcome, such as stronger side-by-side comparison or simpler navigation on large trees.

Observation 4 - Participants provided useful feedback on issues, and potential new features

5.1.4 Qualitative Feedback for Improvement. Participants provide a range of useful suggestions and issues encountered. Suggestions clustered around a small set of recurrent needs. First, a side-by-side diff between two leaves to make differences explicit at a glance. Second, a quick way to run or sanity-check generated tests on the active node to tighten the loop between suggestion and validation. Third, smoother graph navigation on large trees, with requests for zoom-to-fit, a mini-map, and easier panning. Additional proposals included trimming ergonomics (multi-select Trim, clearer affordances, and an easy undo), brief summaries of what each step changed or what was sent to the model (prompt/change notes), keyboard shortcuts for frequent actions, and clearer labeling or naming of nodes. Feedback converged on a small set of concrete enhancements. First, side-by-side diff between two leaves and small overlays for salient deltas would turn border-driven salience into instant evidence: *"Diff highlighting between candidates would make the decision obvious" (P9).* Second, a quick way to run or sanity-check generated tests on the active node would shorten the loop between hypothesis and confirmation: *"Being able to run the generated tests on the modified code" (P1).* Third, a compact prompt summary or change note per step would clarify how suggestions evolved across the session. Finally, participants asked for minor interaction tweaks (progress cues during node generation, clearer affordances on Trim, and zoom ergonomics) to keep the experience smooth under load.

Reported issues concentrated on performance and interaction details rather than failures. Several responses mentioned intermittent delays while generating nodes, zoom sensitivity in dense graphs, and moments where the selection highlight was harder to relocate after scrolling. There were isolated notes about minor visual glitches (edge overlaps, palette contrast for some color-vision profiles), discoverability of export/import (location of the control and saved files), and occasional UI freezes on very large trees. In the open comments, many respondents described the workflow as easy to pick up and noted that branching plus pruning helped maintain context while exploring alternatives. A recurring request was more status feedback during generation (progress indicators) and a convenient default for the number of variants (N). Multiple respondents indicated they would use the tool again for short bug-fixing tasks of similar scope.

5.1.5 Prompt-techniques comparison: similarity to human-guided BonsAIDE solutions. To complement the qualitative observations from the study with an objective reference point, we introduce a prompt-techniques comparison that contrasts *final participant-selected BonsAIDE patches* against *code produced via static prompts*. The purpose of this comparison is not to evaluate correctness—all candidates considered here are correct bug fixes—but to characterize how closely prompt-generated outputs resemble the solutions selected through BonsAIDE's exploration-and-prune workflow.

Comparison setup. We use the same two QuixBugs-derived buggy inputs used in the study (`find_in_sorted` and `find_first_in_sorted`). For each problem, we generate one output per prompt variant using four representative prompt-engineering strategies adapted from Khojah et al. [23]: *zero-shot*, *few-shot*, *signature-based*, and *persona-based*. The prompt templates and generated outputs used in this comparison are available in our replication package. All

Table 1. Best-matching prompt variant (argmax) frequency across final participant-selected BonsAIDE patches ($N = 10$).

	count	fraction
zero-shot	6	60.0%
persona-based	2	20.0%
few-shot	2	20.0%

Table 2. Distribution of cosine similarities (TF–IDF word unigrams and bigrams) between final participant-selected BonsAIDE patches and prompt-generated outputs, summarized per prompt variant.

	count	mean	std	min	25%	50%	75%	max
zero-shot	10	0.394	0.230	0.037	0.206	0.456	0.561	0.672
few-shot	10	0.138	0.100	0.030	0.076	0.118	0.182	0.359
signature-based	10	0.365	0.210	0.046	0.200	0.416	0.531	0.616
persona-based	10	0.369	0.226	0.033	0.186	0.402	0.514	0.654

candidate solutions considered in this analysis are correct with respect to the bug-fixing task; therefore, this comparison is not about correctness but about how closely prompt-generated outputs resemble human-guided BonsAIDE outcomes.

Similarity metric. We compute a textual similarity proxy using TF–IDF (word unigrams and bigrams) and cosine similarity across all artifacts—prompt-generated outputs and participant-selected BonsAIDE patches—separately for each problem. We visualize this as an all-by-all similarity matrix with four blocks (prompt↔prompt, prompt↔BonsAIDE, BonsAIDE↔prompt, BonsAIDE↔BonsAIDE). Similarity should be interpreted as a coarse proxy for structural/lexical resemblance (not code quality).

Results and patterns. Fig. 7 shows substantial variability across participants and problems. The all-by-all view also highlights that the different prompt types are often fairly similar to each other (prompt↔prompt block), while participant-selected BonsAIDE patches can exhibit additional diversity in surface form; cross-set similarity (prompt↔BonsAIDE) is moderate and inconsistent. In our sample, the *few-shot* output appears more distinct from the other prompt variants in some cases; we interpret this cautiously as an artifact of (i) TF–IDF's sensitivity to lexical choices and scaffolding (e.g., naming, comments, formatting, and idioms) and (ii) our choice to use a single representative generation per prompt strategy as a controlled reference point, where one output can disproportionately. Even when prompt-based generation yields correct fixes, the resulting code frequently differs in structure and expression from the patches ultimately selected via BonsAIDE.

