Aporia: Assistant-Facilitated Decision Making for Property-Based Testing

SAKETH RAM KASIBATLA, UC San Diego, USA
SORIN LERNER, Cornell University, USA
HILA PELEG, Technion, Israel
BENJAMIN C. PIERCE, University of Pennsylvania, USA
HARRISON GOLDSTEIN, University at Buffalo, SUNY, USA

AI agents have the potential to help developers by increasing their productivity and supporting a state of flow. However, developers struggle to exercise judgement when using agents, leading to flawed code and concerns about quality. To help developers exercise judgement while reaping the benefits of using agents, we propose a novel interaction mode, *assistant-facilitated decision making*, which makes *implicit* user decisions *explicit*. In assistant-facilitated decision making, an AI assistant elicits user decisions with questions and keeps track of these decisions. Tracking decisions (1) allows them to be used as an input to code generation, (2) helps users clarify their thinking, and (3) helps users consider how different choices impact the final product. As an initial exploration, we implement Aporia, a tool which applies assistant-facilitated decision making in the context of writing custom generators for property-based testing, a known barrier to PBT adoption. We conducted a pilot evaluation, which suggests that Aporia may help users better understand their decisions and the overall codebase, indicating the promise of the assistant-facilitated decision making approach.

ACM Reference Format:
Saketh Ram Kasibatla, Sorin Lerner, Hila Peleg, Benjamin C. Pierce, and Harrison Goldstein. 2026. Aporia: Assistant-Facilitated Decision Making for Property-Based Testing. In *Proceedings of the 15th PLATEAU Workshop on Programming Languages and Human-Computer Interaction (PLATEAU '26).* 11 pages. https://doi.org/XX.XXXX/XXXXXXX.XXXXXXX

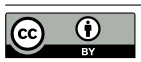
1 Introduction

AI programming tools are changing the way software is written. The change began with the advent of GitHub Copilot [16] in 2022, and now 84% of developers use AI tools, 47% daily [17]. Modern tools go far beyond auto-completion, with *AI agents* [7, 18, 25] now autonomously editing files and executing shell commands. Developers report that using agents can help them achieve flow states [9, 26] and improve their productivity [14].

Despite these potential benefits, present-day AI agents have significant drawbacks. Developers consider programming with agents "fast but flawed" and are concerned about the quality of agent-written code [9]. They struggle to communicate their intent to agents [26], and they hesitate to use agents to implement core business logic, preferring to rely on their own judgement [14].

We believe developers should not have to choose between benefitting from AI assistance and exercising their own judgement about code quality. To this end, we propose a new interaction mode we call *assistant-facilitated decision making*, which centers around making *implicit* user decisions *explicit*. In assistant-facilitated decision making, an AI does not necessarily generate code directly.

Authors' Contact Information: Saketh Ram Kasibatla, UC San Diego, La Jolla, CA, USA; Sorin Lerner, Cornell University, Ithaca, NY, USA; Hila Peleg, Technion, Haifa, Israel; Benjamin C. Pierce, University of Pennsylvania, Philadelphia, PA, USA; Harrison Goldstein, University at Buffalo, SUNY, Buffalo, NY, USA.



PLATEAU '26, Pittsburgh, PA

https://doi.org/XX.XXXX/XXXXXXX.XXXXXXX

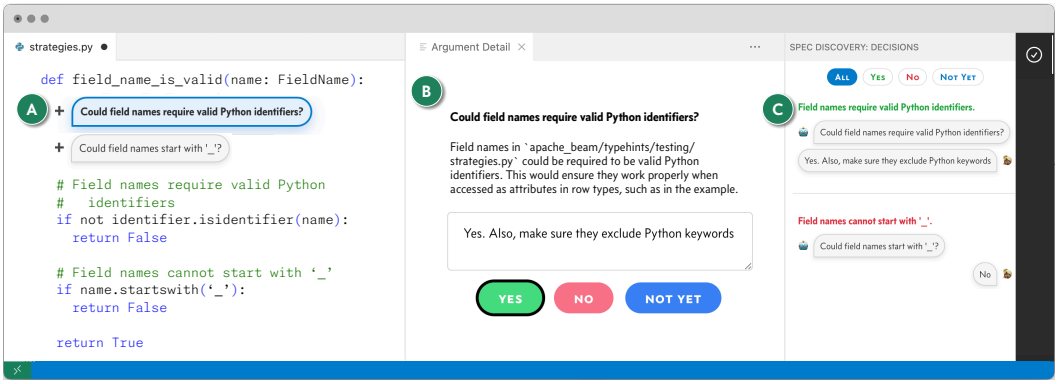


Fig. 1. Aporia interface, shown for a generator for field names. A Aporia asks users *inline questions* about the decision space of their code. B The *argument panel* shows further context for questions, and prompts users to make decisions. C The *decision panel* displays user decisions, allowing the user to inspect and revise their thinking.

Instead, an agent asks the developer questions that clarify their thought process and allow them to articulate concrete decisions about their code. The assistant tracks these decisions, which can then be used in a variety of ways. They can

(1) be used as an input to code generation, giving AI additional context about how code connects to the rest of the codebase;
(2) help users clarify their thinking (*e.g.* by helping identify contradictory decisions); and
(3) enable low-cost "what-if" analyses, letting the user see how the final product changes as their decisions do.

As an initial exploration, we have implemented a prototype of assistant-facilitated decision making in a constrained context where we predicted it would be especially effective: helping users write custom generators for property-based testing (PBT). The process of writing custom generators—recipes that randomly produce values to test—is a known barrier to PBT adoption [11]. When implementing generators, developers must think carefully about the preconditions that must be satisfied by generated data [11]. Consider the generator for field names in Figure 1, an example from a real codebase. The generator produces valid Python identifiers, with the additional constraint that field names must not start with '_'. The generator must fit these requirements because these assumptions are implicitly made in the codebase and tests.

The burden of ensuring that code fits such requirements is placed on developers, who rely on their expertise both to verify generated code and to prompt their AI tools with detailed requirements up front [14]. But this means that the code is only as good as developers' understanding of the requirements, which may change over the course of implementation.

Our tool, Aporia,[1] asks the user questions about design considerations, grounded in evidence from the codebase, such as "Should field names always be valid Python identifiers?" In response, the user makes decisions, which Aporia tracks for future reference. When Aporia generates code, it does so with these decisions as context. The user can revise decisions later, iteratively arriving at their final product.

Our contributions are:

[1]In rhetoric, aporia is a device whereby a speaker expresses a doubt or asks the audience rhetorically how to proceed.