Looking at each participant patch and taking the *best-matching* prompt variant (i.e., the maximum similarity per row), resemblance is moderate and inconsistent (mean 0.442, median 0.490, min 0.070, max 0.672). This suggests that prompt-techniques workflows do not reliably reproduce the same surface form that emerges when participants iteratively branch, compare, and prune alternatives.

The prompt variant that matches best also varies across participants and tasks. In our sample, *zero-shot* most often yields the closest match (6/10), while *few-shot* and *persona-based* are the closest in 2/10 cases each, showed in Table 1. Consistent with this, Table 2 summarizes the distribution of similarities per prompt variant: prompt strategies differ in average similarity and spread (e.g., *few-shot* is lowest on average), but none dominates across all cases.

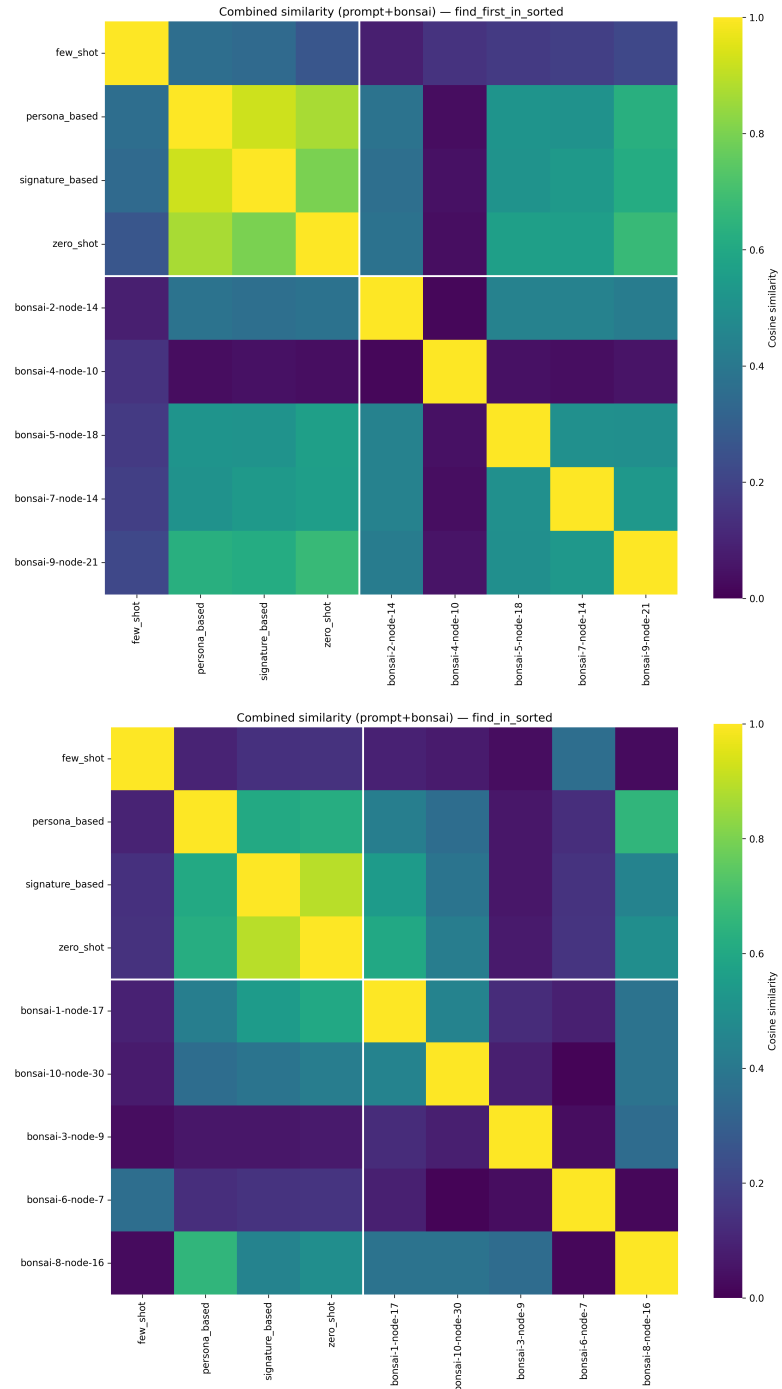

Fig. 7. Prompt-techniques comparison. Heatmaps show cosine similarity across prompt-generated outputs and final participant-selected BonsAIDE patches (all-by-all), per problem. White lines separate prompt and BonsAIDE blocks.

Interpretation. BonsAIDE outcomes are not reducible to prompt-techniques usage, even when using established prompt-engineering strategies. BonsAIDE exposes intermediate alternatives and enables participants to iteratively grow, compare, and prune solution paths, a process that is not captured by static prompt execution. The human-guided exploration and pruning process shapes the final patch in ways that prompt-based generation does not reproduce.