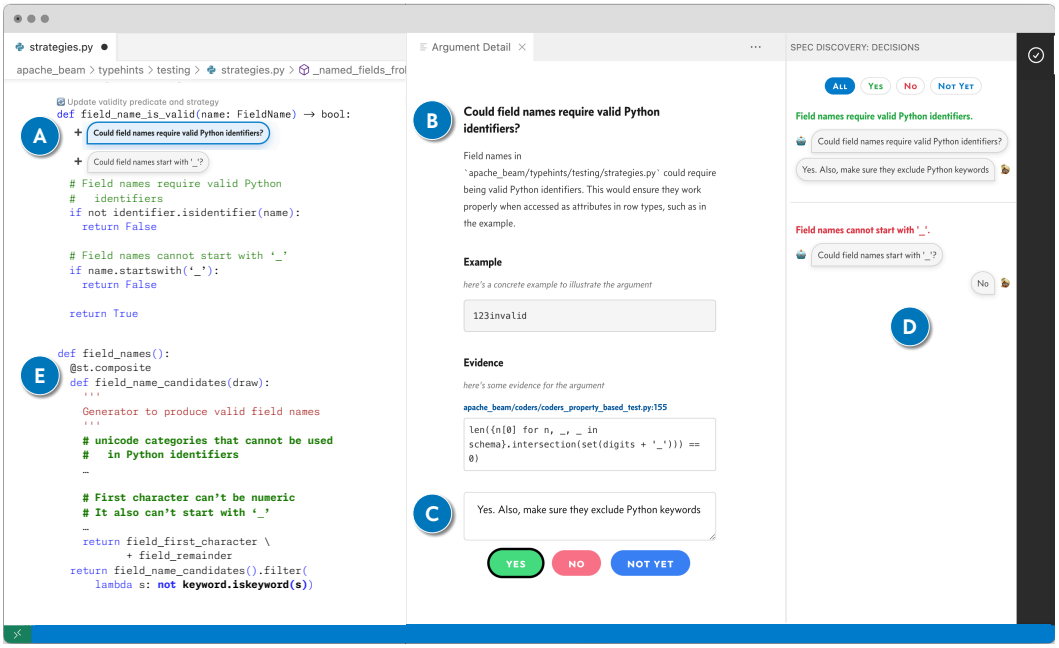


Fig. 2. Writing a custom generator for field names using Aporia. (A) Aporia asks *inline questions* about a validity predicate, which returns `True` if a field name should be generated. (B) Clicking on an inline question opens the *argument panel*, which shows an argument, concrete example, and references to the codebase. (C) The user makes a decision in response to the argument. (D) Decisions are saved in the *decision panel*, where they can be reviewed and edited. (E) Aporia updates the generator with the user's decisions as input.

(1) *assistant-facilitated decision making*, a novel interaction mode for LLM-boosted coding that makes user decisions explicit by having the assistant ask clarifying questions and keep track of choices;
(2) Aporia, an implementation of this interaction mode in an assistant for writing PBT generators, lowering a known barrier to PBT adoption; and
(3) a pilot evaluation suggesting that Aporia helps users better understand their decisions and the codebase overall when implementing generators.

2 Aporia

2.1 Use Case

Consider the scenario illustrated in Figure 2. Chris is a software developer who is contributing to Apache Beam [10], a batch streaming and data processing library. Using the Hypothesis library [24], she has written some property-based tests for code that serializes and deserializes row type constraints. Unfortunately, the tests are failing with field names that will not be encountered at runtime, such as "" (the empty string).

Chris decides she needs a custom generator to produce realistic field names. The codebase is quite complicated and she is not sure what kinds of data she would like to generate, so she opens Aporia to get some ideas. Aporia helps her edit a *validity predicate* (A), which returns `True` if a field name should be generated, and false if it should not. It will use this validity predicate to build the final generator.

Questions inline with code. Aporia inspects the codebase and presents Chris with two *inline questions* (A) about the design space of her code:

(1) "Could field names require valid Python identifiers?," and
(2) "Could field names start with '_'?"

Argument panel. Chris clicks on the first question, and is presented with an *argument panel* (B), which gives more details behind the question. This panel includes a short argument for why Chris might want to require valid identifiers, along with a concrete example and a citation pointing to locations in the codebase that support the argument.

Making decisions. After investigating based on the reference, Chris realizes that the code under test passes the field name to a `NamedTuple`, which requires a valid Python identifier as input. In light of this, she writes that field names should be valid Python identifiers in the text box at the bottom of the argument panel (C), clicking on "Yes" as her answer to the original question. She could also have answered "No" to negate the question, or "Not Yet" to skip answering the question. Aporia saves Chris's decision to the *decision panel* (D).

Upon further inspection of the codebase, Chris also notices that she wants to prevent the generator from producing Python keywords. She clicks on her previous decision and modifies it to note that keywords should be excluded.

Updating code. After inspecting the rest of the questions and deciding that field names should also not start with '_', Chris is ready to try out a generator that fits her decisions. She clicks the "update validity predicate and strategy"[2] codelens (A) . Aporia then adds two conditions—one per decision—to the validity predicate, and rewrites the generator (E) with Chris's decisions as context. Chris now runs her tests, which pass, given the more realistic field names.

2.2 Implementation

Aporia is implemented as a Visual Studio Code extension which communicates with Claude Code[3], and is written in React and Typescript. Aporia works with 2 kinds of data, arguments and decisions. An *argument* consists of a short question; an argument supporting the question; a concrete example; and citations, which can refer to code in the codebase or previously made decisions. A *decision* consists of a reference to an argument; user-provided reasoning; and an answer to the argument's question—"yes," "no," or "not yet". Arguments and decisions are stored in a database accessible to both the extension and Claude Code.

When the user has a file open, Aporia parses all validity predicates in the file. These are formatted in an exit-early style, with a comment before each condition, as shown in Figure 2. After parsing, Aporia invokes Claude Code to generate 2 kinds of arguments:

(1) arguments for *new* conditions to be added to the predicate, and
(2) arguments to *refine* existing conditions, centered around an example that currently does not pass, but should pass.