6 DISCUSSION

We now discuss three implications on the results of the study as well assessment on our vision. These discussions also includes the open-ended comments and suggestions by the participants.

6.1 Trees are potential indicators of developer thinking and working styles

As shown in Observation 1, the diversity of BonsAIDE-generated trees reflects the support for varied working styles: users can adopt either breadth-first exploration of alternatives or early commitment to a single path, both within the same tool. Differences in Trim usage—ranging from micro-pruning to macro-pruning—reveal how participants manage cognitive load as the decision tree grows, transforming pruning from a cosmetic action into a strategic mechanism for controlling focus. In some cases, the absence of a *Refactor* action reflects pragmatic decisions in short, correctness-focused tasks, where small changes are deemed sufficient under time pressure.

Overall, these trees serve as a compact, traceable record of explored alternatives, local evidence, and trade-offs—enabling cross-participant comparison and future analysis. Notably, the visual structure of the trees aligns with aesthetic principles such as asymmetry, simplicity, proportion, and depth. These patterns also reflect individual differences in how users represent and interpret the same visualization, underscoring the tool's adaptability to diverse cognitive and stylistic preferences.

Across sessions, participants followed a recurring divergence–convergence rhythm: they began with *Fix the problem*, branched to maintain multiple competing directions, alternated between leaves to compare outcomes, and pruned decisively when a path appeared less promising. Early in sessions, *Generate tests* were used to probe corner cases, while *Handle exceptions* emerged in response to robustness concerns—both reflecting adaptive reasoning. *Refactor* was typically reserved for polishing a promising path, indicating that users prioritized correctness and efficiency over structural transformation.

This process was often summarized as a loop: *"branch, compare, reason, prune, and repeat"*, aligning directly with BonsAIDE's design goal of making exploration explicit, reversible, and traceable. Participants frequently cited *similarity borders* as lightweight cues for next steps: *"It's clear and quick to see the code similarity, and it helps me decide which node to go for" (P3).* When alternatives accumulated, **Trim** was used to focus attention on a manageable set of candidates: *"It's good to prune the bonsai—it's like debugging" (P8).* A minority preferred deep iteration on a single path before branching, but consistently emphasized that the ability to branch later removed the fear of losing progress.

Each activity triggers an automatically generated prompt sent to the backend model, constructed from the current code state, optional user hypothesis, task intent, and constraints, and refined across steps to maintain useful context while minimizing scope and requesting minimal, well-scoped changes with a brief rationale (see §4.2 and [32]). While participants generally welcomed not having to handcraft prompts, they often requested greater transparency—such as a summary of what was requested and how prompts evolved. Comments like *"Show text that describes the main variation against the previous step" (P2)* and *"an explanation of the code, or simply the errors/bugs" (P6)* highlight a clear need for a compact, human-readable prompt summary or delta near the details panel to bridge the gap between intent and generation.

Researchers may apply this concept to gain insight into developer thinking during software development tasks, such as bug fixing. Developers and practitioners can also use it to reflect on and understand the diverse styles they adopt when solving programming problems.

6.2 Comparison cues and metrics support participants' decision-making

Consistent with Observations 2 and 3, participants expressed strong positive feedback on the BonsAIDE prototype.

At selection time, participants evaluated AI-generated solutions using three sources of evidence:

- The *border gradient* helped triage leaves by proximity to the current anchor,
- The *Code view* grounded comparisons in concrete code edits,
- The *Metrics view* provided a quick sense of size and scope (e.g., token count), which some used as a tiebreaker when variants were otherwise equivalent.

Reasoning was used variably: several found short, targeted rationales helpful—*"It feels like a mix of debugging and exploration" (P9)*—while others skimmed or ignored longer explanations, relying instead on code diffs and borders. Requests consistently focused on improving visibility of changes: *"Diff highlighting between candidates would make the decision obvious" (P9).* A few participants also asked for concise textual deltas—e.g., *"Show text that describes the main variation against the previous step" (P2)*—highlighting a need to surface actionable, decision-relevant rationale directly in the interface.

The prompt-techniques comparison (§5.1.5) further supports this interpretation: even when prompt-based workflows yield correct fixes, their similarity to final participant-selected patches is moderate and inconsistent, and no prompt variant consistently matches human-guided outcomes (Fig. 7, Tables 1–2). This aligns with our broader claim that BonsAIDE complements prompt engineering by making exploration, comparison, and pruning explicit, allowing developers to externalize and manage decision-making rather than relying on a single prompt-and-accept cycle.

Complementing related work on AI-as-a-Judge, these insights point to opportunities for future research on how developers make decisions when interacting with AI-generated code. In particular, understanding the strategies developers adopt in controlled settings may inform the design of interfaces that better surface decision-relevant information. Developing a taxonomy of such strategies for tasks like bug fixing could help guide which comparison cues, metrics, and explanations are most useful for supporting human judgment.