Claude Code has access to a custom Model Context Protocol (MCP) server, which provides `submit_argument`, a tool that saves arguments to the database. Aporia then renders arguments' questions inline with the code as shown in Figure 2. When a user submits a decision through the argument panel, it is also saved to the database and rendered in the decisions panel. When the "update validity predicate and strategy" CodeLens is clicked, Aporia prompts Claude Code to modify the validity predicate and its corresponding generator, passing user decisions as context.

[2]"Strategy" is a term used in Hypothesis to refer to generators.
[3]Using the claude-opus-4-1-20250805 model

3 Evaluation

We conducted a pilot evaluation to see how users interact with Aporia. This evaluation sought to answer two research questions:

RQ1: How does Aporia affect users' understanding of their generators?
RQ2: How does using Aporia affect users' understanding of their overall codebase?

3.1 Tasks

We conducted a within-subjects study, with participants performing each of two tasks with one of two assistants: the Claude Code CLI or Aporia. We randomized the order in which tasks were performed and which assistant was used with each task. Each task is based on a real-world repository, and is centered around a custom generator for one of its critical data structures:

T1: *Subtitles* is based on the `srt` library [4], which contains utilities for working with video subtitle files. Participants work with `subtitles()`, which generates instances of the `Subtitle` class. It contains 36 property-based tests that use `subtitles()`. We removed all but four, which tested subtitle equality, parsing content with blank lines, filtering of subtitles by timestamp, and sorting collections of subtitles.

T2: *FieldNames* is based on Apache Beam [10] (seen in subsection 2.1), a batch streaming and data processing library. Participants work with `field_names()`, which generates valid field names (strings) to test serializing and deserializing a row type constraint.

We modified `subtitles()` and `field_names()` to naïvely produce examples, erasing any logic the library developers had included to satisfy preconditions for the data structures. Then, for each task, participants were instructed to modify the generator to "produce realistic examples."

3.2 Participants

We recruited four participants, all Ph.D. students. P1, P2, and P4 reported high levels of experience with Python, with P3 reporting moderate levels. P1 and P2 reported significant knowledge of PBT, with P1 having written property-based tests using Hypothesis. P3 and P4 rated themselves as novices in PBT. All participants reported using non-agentic AI tools, with all but P4 having used AI agents. All participants reported low to moderate experience using AI agents. Each participant received a $30 gift card for their participation in the study.

3.3 Study Procedure

Studies were conducted in person with participants performing tasks on the first author's computer. Each session was recorded and transcribed for qualitative analysis. Sessions were organized as follows:

- The *tutorial* (20 minutes) started with a brief introduction to property-based testing, hypothesis, and generators. Participants were then guided through using each assistant to modify a sample generator.
- For each *task* (25 minutes each), participants were shown a description of the task and were instructed to use the assistant as much as possible to complete the task. They also had access to a cheat sheet with instructions on how to use each assistant, and were provided guidance about the location of the strategies, validity predicates, and tests in the codebase. We tracked task completion time in this step.
- After completing each task, participants completed a *post-task survey* (5 minutes each), which asked them five NASA Task Load Index [13] questions, and to rate their confidence (on a 5-point Likert-scale) in the correctness of their answer, their understanding of their decisions,

and their understanding of how the generator is connected to the broader codebase. They were also asked one question to test their codebase comprehension.
- Finally, participants completed a *post-study survey* where they answered questions about their background and about the usability of Aporia. These included Likert-scale ratings and open-ended questions to facilitate a semi-structured interview.

3.4 Results

To get a qualitative understanding of patterns from the study, we conducted a lightweight thematic analysis over our session recordings. While we also collected quantitative metrics, our sample size was too small to see significant differences between Aporia and Claude Code. We plan to iterate on these metrics and increase our sample size in future iterations of the study (see section 4).

RQ1: How does Aporia affect users' understanding of their generators? We collected several self-reported metrics, including the TLX questions, metrics about participants' confidence in the correctness of their answer, their understanding of their decisions, and whether participants felt that Aporia helped them feel more confident in the generators they modified. In the post-study interview, we asked participants to discuss their confidence in their decisions, understanding of generators' details, and situations in which they would prefer Aporia or Claude Code.

All four participants felt that Aporia helped them better understand their decisions about generators. When asked "Did [Aporia] help you better understand what decisions you were making about your code?"[4] P2 said that they liked that Aporia presented them with concrete arguments about the generator. They used these arguments to focus their thinking and better understand different interpretations of the code. For example, they initially assumed that a comment mentioning blank lines meant "lines without any content [at all]," but later realized that the `srt` codebase counted whitespace as blank. P4 liked that decisions were "anchored to the code, and felt granular," and the other participants also mentioned that they liked that each decision felt independent of the others. Looking at the decision panel, P3 actually caught a contradiction in their decisions, noting that their "first decision and later decisions were kind of in conflict."

However, participants' understanding of their decisions did not always translate into a better understanding of their generators' code. P3 and P4, who were less experienced with PBT, were less confident in their generators. P4 noted that they felt confused when reading strategies, as they were unfamiliar with Hypothesis. P3, who felt that Aporia neither helped nor hurt their understanding, mentioned that Aporia de-emphasized the generator in the UI, which led them to focus on getting the tests to pass without spending enough time inspecting the generator. On the other hand, P1 and P2, who had significant knowledge of PBT, reported that Aporia made them feel confident in the generator that they modified. When asked "Did [Aporia] help you better understand the details of the generator you were modifying?" P1 said that explicitly making decisions by clicking on inline questions helped them feel more confident in their generator. In contrast, they said that "Claude Code will make the decisions for you and you just have to say yes."

Overall, participants felt that they were in control when using Aporia. P1 said that if they had used Aporia for T1 (which participants found more difficult than T2), "I would have done worse, but I would have known what I did." When asked about what kinds of situations would be better suited to Aporia vs. Claude Code, P2 and P3 said that Aporia would be more useful when they were unsure about their intent, and P1, P2 and P3 said that Claude Code would be useful when they knew exactly what they needed. P1, P2, and P4 mentioned that it would be useful to use Aporia along with Claude Code in order to get the best of both worlds.

[4]We recognize that this is a leading question. In future iterations of the study we plan to make it more neutral.

RQ2: How does using Aporia affect users' understanding of their overall codebase? The participants felt that Aporia helped them understand the codebase in general, beyond just the generator they were working on. P2 used the inline questions to help direct their exploration of the codebase, referring to the questions to guide them to points of interest when they got confused. P3 said that Aporia encouraged them look more closely at the tests and the functions called in those tests. P4 felt that Aporia's inline questions, which appear directly in the code, helped shift their attention to the codebase rather than a chat window.

This increased awareness of the codebase may, however, come at a cost. P3 observed that Aporia de-emphasized the AI-generated code, leading them to spend less time inspecting it. P1 and P2 remarked that Aporia felt less responsive than Claude Code. We also observed a pattern where participants using Claude Code while performing their task generally spent time reading its output and inspecting its code. This suggests that there may be a tradeoff between emphasizing the AI-assistant's output and encouraging users to engage more with the codebase.

4 Discussion and Future Work

Steering Aporia's questions. While participants enjoyed many aspects of Aporia, they did not always find its questions relevant. P4 said that "it was kind of difficult to recognize my intent in the questions that [...] were available sometimes," and wondered if Aporia could be directed to ask questions around a specific topic. P2 asked if it was possible to focus questions around a particular test case, to dismiss inline questions, and to ask for more suggestions. Adding functionality to let users steer Aporia's questions based on their current goal may support users' focus and improve Aporia's usefulness in clarifying user thinking.

Evaluating the quality of user decisions. During the study, P1 appreciated that Claude Code was able to automatically run tests to see the results of its changes. When asked "What do you wish [Aporia] could do that it doesn't do now?", they suggested that Aporia could run tests in the background when a user makes a decision. In the post-study interview, P2 remarked "I think I understood the decisions that I made. I did not understand if they were the right ones." Having Aporia help evaluate user decisions, perhaps by directly running code, or by interfacing with existing tools such as Tyche [12], could make it easier for users to perform low-cost analyses about how their decisions relate to the final product.