6.3 Beyond Pruning

Consistent with Observation 4, participants provided actionable feedback that will inform future iterations and research directions. *Trim* was routinely used to reduce visual clutter without modifying source files. Participants described it as a way to maintain a small, focused working set while preserving the full exploration trace: *"I used Trim to clear the canvas once I was confident which nodes to discard" (P0).* As canvases grew large, navigation friction emerged—several noted issues with zoom/pan behavior and a lack of spatial awareness: *"The graph is difficult to manage; I had some zoom issues" (P4)* and *"Zooming is not working well—it zooms so fast" (P9).* Direct responses to these observations—such as a lightweight mini-map, improved zoom-to-fit, and multi-select Trim—would reduce cognitive load without altering the core interaction model.

Participants frequently issued fixes without anchoring them to a specific session, often proposing two or more alternatives to sample the solution space early. Generating tests served as a key probe to validate or falsify candidates—several explicitly linked their final choice to test outcomes, such as *"Being able to run the generated tests on*

the modified code" (P1), even when execution was outside the current prototype. *Handle exceptions* was applied when robustness or input validation became critical, typically after an initial fix. *Refactor* emerged as a clean-up step to clarify a near-final solution. Participants appreciated seeing these activities encoded in node colors, as it enabled them to quickly reconstruct the sequence of actions and reason about provenance—e.g., *"I wanted to delete the nodes that had only tests but no code" (P4)* when a branch no longer contributed to forward progress.

These results highlight significant research opportunities for improving both functionality and empirical evaluations, to fully capture developer activities. For developers, adoption of this alternative programming paradigm depends on whether they perceive tangible benefits in the new visualization of software engineering tasks.

6.4 Distinguishing BonsAIDE from version control systems

Although BonsAIDE shares surface-level concepts with traditional version control systems (VCS), such as branching and history, the underlying goals and operating assumptions differ substantially. Conventional VCS are designed to record and manage human-authored code changes after decisions have already been made, a limitation that has been observed in prior work on exploratory programming and commit practices [5, 29].

BonsAIDE, in contrast, operates prior to commitment, focusing on decision-making during AI-assisted code generation. Its branching model is AI-centric: branches represent alternative AI-generated candidates that coexist simultaneously as part of an exploration space, rather than isolated lines of development. These branches are intentionally ephemeral and frequently pruned, reflecting exploratory reasoning rather than parallel human workflows.

Another key distinction lies in provenance. While VCS primarily capture what changed between versions, they provide limited support for capturing why changes occur, especially in the presence of AI-generated code [19]. BonsAIDE explicitly preserves prompts, regeneration paths, and lightweight metrics alongside code variants, enabling developers to reason about AI behavior and decision context.

Finally, comparison and pruning are first-class interactions in BonsAIDE. Whereas VCS rely on manual diff inspection after changes are accepted, BonsAIDE supports continuous comparison among alternatives during generation, enabling developers to externalize and manage decision-making before committing to a solution. In this sense, BonsAIDE complements rather than replaces version control systems, addressing challenges that arise specifically in AI-assisted programming workflows.

6.5 Interpreting the prompt-techniques comparison

The prompt-techniques comparison should be interpreted as a *difference-in-outcome* lens rather than a performance benchmark. Because all compared candidates are correct bug fixes, the observed variability in similarity primarily reflects differences in *how* solutions are expressed (e.g., control-flow structure, choice of idioms, placement of checks, and naming) rather than whether they solve the task.

From this perspective, moderate and inconsistent similarity is aligned with the BonsAIDE workflow: participants do not simply accept a single generated completion, but iteratively branch, compare, and prune alternatives based on intermediate evidence (e.g., tests, exception handling, and lightweight comparison cues). Importantly, this suggests that the final patches selected by participants cannot be explained solely as the result of choosing a more effective prompt, but reflect additional human judgment applied throughout the exploration process. Through this process, the final solution can be shaped by the developer's local preferences and reasoning, even when prompt-based strategies are also capable of producing correct fixes.

The all-by-all heatmaps (Fig. 7) further support this reading by making within-set and cross-set resemblance visible: in our sample, prompt-generated variants tend to cluster more closely with each other than with the participant-selected BonsAIDE patches, while BonsAIDE patches can show greater diversity in surface form. However, apparent outliers (e.g., a comparatively distinct few-shot output) should be interpreted cautiously.

At the same time, we emphasize two boundaries of this analysis. First, TF–IDF + cosine similarity is a lexical proxy and does not capture semantic equivalence. Second, our comparison uses one generation per prompt strategy and problem, which provides a concrete reference point but does not characterize the full variability of prompt-generated outputs. These constraints reinforce that the comparison is intended to support the paper's qualitative claims about the role of human-guided exploration and pruning in AI-assisted IDE workflows, not to establish superiority over prompt engineering techniques.

7 THREATS TO VALIDITY

This section discusses possible threats to the validity of our study. Following common guidelines in empirical software engineering [36], we organize them into internal, construct, and external validity.

7.1 Internal validity

Internal validity concerns whether the observed effects are truly driven by the study design rather than by uncontrolled factors. While a brief tutorial and a fixed *fix-first* protocol help mitigate early learning and order effects, residual biases may still persist. Additionally, variations in LLM output due to backend updates or stochastic generation—common in AI-driven systems—pose a risk of non-determinism and model drift; these are addressed through consistent prompting strategies. Usability-related issues such as zoom behavior or latency are acknowledged as potential sources of instrumentation bias, and we report them transparently to ensure methodological rigor. Finally, participant heterogeneity in prior experience with IDEs and AI tools remains, even under standardized instruction, which may influence task performance and strategy selection.

7.2 Construct validity

Construct validity addresses whether the measures used in the study genuinely capture the intended concepts, rather than serving as unintended proxies. While the questionnaire employs compact acceptance items that may limit reliability, the results are presented as descriptive rather than inferential. Self-report data—particularly post-task justifications—can reflect post-hoc rationalizations rather than genuine reasoning; to mitigate this, we cross-validate responses with exported activity graphs to provide a more objective basis for interpretation. Furthermore, perceived ease of task completion does not necessarily correlate with actual correctness or code maintainability, highlighting the distinction between subjective experience and objective outcomes.

In addition, the prompt-techniques comparison relies on TF–IDF + cosine similarity as a proxy for resemblance between patches. This metric captures lexical overlap and some structural cues, but it is not a semantic equivalence measure. We chose TF–IDF here as a lightweight, transparent, and easily reproducible baseline that can be computed directly from the released artifacts without additional model dependencies. Moreover, our current comparison uses one generation per prompt variant and problem; while sufficient to provide an objective point of comparison, repeated runs could further characterize variability in prompt-generated outputs. Furthermore, our prompt set is deliberately small: for each problem we generated one output for each of four prompt-engineering strategies (zero-shot, few-shot, signature-based, and persona-based), using a single prompt formulation per strategy adapted from Khojah et al. Therefore, the

comparison does not exhaust the space of possible prompt techniques or prompt wordings, and it does not characterize variability across repeated generations (e.g., due to sampling randomness or model updates).

7.3 External validity

External validity refers to the extent to which the findings can be generalized to other contexts, tasks, and populations. The sample size of ten participants restricts the generalizability to broader or more diverse user groups. The study focuses on one-file bug-fixing tasks in Python, which may not adequately represent the complexity or scale of larger or multi-language software projects. The single-session design limits insights into long-term adoption, sustained use, or collaborative workflows. While the replication package helps mitigate variability, it cannot fully eliminate differences introduced by changes in LLM models or APIs over time. These limitations highlight the need for further research with larger, more diverse samples and broader task scopes to strengthen external validity.

8 AN AGENDA FOR AN AI-NATIVE IDE

This research agenda outlines seven challenges in AI-driven software development in the IDE, from code provenance tracking to human-AI collaboration. In the following sections, we will explore the research directions, identify the associated challenges, and outline potential research topics that could advance the field. These research directions aim to tackle key challenges in AI-assisted software development, focusing on developer workflows, code quality, and human-AI collaboration.

8.1 Provenance and Code Evolution Tracking:

The ability to trace the origin and evolution of AI-generated code is essential for maintaining a clear development history. In this research we address the challenges of tracking provenance while integrating AI-driven version control systems. Key difficulties include preserving precise provenance—capturing how and why each snippet was produced—and efficiently storing and retrieving prompt-based code evolution graphs without overburdening developers, as well as managing traceability between prompt modifications, AI refinements, and subsequent code changes. Our research agenda focuses on AI-driven version control: determining how AI-generated code can be incorporated into Git workflows or alternative systems, and developing prompt lineage tracking algorithms that monitor and visualize the transformation from prompt to code.

8.2 Intent-Based Code Navigation and Organization

Traditional file-based structures impose rigid hierarchies that constrain AI-generated code, whereas this research explores replacing them with more flexible, intent-driven navigation systems. The primary challenge is to substitute the file-system–based layout with a prompt-driven, context-aware mechanism while ensuring that developers' intuition remains aligned with non-linear, AI-generated structures; another challenge is creating a semantic representation of codebases that permits flexible retrieval and reorganization. Research topics focus on investigating how code can be modeled as an evolving graph rather than static files—enabling developers to efficiently query, browse, and refine AI-generated code—and on AI-assisted refactoring techniques that automatically restructure projects in response to evolving intent.

8.3 Interactive Regeneration and Constraints

To ensure that AI-generated code satisfies a developer's intent, the regeneration process must support fine-grained constraints and iterative refinements. The primary challenges are enabling precise constraints on AI output while preserving developer control, maintaining consistency when multiple AI edits overlap within the same code sections, and creating an interactive, intuitive UI that lets developers guide refinement dynamically. Research topics focus on constraint-based AI generation—introducing semantic constraints into assisted coding so developers can modify generated code at various abstraction levels—and on context-aware regeneration, where models adapt edits to fit the existing style of a project.