Generalizing Aporia beyond PBT generators. In the post-task survey, participants described Aporia as constrained and niche. As it is currently designed, Aporia asks questions focused on what examples the validity predicate should accept or reject, but Aporia's approach—asking questions inline with code, connecting arguments and decisions to concrete locations in the code, and explicitly displaying user decisions—could also support use cases beyond PBT generators. We are interested in exploring other contexts Aporia could support in future work.

Iterating on user study design. Our pilot study showed several limitations that will inform future iterations of our user study. We designed T1, which had four tests as opposed to T2's single test, to be a more challenging, realistic task. However, participants expressed confusion about the codebase, and about the output format of T1's tests. A future iteration of T1 may benefit from a simplified codebase and more convenient tools to help interpret test results. Also, due to our small sample size (n=4), our pilot was unable to draw a quantitative distinction between Aporia and Claude Code. In a future iteration, we would like to use a larger sample with a mix of expertise with PBT and AI tools. Another challenge in drawing a quantitative distinction between Aporia and Claude Code is the open-ended nature of the decision making process. We plan to refine our correctness metrics and task design in order to address this. Finally, as discussed in subsection 3.4,

we are interested in a potential tradeoff between emphasizing agent outputs and encouraging exploring the codebase. We could examine this possibility by measuring time spent performing tasks such as reading the codebase, running tests, reading AI reasoning, and inspecting AI-generated code.

5 Related Work

AI Programming Assistants. The transformer architecture [31] and large language models (LLMs) [5] enable the variety of AI programming tools we see today. The first of these tools, Github Copilot [16], suggests completions for the line of code the programmer is currently editing. Edit prediction interfaces, popularized by the Cursor IDE [7], predict the next *edit* a user will make, rather than trying to complete the current line of code, *e.g.* adding an import to the top of a file when a user calls a new helper function. Recently, advances have shifted to *AI agents* [22, 25], which can perform multi-step tasks. They do so by using programmatic *tools*, which allow them to interact with an environment to *e.g.* read and edit files and execute shell commands. Andrej Karpathy coined the term *vibe coding* [20] in reference to a mode of interaction where users chat with agents to build software without directly reading or writing code, relying instead on high-level intent specification and automated code generation. In building Aporia, we would like to support user agency in exercising judgement while reaping the benefits of using these tools.

AI-centric IDEs such as Cursor [7] and Windsurf [18] seek to provide a more seamless user experience when interacting with agents. A notable example in this space is Kiro [29], which structures interactions with agents by having the user and agent collaboratively edit natural language documents detailing the requirements, design, and tasks involved in new features. Aporia also structures the code generation process, having the user and agent collaboratively edit a validity predicate. However, the validity predicate describes user intent formally in code, avoiding the ambiguities of natural language.

Studying the usability of AI programming assistants. There are several studies exploring how users interact with completion models. Grounded Copilot [1], which studied programmers using Github Copilot, found that subjects use Copilot in 2 main modes: exploration, where programmers explore their options, as they are unsure of their goal; and acceleration, where programmers' goal is clear, and they use Copilot to achieve it more quickly. Expectation vs. experience [30], which also examined interactions with Copilot, found that while users preferred using Copilot, it impeded their ability to complete tasks.

Recent research has also begun examining how users interact with AI agents. Pimenova et. al. [26] build a grounded theory using semi-structured interviews, reddit threads, and LinkedIn posts. They note that at their best, programming agents can support developer flow, but that agentic programming has many pain points, such as developers struggling to communicate their intent to agents. They also note best-practices that have developed in order to mitigate these pain points. Fawzy et. al. [9] conduct a grey literature review of developer accounts, and find that coding with agents has a speed-quality tradeoff, where developers use agents for enhanced flow, but overlook common QA practices. Sarkar and Drosos [28] analyze video from YouTube and Twitch, identifying an iterative process between prompting AI, reviewing AI-generated code, and manually editing code. They note a shift of programmer expertise from producing code to evaluating code and deciding when to transition between prompting and editing. Huang et. al. [14] focus on how professional software developers use AI agents and discuss strategies that they employ to control agents. They suggest that carefully designed interfaces can help guide software developers to interact with agents more productively. We believe that assistant-facilitated decision making

and Aporia can serve such a role, using agents to help clarify developers' thought process, while providing developers with affordances to better manage agents.