8.4 Multiple Generations and Parallel Exploration

AI can produce several distinct solutions to the same problem, yet selecting, comparing, and integrating these alternatives remains a major hurdle. The key challenges are enabling developers to explore alternative AI-generated implementations without incurring decision fatigue or fragmenting the codebase when multiple versions are tested, while also providing context-aware comparisons between generations. Research topics focus on multi-objective code generation—producing diverse solutions optimized for different criteria such as performance, readability, and security—and on AI-assisted decision-making tools that rank, evaluate, and merge alternative implementations, all the while supporting parallel AI branching within a version control system.

8.5 Continuous Regeneration Pipelines

The evolving nature of software demands that AI-generated code remain current as new dependencies, libraries, and best practices emerge. Key challenges involve developing mechanisms to update AI output dynamically in response to these changes while balancing automation with developer oversight—preventing excessive or unwanted modifications—and avoiding unintended regressions when the AI re-generates parts of the codebase. Research topics focus on AI-powered continuous integration—determining how AI-generated code should be incorporated into CI/CD pipelines—and evolutionary AI systems that self-update, maintaining software over time while still allowing human-in-the-loop regeneration.

8.6 AI Code Sandboxing and Safety

Verifying AI-generated code for security vulnerabilities, performance issues, and compliance risks before it is integrated into production systems is essential. The primary challenge lies in ensuring that the generated code is safe, secure, and performant—automatically detecting potential security flaws, unsafe coding patterns, and performance bottlenecks—before deployment. In the context of BonsAIDE, AI Code Sandboxing refers to isolating AI-generated code variants prior to commitment, treating them as provisional artifacts rather than immediately executable or mergeable changes. Each generated branch exists in a sandboxed state where it can be inspected, compared, tested, or discarded without affecting the working codebase. This design allows developers to reason about safety, correctness, and robustness incrementally, before any integration with version control or production pipelines occurs. Research topics focus on automated AI code verification—sandboxing and testing code prior to execution—and on legal and ethical considerations, such as guaranteeing that AI-generated artifacts comply with licensing requirements and adhere to established ethical guidelines.

8.7 Human-AI Collaboration and Cognitive Load

As AI tools become deeply embedded in development workflows, reducing developers' cognitive load is essential to prevent overwhelm from excessive choices. The primary challenges involve avoiding cognitive overload when presenting numerous AI-generated options while designing interfaces that strike a balance between automation and developer control, and developing mechanisms that enhance trust by providing transparency and explainability of the generated code. Research topics focus on human factors in AI—investigating how AI-assisted coding affects developers' cognitive load and decision-making—and on explainable AI in software engineering, aiming to make AI output interpretable and trustworthy while tailoring assistance to individual developers' coding styles and expertise levels.

9 SUMMARY

In this work, we propose a method for evolving and refining software engineering solutions by exploring multiple alternative pathways. As generative AI technologies—such as Large Language Models and AI Agents—become increasingly emergent, there is a growing need to rethink how developers can leverage these tools to enhance and amplify human creativity, fostering innovation. Through the Bonsai concept and the BonsAIDE prototype, we demonstrate that developers naturally adopt diverse strategies when tackling bug-fixing tasks. Our findings also point to key future research directions, including further technical refinements and the exploration of more complex software engineering scenarios to better support real-world development workflows.

ACKNOWLEDGMENTS

We thank all participants in our study for their time and valuable feedback. This research is supported by JSPS Kakenhi (A) JP24H00692.

REFERENCES

[1] Toufique Ahmed, Premkumar Devanbu, Christoph Treude, and Michael Pradel. 2025. Can LLMs Replace Manual Annotation of Software Engineering Artifacts?. In *2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR)*. IEEE, 526–538. https://doi.org/10.1109/msr66628.2025.00086
[2] Stefano V. Albrecht and Peter Stone. 2018. Autonomous agents modelling other agents: A comprehensive survey and open problems. *Artificial Intelligence* 258 (May 2018), 66–95. https://doi.org/10.1016/j.artint.2018.01.002
[3] Saleema Amershi, Dan Weld, Mihaela Vorvoreanu, Adam Fourney, Besmira Nushi, Penny Collisson, Jina Suh, Shamsi Iqbal, Paul N. Bennett, Kori Inkpen, Jaime Teevan, Ruth Kikin-Gil, and Eric Horvitz. 2019. Guidelines for Human-AI Interaction. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems (CHI '19)*. Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/3290605.3300233
[4] Nolan Bard, Jakob N. Foerster, Sarath Chandar, Neil Burch, Marc Lanctot, H. Francis Song, Emilio Parisotto, Vincent Dumoulin, Subhodeep Moitra, Edward Hughes, Iain Dunning, Shibl Mourad, Hugo Larochelle, Marc G. Bellemare, and Michael Bowling. 2020. The Hanabi challenge: A new frontier for AI research. *Artificial Intelligence* 280 (2020), 103216. https://doi.org/10.1016/j.artint.2019.103216
[5] Titus Barik, Kevin Lubick, and Emerson Murphy-Hill. 2015. Commit bubbles. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 2. IEEE, 631–634.
[6] Andrew Bragdon, Steven P. Reiss, Robert Zeleznik, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. LaViola. 2010. Code bubbles: rethinking the user interface paradigm of integrated development environments. In *Proc. ICSE (ICSE '10)*. Association for Computing Machinery, New York, NY, USA, 455–464.
[7] Yuzhe Cai, Shaoguang Mao, Wenshan Wu, Zehua Wang, Yaobo Liang, Tao Ge, Chenfei Wu, Wang You, Ting Song, Yan Xia, Jonathan Tien, Nan Duan, and Furu Wei. 2024. Low-code LLM: Graphical User Interface over Large Language Models. arXiv:cs.CL/2304.08103 https://arxiv.org/abs/2304.08103
[8] P. Chan. 2014. *Bonsai: The Art of Growing and Keeping Miniature Trees.* Skyhorse.
[9] P. Chan. 2018. *The Bonsai Beginner's Bible.* Octopus Books.
[10] Dong Chen, Shaoxin Lin, Muhan Zeng, Daoguang Zan, Jian-Gang Wang, Anton Cheshkov, Jun Sun, Hao Yu, Guoliang Dong, Artem Aliev, Jie Wang, Xiao Cheng, Guangtai Liang, Yuchi Ma, Pan Bian, Tao Xie, and Qianxiang Wang. 2024. CodeR: Issue Resolving with Multi-Agent and Task Graphs. arXiv:cs.CL/2406.01304 https://arxiv.org/abs/2406.01304

[11] Weize Chen, Yusheng Su, Jingwei Zuo, Cheng Yang, Chenfei Yuan, Chi-Min Chan, Heyang Yu, Yaxi Lu, Yi-Hsin Hung, Chen Qian, Yujia Qin, Xin Cong, Ruobing Xie, Zhiyuan Liu, Maosong Sun, and Jie Zhou. 2023. AgentVerse: Facilitating Multi-Agent Collaboration and Exploring Emergent Behaviors. arXiv:cs.CL/2308.10848 https://arxiv.org/abs/2308.10848

[12] Giuseppe Crupi, Rosalia Tufano, Alejandro Velasco, Antonio Mastropaolo, Denys Poshyvanyk, and Gabriele Bavota. 2025. On the Effectiveness of LLM-as-a-Judge for Code Generation and Summarization. *IEEE Transactions on Software Engineering* 51, 8 (2025), 2329–2345. https://doi.org/10.1109/TSE.2025.3586082

[13] Mikhail Evtikhiev, Egor Bogomolov, Yaroslav Sokolov, and Timofey Bryksin. 2023. Out of the BLEU: How should we assess quality of the Code Generation models? *Journal of Systems and Software* 203 (Sept. 2023), 111741. https://doi.org/10.1016/j.jss.2023.111741

[14] Gang Fan, Xiaoheng Xie, Xunjin Zheng, Yinan Liang, and Peng Di. 2023. Static Code Analysis in the AI Era: An In-depth Exploration of the Concept, Function, and Potential of Intelligent Code Analysis Agents. arXiv:cs.SE/2310.08837 https://arxiv.org/abs/2310.08837

[15] Stan Franklin and Art Graesser. 1996. Is it an Agent, or Just a Program? A Taxonomy for Autonomous Agents. In *Proceedings of the Workshop on Intelligent Agents III, Agent Theories, Architectures, and Languages (ECAI '96)*. Springer-Verlag, Berlin, Heidelberg, 21–35.

[16] Cuiyun Gao, Xing Hu, Shan Gao, Xin Xia, and Zhi Jin. 2025. The Current Challenges of Software Engineering in the Era of Large Language Models. *ACM Trans. Softw. Eng. Methodol.* 34, 5, Article 127 (May 2025), 30 pages. https://doi.org/10.1145/3712005

[17] Jesus M. Gonzalez-Barahona. 2024. Software Development in the Age of LLMs and XR. In *2024 IEEE/ACM First IDE Workshop (IDE)*. 66–69. https://doi.org/10.1145/3643796.3648457

[18] Junda He, Christoph Treude, and David Lo. 2025. LLM-Based Multi-Agent Systems for Software Engineering: Literature Review, Vision, and the Road Ahead. *ACM Trans. Softw. Eng. Methodol.* 34, 5, Article 124 (May 2025), 30 pages. https://doi.org/10.1145/3712003

[19] Amber Horvath, Andrew Macvean, and Brad A Myers. 2024. Meta-manager: A tool for collecting and exploring meta information about code. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems*. 1–17.

[20] Yue Hu, Yuzhu Cai, Yaxin Du, Xinyu Zhu, Xiangrui Liu, Zijie Yu, Yuchen Hou, Shuo Tang, and Siheng Chen. 2024. Self-Evolving Multi-Agent Collaboration Networks for Software Development. arXiv:cs.SE/2410.16946 https://arxiv.org/abs/2410.16946

[21] Ziwei Ji, Nayeon Lee, Rita Frieske, Tiezheng Yu, Dan Su, Yan Xu, Etsuko Ishii, Ye Jin Bang, Andrea Madotto, and Pascale Fung. 2023. Survey of Hallucination in Natural Language Generation. *ACM Comput. Surv.* 55, 12, Article 248 (March 2023), 38 pages. https://doi.org/10.1145/3571730

[22] Martin Josifoski, Lars Klein, Maxime Peyrard, Nicolas Baldwin, Yifei Li, Saibo Geng, Julian Paul Schnitzler, Yuxing Yao, Jiheng Wei, Debjit Paul, and Robert West. 2024. Flows: Building Blocks of Reasoning and Collaborating AI. arXiv:cs.AI/2308.01285 https://arxiv.org/abs/2308.01285

[23] Ranim Khojah, Francisco Gomes de Oliveira Neto, Mazen Mohamad, and Philipp Leitner. 2025. The Impact of Prompt Programming on Function-Level Code Generation. arXiv:cs.SE/2412.20545 https://arxiv.org/abs/2412.20545

[24] Tim Krauter, Patrick Stunkel, Adrian Rutle, and Yngve Lamo. 2024. The Visual Debugger: Past, Present, and Future. In *Proceedings of the 1st ACM/IEEE Workshop on Integrated Development Environments (IDE '24)*. Association for Computing Machinery, New York, NY, USA, 1–6. https://doi.org/10.1145/3643796.3648443

[25] Guohao Li, Hasan Abed Al Kader Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. 2023. CAMEL: Communicative Agents for "Mind" Exploration of Large Language Model Society. arXiv:cs.AI/2303.17760 https://arxiv.org/abs/2303.17760

[26] Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. 2017. QuixBugs: A multi-lingual program repair benchmark set based on the Quixey Challenge. In *Proceedings Companion of the 2017 ACM SIGPLAN international conference on systems, programming, languages, and applications: software for humanity*. 55–56.

[27] Mark Marron. 2024. A New Generation of Intelligent Development Environments. In *Proceedings of the 1st ACM/IEEE Workshop on Integrated Development Environments (IDE '24)*. Association for Computing Machinery, New York, NY, USA, 43–46. https://doi.org/10.1145/3643796.3648452

[28] André N. Meyer, Earl T. Barr, Christian Bird, and Thomas Zimmermann. 2021. Today Was a Good Day: The Daily Life of Software Developers. *IEEE Transactions on Software Engineering* 47, 5 (2021), 863–880. https://doi.org/10.1109/TSE.2019.2904957

[29] Hiroaki Mikami, Daisuke Sakamoto, and Takeo Igarashi. 2017. Micro-versioning tool to support experimentation in exploratory programming. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*. 6208–6219.

[30] Khanh Nghiem, Anh Minh Nguyen, and Nghi Bui. 2024. Envisioning the Next-Generation AI Coding Assistants: Insights & Proposals. In *Proceedings of the 1st ACM/IEEE Workshop on Integrated Development Environments (IDE '24)*. Association for Computing Machinery, New York, NY, USA, 115–117. https://doi.org/10.1145/3643796.3648467

[31] Agnia Sergeyuk, Sergey Titov, and Maliheh Izadi. 2024. In-IDE Human-AI Experience in the Era of Large Language Models; A Literature Review. In *Proceedings of the 1st ACM/IEEE Workshop on Integrated Development Environments (IDE '24)*. Association for Computing Machinery, New York, NY, USA, 95–100. https://doi.org/10.1145/3643796.3648463

[32] Mahan Tafreshipour, Aaron Imani, Eric Huang, Eduardo Almeida, Thomas Zimmermann, and Iftekhar Ahmed. 2025. Prompting in the Wild: An Empirical Study of Prompt Evolution in Software Repositories. arXiv:cs.SE/2412.17298 https://arxiv.org/abs/2412.17298

[33] H. Tomlinson. 1995. *The Complete Book of Bonsai: A Practical Guide to Its Art and Cultivation*. WW Norton.

[34] Viswanath Venkatesh, James YL Thong, and Xin Xu. 2012. Consumer acceptance and use of information technology: extending the unified theory of acceptance and use of technology. *MIS quarterly* (2012), 157–178.

[35] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed H. Chi, Quoc Le, and Denny Zhou. 2022. Chain of Thought Prompting Elicits Reasoning in Large Language Models. *CoRR* abs/2201.11903 (2022). arXiv:2201.11903

[36] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in software engineering*. Springer Science & Business Media.

[37] Yuwei Zhao, Ziyang Luo, Yuchen Tian, Hongzhan Lin, Weixiang Yan, Annan Li, and Jing Ma. 2025. CodeJudge-Eval: Can Large Language Models be Good Judges in Code Understanding?. In *Proceedings of the 31st International Conference on Computational Linguistics*, Owen Rambow, Leo Wanner, Marianna Apidianaki, Hend Al-Khalifa, Barbara Di Eugenio, and Steven Schockaert (Eds.). Association for Computational Linguistics, Abu Dhabi, UAE, 73–95. https://aclanthology.org/2025.coling-main.7/