Programming Practices for Open-Ended Tasks. Programmers employ a variety of practices to explore new ideas and develop a better understanding of open-ended tasks.

Exploratory programming involves writing experimental or prototype code, usually in pursuit of open-ended goals that evolve throughout the programming process [3]. Some examples of tools that support exploratory programming are Variolite [21], which features lightweight versioning, and Jupyter Notebooks, a widely-adopted environment for data scientists [15, 19].

In Test-driven development (TDD) [2], expected behavior of code is formalized in the form of tests before any code is written. Behavior-driven development (BDD) [8] is an evolution of TDD that focuses on higher-level concerns like overall system behavior. Aporia also supports open-ended tasks by eliciting and tracking user decisions to make them explicit. It also formalizes user decisions into a validity predicate.

Property-based Testing. Property-based testing (PBT) is a technique for testing executable properties of code with random inputs [6]. Instead of testing one input-output pair at a time, as with traditional unit testing, PBT practitioners specify general properties about their code, and use *generators* to produce random inputs, checking that the properties hold for many inputs. PBT libraries are available for most modern programming languages; the examples in this paper use the Hypothesis library for Python [24]. PBT is an open area of research, including work on how practitioners use PBT in their codebases [11, 27], and on tools to support practitioners [12, 23]. While we aim to enable assistant-facilitated decision making for multiple programming contexts, Aporia is currently centered around helping programmers decide what values to produce when implementing custom generators.

6 Conclusion

This paper presents assistant-facilitated decision making, a novel interaction mode for AI-agents, which centers around making implicit decisions explicit. We demonstrate this interaction mode with Aporia, a tool which applies assistant-facilitated decision making in the context of writing custom generators for property-based testing, a known barrier to PBT adoption. Aporia asks the user inline questions about which examples should be generated, contextualizing these questions with a concrete argument, examples, and references to the broader codebase. Aporia also tracks user decisions, allowing the user to inspect and revise them over time. We conducted a pilot evaluation which suggests that Aporia may help users better understand their decisions and engage more with the codebase overall. Our pilot study also suggests several opportunities to refine the design of both Aporia and our study methodology, which we look forward to exploring in future work.

References

[1] Shraddha Barke, Michael B. James, and Nadia Polikarpova. 2023. Grounded Copilot: How Programmers Interact with Code-Generating Models. *Proceedings of the ACM on Programming Languages* 7, OOPSLA1 (April 2023), 85–111. doi:10.1145/3586030

[2] Kent Beck. 2015. *Test-driven development: by example* (20. printing ed.). Addison-Wesley, Boston.

[3] Mary Beth Kery and Brad A. Myers. 2017. Exploring exploratory programming. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, Raleigh, NC, 25–29. doi:10.1109/VLHCC.2017.8103446

[4] cdown. 2014. srt. https://github.com/cdown/srt

[5] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser,

Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374 [cs.LG] https://arxiv.org/abs/2107.03374

[6] Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. *SIGPLAN Not.* 35, 9 (Sept. 2000), 268–279. doi:10.1145/357766.351266

[7] Cursor. 2023. Cursor. https://cursor.com

[8] Muhammad Shoaib Farooq, Uzma Omer, Amna Ramzan, Mansoor Ahmad Rasheed, and Zabihullah Atal. 2023. Behavior Driven Development: A Systematic Literature Review. *IEEE Access* 11 (2023), 88008–88024. doi:10.1109/ACCESS.2023.3302356

[9] Ahmed Fawzy, Amjed Tahir, and Kelly Blincoe. 2025. Vibe Coding in Practice: Motivations, Challenges, and a Future Outlook – a Grey Literature Review. arXiv:2510.00328 [cs.SE] https://arxiv.org/abs/2510.00328

[10] The Apache Software Foundation. 2016. Apache Beam. https://github.com/apache/beam

[11] Harrison Goldstein, Joseph W Cutler, Daniel Dickstein, Benjamin C Pierce, and Andrew Head. 2024. Property-based testing in practice. ACM, New York, NY, USA, 1–13. doi:10.1145/3597503.3639581

[12] Harrison Goldstein, Jeffrey Tao, Zac Hatfield-Dodds, Benjamin C Pierce, and Andrew Head. 2024. Tyche: Making sense of PBT effectiveness. In *Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology*, Vol. 1. ACM, New York, NY, USA, 1–16. doi:10.1145/3654777.3676407

[13] Sandra G Hart and Lowell E Staveland. 1988. Development of NASA-TLX (Task Load Index): Results of empirical and theoretical research. In *Advances in psychology*. Vol. 52. Elsevier, 139–183.

[14] Ruanqianqian Huang, Avery Reyna, Sorin Lerner, Haijun Xia, and Brian Hempel. 2025. Professional Software Developers Don't Vibe, They Control: AI Agent Use for Coding in 2025. arXiv:2512.14012 [cs.SE] https://arxiv.org/abs/2512.14012

[15] Ruanqianqian (Lisa) Huang, Savitha Ravi, Michael He, Boyu Tian, Sorin Lerner, and Michael Coblenz. 2025. How Scientists Use Jupyter Notebooks: Goals, Quality Attributes, and Opportunities. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering* (Ottawa, Ontario, Canada) *(ICSE '25)*. IEEE Press, 1243–1255. doi:10.1109/ICSE55347.2025.00232

[16] GitHub Inc. 2022. GitHub Copilot. https://github.com/features/copilot

[17] Stack Exchange Inc. [n. d.]. 2025 Stack Overflow Developer Survey. https://survey.stackoverflow.co/2025

[18] Windsurf Inc. 2024. Windsurf. https://windsurf.com

[19] Project Jupyter. 2014. Jupyter. https://jupyter.org

[20] Andrej Karpathy. 2025. Twitter. https://x.com/karpathy/status/1886192184808149383 Tweet.

[21] Mary Beth Kery, Amber Horvath, and Brad Myers. 2017. Variolite: Supporting Exploratory Programming by Data Scientists. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*. ACM, Denver Colorado USA, 1265–1276. doi:10.1145/3025453.3025626

[22] Junwei Liu, Kaixin Wang, Yixuan Chen, Xin Peng, Zhenpeng Chen, Lingming Zhang, and Yiling Lou. 2025. Large Language Model-Based Agents for Software Engineering: A Survey. arXiv:2409.02977 [cs.SE] https://arxiv.org/abs/2409.02977

[23] Muhammad Maaz, Liam DeVoe, Zac Hatfield-Dodds, and Nicholas Carlini. 2025. Agentic Property-Based Testing: Finding Bugs Across the Python Ecosystem. In *NeurIPS 2025 Fourth Workshop on Deep Learning for Code*. https://openreview.net/forum?id=0ajBvBWKrB

[24] David R. MacIver, Zac Hatfield-Dodds, and Many Other Contributors. 2019. Hypothesis: A new approach to property-based testing. *Journal of Open Source Software* 4, 43 (2019), 1891. doi:10.21105/joss.01891

[25] Anthropic PBC. 2025. Claude Code. https://claude.com/product/claude-code

[26] Veronica Pimenova, Sarah Fakhoury, Christian Bird, Margaret-Anne Storey, and Madeline Endres. 2025. Good Vibrations? A Qualitative Study of Co-Creation, Communication, Flow, and Trust in Vibe Coding. arXiv:2509.12491 [cs.SE] https://arxiv.org/abs/2509.12491

[27] Savitha Ravi and Michael Coblenz. 2025. An Empirical Evaluation of Property-Based Testing in Python. *Proceedings of the ACM on Programming Languages* 9, OOPSLA2 (Oct. 2025), 3897–3923. doi:10.1145/3764068

[28] Advait Sarkar and Ian Drosos. 2025. Vibe coding: programming through conversation with artificial intelligence. arXiv:2506.23253 [cs.HC] https://arxiv.org/abs/2506.23253

[29] Amazon Web Services. 2025. Kiro. https://kiro.dev

[30] Priyan Vaithilingam, Tianyi Zhang, and Elena L. Glassman. 2022. Expectation vs. Experience: Evaluating the Usability of Code Generation Tools Powered by Large Language Models. In *CHI Conference on Human Factors in Computing Systems Extended Abstracts*. ACM, New Orleans LA USA, 1–7. doi:10.1145/3491101.3519665

[31] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems*, I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf