Beyond Code Generation: LLM-supported Exploration of the Program Design Space

ANONYMOUS AUTHOR(S)

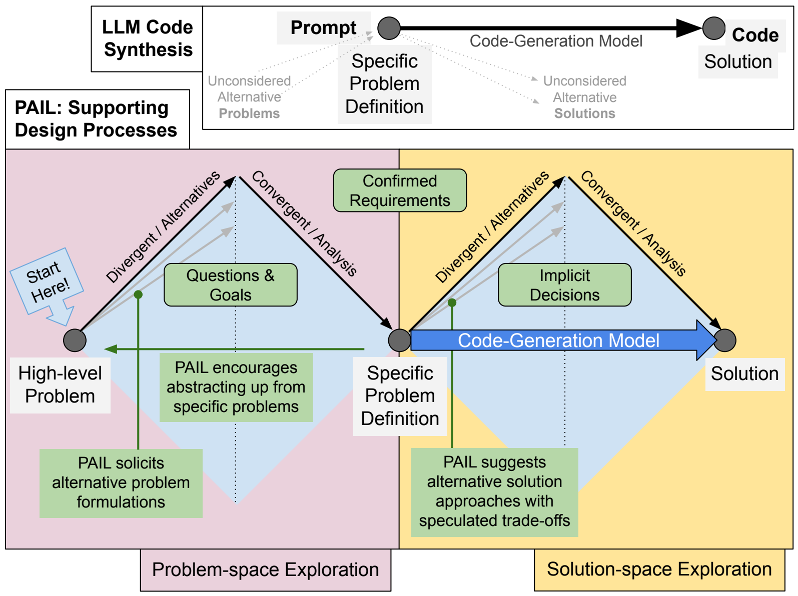

Fig. 1. LLM use in programming has often been formulated as a "prompt"-to-"code" pipeline (top), with the generated code iterated on through updates to the prompt and re-running from scratch. We propose Pail, an IDE that leverages the well-established "Double Diamond" design process (bottom). Pail pulls developers towards a deeper understanding of the problem space, and helps them explore both alternative problem formulations as well as alternative solutions, tracking design goals and requirements, and surfacing implicit decisions (green boxes).

In this work, we explore explicit support for iterative design of computer programs. Program design, like other design activity, is characterized by navigating a space of alternative problem formulations and associated solutions in an iterative fashion. LLMs are potentially powerful tools in helping this exploration; however, by default, code-generation LLMs deliver code that represents a particular point solution. This obscures the larger space of possible alternatives, many of which might be preferable to the LLM's default interpretation and its generated code. We contribute an IDE that supports program design through generating and showing new ways to frame problems alongside alternative solutions, tracking design decisions, and identifying implicit decisions made by either the programmer or the LLM. In a user study, we find that with our IDE, users combine and parallelize design phases to explore a broader design space.

1 Introduction

A common vision of future "programming" relies on LLMs to do it all – identify requirements, write code and tests, perform simulated QA testing, deploy, etc. We are making shockingly rapid progress towards this vision on many fronts [1, 12, 16, 23, 51]. Yet, the role of the human in this process is under-explored. In the predominant view, humans will be limited to setting an initial goal, possibly clarifying a few questions asked, and then rating the output. For some changes to the goal or generated code, the process will restart from, approximately, scratch. Many versions of this vision exist: some scale up the "autocomplete," CoPilot-like [19] interactions relying on recognition-over-recall [45], others scale up the "chat" [1] or focus on providing point solutions [23].

While these visions best suit an understanding of program design as starting with one, known goal, program design in fact involves a design process iterating on both the implementation details *and design goals*. For instance, a journalist may begin an analysis with the goal of assessing a relationship between two variables but then in the process of exploring possible visualizations realize that one of the variables has a lot of missing data. As a result, the journalist can pivot to assess another pair of relationships or consider alternative visualizations and statistical analyses to triangulate the initial relationship of interest. Similarly, a game creator faces questions of narrative, characters, interactions. Even a straightforward backend engineering task (e.g., storage system selection) requires consideration of trade-offs (e.g., latency, consistency, scale) that ultimately require refining the higher-level goals of the task (e.g., terms of a latency service agreement).

Viewing programming as a design activity illuminates the human's need for tighter iterative loops and more fine-grained control than implied by the predominant vision. Design activity begins with consideration of different *problem* formulations before solutions are even discussed—as captured iconically by the British Design Council's *Double Diamond*, which we adapt to illustrate our work's capabilities, in Fig. 1. Designers test high-level hypotheses (e.g., around user needs, or problem urgency) with sketches and prototypes before expending engineering effort. All these exploratory results feed back into a better understanding of the problem and solution design spaces.

Designing interactive artifacts is thus often limited by human bandwidth and time. But LLMs promise a near-free pool of "cognitive resources" that could in theory complement human cognition to dramatically improve the experience of designing and programming interactive artifacts and improve the final artifacts' overall quality—by using LLMs when they excel where humans are limited, and relying on humans when they excel where LLMs are limited.

In this project, we explore how to provide tighter iterative design support when programming with LLMs. In formative work, we found that (1) working through a user-centered design process in chat alone leads to lost requirements and a lot of scrolling through to find questions, decisions, and discussions, and (2) individual point solutions—without consideration of alternatives—led to significant anchoring bias. Combining these findings with knowledge from prior work in AI assistance for design, we developed the Pail IDE. Pail embeds chat with other LLM-based agents into an integrated environment to facilitate higher-level consideration of design choices while programming with LLMs. Specifically, Pail elicits design alternatives, rationales, and prompts to steer users towards user-centered design principles; it tracks requirements and implicit decisions made by the (black-box) LLM without user input; surfaces abstractions that may be useful for establishing common ground; and speculatively explores alternative problem and solution formulations, generating interactive prototypes.

Our work here builds on other recent forays into enabling iteration over generated code with *explicit* design-oriented feedback: Angert et al.'s Spellburst [5] for creative coding relies on LLMs to synthesize or modify code in response to natural language prompting, tracking exploration history in a node-graph style that offers a visual representation of changes over time. Similarly, Kazemitabaar et al. [27] explore LLM-driven code generation for data analysis tasks, exposing "editable assumptions" used to generate code for subtasks, enabling human users to iterate on the code generation *process* directly. Here, we extend these explorations of LLM support beyond creative coding and data analysis, examining the broader question of how to support program design more generally in the process of creating interactive applications, through an interconnected set of affordances aimed directly at supporting the *problem formulation and exploration* tasks we believe represent the next frontier of AI-assisted programming.

In a lab study with 11 participants, we use Pail as a probe to understand the design space of how LLM-enabled programming tools can explicitly support program design. Pail helped participants consider their audience and communication goals; participants expressed appreciation for having a direct summary and for the ability to manipulate those decisions in situ, and 100% of participants found at least one unconsidered alternative that influenced their design work. Further, we find that rapid code generation allows actual interactive programs to fit into a "sketch" role: a cheap, "disposable," [11] epistemological artifact, rather than a "work-in-progress" prototype representing an investment of time and resources—an ability that is suggestive of an altered program design process. Finally, we also see evidence of an emerging issue in managing user attention: as more agents and UI affordances lay claim to being helpful and are integrated into developer environments and workflows, these tools will also need to offer guidance about "where to look."

Our contributions are as follows:

(1) A prototype IDE that introduces a new set of interactions supporting design activity integrated alongside the usual code, visual output, and AI chatbot components.
(2) A qualitative study focused on exploring how and when Pail's design support is useful (and not) for creating interactive software prototypes.
(3) Implications for the design and development of future program design assistance integrating cognitive design aids.

2 Related Work

We draw on empirical studies and theories of design and recent work on LLMs for creative tasks.

2.1 Program Design Processes; Sketching and Comparing Alternatives

In the fields of design and HCI, the process of designing is characterized by exploring the space of solutions and iteratively reformulating the problem to solve. Sketching, a form of rapid prototyping for quickly exploring the essential dimensions of a solution space, is a widely accepted, studied, and practiced tool for thought [11, 33]—serving in part to help designers explore alternatives, sometimes even in parallel [17]. Considering and examining alternatives, as part of a design process, has a long history in HCI: recent examinations in the context of prompting text-to-image models like Promptify [10] and Dreamsheets [3] echo prior systems like Kery et al's Variolite and Verdant for computational notebooks [29, 30], Hartmann et al's Juxtapose system for exploring alternatives in parallel in code [22], and Terry et al.'s *Parallel Paths* approach for parameterized vector art [57]. These systems in turn trace their lineages back to Marks et al's Design Galleries [42] for expensive-to-generate computer images, and are complemented by a set of theory-driven work aimed at understanding how and why these processes help designers [41, 47].

These systems and studies all point to parallel exploration and tracking of alternatives as a well-established, essential process in exploratory programming. [28] In *The Essence of Software*, Jackson argues that software design [25] is equally, if not more, important than software engineering. Software design, like program design, involves consideration of alternatives and, crucially, revisiting and revising the goal of programming in the first place.

2.2 AI Assistance for Programming and Design

Our work here also builds on two major threads of research in AI assistance: assistance for programming, and assistance for design. On the programming side, the biggest impact comes from autocomplete-focused assistance through tools like CoPilot [19], which has been extensively studied [7, 15, 46] and shown to serve both “recognition-over-recall” [45] and epistemic (“oh, I didn’t know about list comprehension in Python!”) goals [7]. Other work has explored how LLMs can be used in the service of design processes for creative coding [5], data analysis [20, 32, 38, 43], and even, circularly, for the assessment of LLM outputs themselves [50].

Pre-LLM work in both AI assistance [13] and the usability of code synthesis [26, 40] is also relevant here, offering insight into the kinds of assistance programmers are looking for: support for writing mundane boilerplate or glue code, for reasoning, and for rapid iteration. Traditional code synthesis has found uses in a number of ways, most notably for HCI through a line of work on programming by demonstration—often repeated demonstrations following a feedback-driven design process, in the service of spreadsheet formula construction [21] and web scraping and automation [8, 36].

This empirical work is complemented by a set of theoretical, speculative, and design-oriented work around the opportunities and challenges of effectively instructing AI systems and designing both with and through them. These include studies of prompting [39, 60, 61], impacts of current LLMs on creative design processes [4, 53] speculative future design processes [58], new conceptual models (e.g., [52, 56], and questions of agency (e.g., [34] and perception (e.g., [31]). This body of work emphasizes the extent to which iteration is critical to design, especially when working with black-box models, that evaluating LLM outputs for correctness is challenging due to intrinsic unpredictability, and that steering LLMs benefits from understanding how LLMs go about performing the tasks a user asks them to perform.

2.3 Workflows Integrating LLMs in Complex Tasks

In order to take advantage of LLM’s capabilities to perform simple tasks well and apply them to more complex tasks, researchers have explored workflows that integrate LLMs. Wu et al. proposed chaining as a technique for connecting LLM calls to each other [59]. Building on this interaction model, Arawjo et al. developed ChainForge [6], an interface for composing LLM prompts and assessing the results of prompts. Exploring alternatives to linear composition, Kazemitabaar et al. [27] compare two different forms of task composition: (i) phase-wise decomposition which batches steps together and (ii) step-wise decomposition which iterates on each step piecewise. Across this work, a common finding is that systems need to scaffold LLM usage in order for users *across experience levels* to make the most of an LLM’s capabilities.

While this set of work also shows us that individual interventions to aid in design, in organization, in evaluation, and in grounding assumptions can all be helpful. Here, we aim to shed some light on what challenges will arise when we begin to build more complex cognitive support tools that integrate multiple of these affordances into a single system. Indeed, research in the AI community has shifted towards developing cognitive architectures for coordinating multiple agents [54], even to complement each other’s strengths and weaknesses [23].

3 Designing Pail

Through this work, we aim to understand how explicit design support can impact programmers’ design and prototyping processes. We adopt a Research through Design approach and develop Pail to probe into what programming with LLMs could look like in the future.

3.1 Design Process

Pail’s primary goal is to facilitate reflection-on-action [49] during program design and prototype implementation. We developed Pail iteratively based on prior work and a formative study. Throughout, we discussed and incorporated feedback among the co-authors.

We started the design process wanting to provide support for generating, keeping track of, and comparing alternative designs. We considered many possible designs for showcasing alternatives and version histories in early prototypes of Pail, including a Git-like "commit" structure inspired by Litt [37]. We ultimately selected a lightweight "pull" model influenced by Kery et al's observation that, in data science workflows, "versioning" in the traditional software sense can be too heavyweight to be useful [30]. We also decided that, as suggested by Lunzer et al. [41], Pail should offer *prospective* (speculative) comparisons of possible alternatives, not only retrospective comparisons of selected alternatives. As a result, we implemented Pail to suggest alternatives whenever possible. Furthermore, based on recent findings in cognitive science on the discovery and usage of abstractions in conversation, we decided to incorporate explicit mechanisms for grounding specific jargon with a description. For example, consider a puzzle game with a specific win condition: explicitly calling out "Win Condition," with a description of what that phrase means, offers users both a specific name to refer to this concept, and confirmation that both the user and Pail are using that phrase to refer to the same sort of underlying concept.

We next conducted a formative study exploring challenges around following design processes for program design when using an LLM-based chatbot. We recruited participants from within our research institution with experience creating interactive software prototypes, and asked them to use ChatGPT and/or Claude to produce prototypes for a variety of interactive programs, pasting any resulting code into a standard `p5.js` environment. Through this process we learned that tracking the outcomes of design discussions was a major challenge, as these discussions rapidly disappeared into chat history and became very challenging to find or recall—and, as decisions receded into the chat history further, the models themselves were also less likely to consider them in future iterations of the design. Additionally, both models limited their responses to one of (1) a single point solution paired with a description of that solution, without discussion of trade-offs or alternative solutions, or (2) several possible solutions, paired with a description, but not comparing solutions with each other or discussing trade-offs among them. These observations reinforced our desire to provide alternatives, with potential impacts prospectively extrapolated, alongside tracking of design discussions and decisions.

While conducting the formative study, we also realized that explicitly surfacing "implicit" decisions (necessarily) made by the LLM when it synthesizes code in response to an ambiguous request could provide a mechanism for considering these decisions and generating possible alternatives. These decisions can be critical to a design, and include decisions about how to represent data, like user progress through an application, or how to operationalize certain constraints, like how to validate a "win condition" in a game. Without surfacing these decisions, discussion only occurred when these decisions they had a visible impact the user noticed and chose to inquire about.

3.2 System Design

What emerged from our design process was the need for two types of components: a set of "agents" that engage directly, such as the ConversationAgent, and indirectly through UI affordances, such as the DesignAgent and ReflectionsAgent, with the user. These agents are complemented with a set of "views" (see below) aggregated into a "design panel" on the right-hand side of Pail (see Figure 3).

The design panel is Pail's primary differentiating feature offering a consistent interface to four design aids, represented as four subsections to the panel, shown in Figure 3:

(1) **DQs** **Design Questions & Goals**: this section tracks the kinds of questions an interaction designer would ask when designing a new interactive piece, including problem formulations,
(2) **Reqs** **Confirmed Requirements**: this section tracks decisions that the human has made or confirmed.

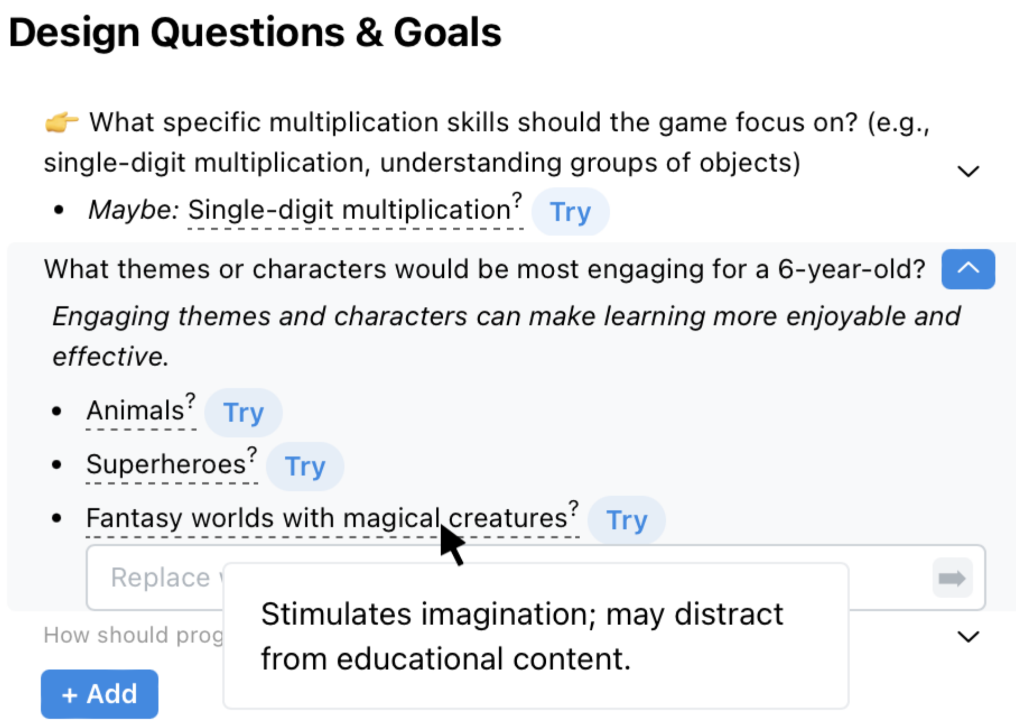

Fig. 2. A screenshot of the DQs *Design Questions & Goals* subsection of the design panel, from P8. The second question ("What themes [...]?") is expanded to show the rationale for that question along with three alternatives. The mouse hover over the third alternative ("Fantasy worlds [...]") reveals a speculated trade-off for trying that option.

(3) Decs **Implicit Decisions**: this section identifies and surfaces decisions the LLM has implemented in the code without explicit confirmation, such as choices regarding how data is structured, or how displayed values are calculated.
(4) UAbs **Useful Abstractions**: this section offers grounding terminology, alongside one-sentence descriptions, for the user and the AI to validate that they are referring to the same concepts when they use project-related language.

Each of these subsections contains a list of items generated by the ReflectionsAgent in response to the ongoing conversation in the chat panel, described below. Each item in the design panel includes a justification ("rationale") for that item as well as 2-3 possible "alternatives." Any alternatives, except some that the ReflectionsAgent identifies as important, are hidden behind an accordion-style view, revealed only when the enclosing item is clicked.

In one study participant's project, within the DQs Design Questions & Goals section, appears the item "What themes or characters would be most engaging for a 6-year-old?" accompanied by the rationale "Engaging themes and characters can make learning more enjoyable and effective." and the alternatives "Animals", "Superheroes", and "Fantasy worlds with magical creatures". Finally, each of these alternatives, in turn, is then speculatively executed by the DesignAgent, which populates each alternative with a sentence about possible cost/benefit trade-offs of that choice. For "Fantasy worlds with magical creatures," for example, the trade-off text is "Stimulates imagination; may distract from educational content." A TRY button next to the alternative triggers the DesignAgent to go ahead and make that change. A screenshot of this portion of the design panel appears in Figure 2. (An additional REVERT button appears in response to clicking TRY, which restores the original code in the IDE; not shown.)

The design panel lies as the rightmost of Pail's four main panels (see Fig. 3): a *code* panel, an *output* panel (with a console for errors and printed output), a *chat* panel, and the *design* panel. The code panel allows users to view and directly modify any project code, and uses a common in-browser code editor, Monaco,[1] containing `p5.js`[2] code that is then run and displayed in the output panel.

The chat panel is the interface to a prompted GPT-4o-based chatbot (ConversationAgent) that can read the project code, patch it, or entirely replace it. Changes to code are summarized within the chat, displayed in "diff" form, and then propagated directly to the code panel, which shows a holistic "diff" over the prior iteration of the project. The ConversationAgent itself is prompted to explicitly guide users through a focused User-Centered Design [2] process: identifying target users, evaluating their needs, assessing possible goals for the project given users' communicated design ideas, and finally generating code for prototypes to test any hypotheses generated through this conversational design process.

The design panel and chat panel operate in a tightly integrated way: conversations in the chat trigger the ReflectionsAgent to update the design panel's contents, while manipulations in the design panel (i.e., trying a specific alternative) trigger code changes via speculative (i.e., uncommitted) executions through the ConversationAgent by the DesignAgent.

3.2.1 System Non-Design. Equally important to the design of Pail is what we did *not* include. Because our focus is on design process support, Pail does not provide assistance with debugging or handling nonfunctional LLM-synthesized code, nor any kind of automated QA or simulated user testing, two areas worth considering for an AI-assisted IDE. We leave investigation of these topics to future work.

3.3 Pail Implementation

Pail is implemented as a React single-page application, proxying calls to GPT-4o through a simple node.js backend that proxies and logs all requests. The code editor uses Microsoft's Monaco code editor set to display code differences inline. The chat component is a custom-built turn-taking component for user communication with the ConversationAgent; it shows chat messages as well as summaries of any code changes made by the ConversationAgent alongside diffs with any changed code.

The code and design panel JSON structures are stored as separate "artifacts" in versioned flat files by the node.js server. To improve latency and offer an *a priori* indication of the impact a particular change might make the DesignAgent speculatively executes a subset of identified alternatives to design panel items.

These speculative executions extend beyond pre-computing the code changes required for that alternative; depending on the nature of the alternative, this could include: (1) how that alternative might impact DQs design goals, such as considerations of target users; (2) what additional Reqs requirements that alternative might reveal; and (3) any new DQs design *questions* that may emerge from consideration of that alternative. Figure 4 shows how agents and artifacts influence one another.

[1]https://microsoft.github.io/monaco-editor/
[2]https://p5js.org

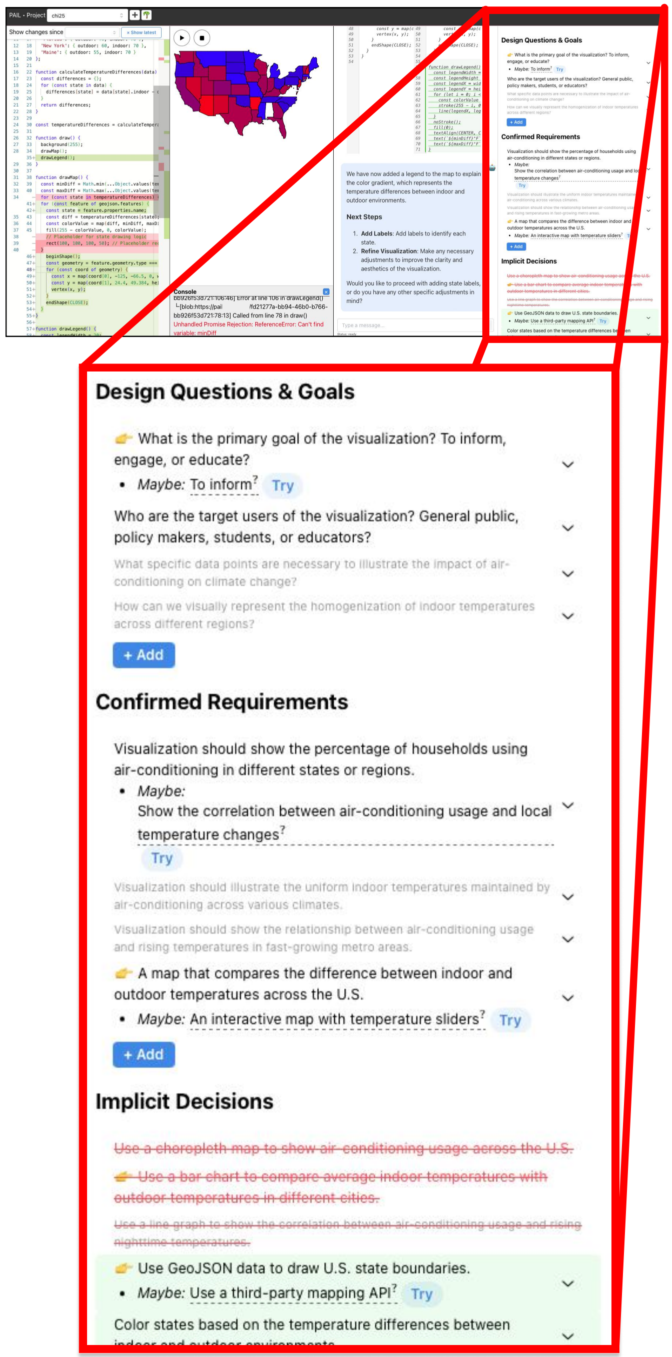

Fig. 3. An exploded view of the Design Panel. Pail engages users in (DQs) defining and refining design goals, including target audience and desired impact, while tracking (Reqs) confirmed requirements and (Decs) decisions implicit in an LLM's synthesized code, and offering explanations and alternatives across the program design space.

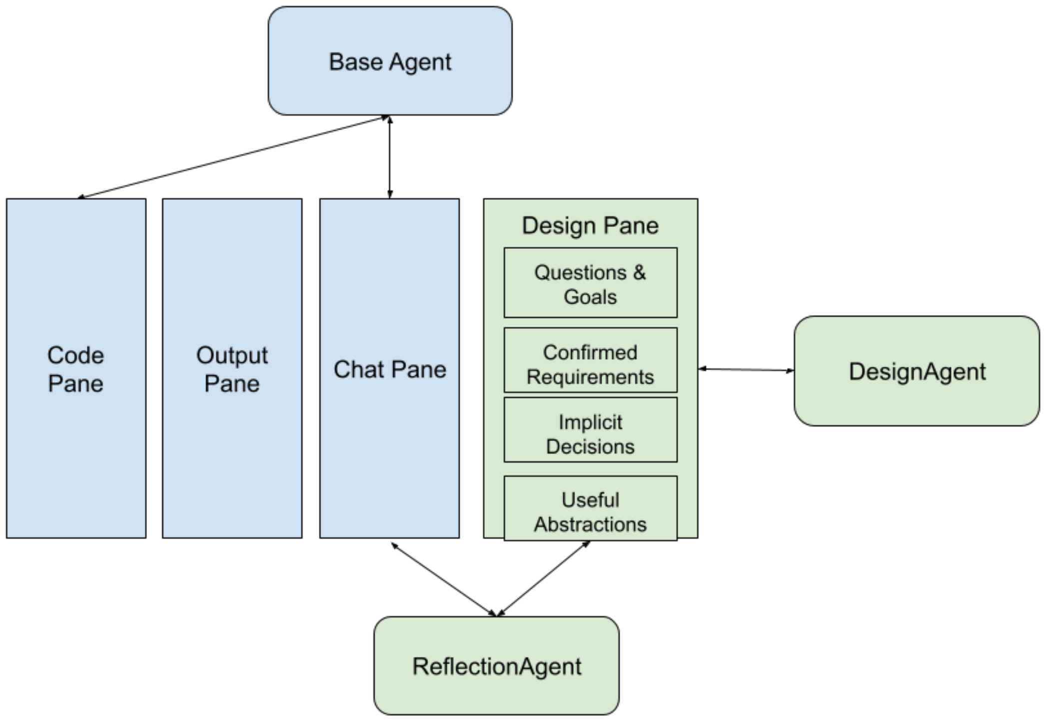

Fig. 4. PAIL consists of 3 agents (CONVERSATIONAGENT, DESIGNAGENT, and REFLECTIONSAGENT), and a 4-pane user interface: code, output, chat, and design. Agents communicate through the data underlying each pane: code, chat messages, and design reflections. The design panel is further segmented in its display to users into four subsections: DQs Design Questions & Goals, REQS Confirmed Requirements, DECS Implicit Decisions, and UABS Useful Abstractions.

4 PAIL Usage: An Example

To make the design and intended use of PAIL concrete, consider the following usage scenario. Sam, an experienced backend system software engineer and parent of two—kids 3 and 5 years of age—is eager to help her 5-year-old, Alex, learn how to read simple words. Sam's tried what feels like all the iPad apps in the store, but none have clicked with Alex. Sam knows she could write a new app, but she's not sure she can commit the time: programming in an unfamiliar environment (e.g., a game environment) often has a learning curve with high variance in time required, and there would likely be a lot of upfront preparation required before Sam could show Alex anything actually interactive—time ultimately wasted if Sam's game ideas don't appeal to Alex.

With PAIL, Sam engages in the folowing scenario:

(1) Sam asks the CONVERSATIONAGENT for help making a game for Alex to practice reading simple words.
(2) PAIL responds with a chat message containing a few questions; a few seconds later, these appear in the DQs Design Questions section of the design panel.
(3) Sam scans the suggestions in the DQs list: about who the user is, what kind of help they need, and what her own goals are for the app, finding the following:

- *What specific skills should the game focus on? (E.g., letter recognition, phonics, simple words)*—with an explicit suggestion she try *Focus on simple words.*
- *What types of activities or game mechanics would be most engaging?*—with the note that engagement will keep the child motivated.
- *Are there any themes or characters that your child particularly likes that we could incorporate?*

See Figure 2 for a screenshot of a similar set of DQs Design Goals.

(4) Sam considers what she knows about Alex, what kind of support she needs, and more. Ultimately she clicks TRY next to the DQs Design Question alternative *Focus on simple words.*

(5) PAIL's DESIGNAGENT notes this suggestion and follows up with another question: *Any kinds of activities your kid enjoys?*

(6) This time, Sam responds in chat: *maybe a word-picture matching game?*

(7) PAIL's CONVERSATIONAGENT offers an idea for an implementation using a drag-and-drop interaction, asking *does this approach sound good to you?*

(8) Sam finds this suggestion confusing. Drag-and-drop isn't an interaction Alex is that familiar with, and in other apps it hasn't worked very well. What are some alternatives? Sam scans the design panel and finds drag-and-drop interaction in the DECS Implicit Decisions section of the design panel. She clicks TRY next to *Use a tap-to-select interface for matching words to pictures.*

(9) PAIL's DESIGNAGENT has speculatively executed on this approach already, and identified that tap-to-select is *simpler for very young children, and can be more intuitive than drag-and-drop*—noting this when Sam was scanning through possible alternatives.

(10) PAIL moves this item up to REQS Confirmed Requirements and produces an initial prototype of the matching interaction: simple words and simple emojis, in a single line. The CONVERSATIONAGENT offers a summary of code changes, and the REFLECTIONSAGENT records that DECS *Words and pictures are displayed in a simplified layout.*

(11) Sam doesn't love this choice, and clicks TRY next to the DECS alternative *Display words and pictures in a grid layout.*

(12) In response to CONVERSATIONAGENT's generated patch to use a grid layout, PAIL's REFLECTIONSAGENT also moves this new requirement to the REQS Confirmed Requirements section.

(13) At this point, Sam shows the prototype to Alex to gather feedback: are the pictures understandable? Are the words too complex, or too simple?

(14) Sam and Alex can then work together to revise the project, incorporating Alex's feedback about what he likes and doesn't like about using the game.

Note that nothing here is beyond Sam's capabilities as a software engineer. She knows how to write code, but is unlikely to find typical "paper prototyping" worthwhile for this use case. PAIL gives Sam the ability to "sketch" with code, overcoming the activation energy required to choose and initialize an environment, figure out a data model and rendering pipeline, or otherwise make a large number of *boilerplate decisions* that must be resolved to have a working program, but serve no other purpose towards Sam's goals around understanding what will help Alex learn to read.

Instead, using PAIL, in 10 minutes Sam has already learned that Alex adores emoji iconography but the targets are too small for his fingers. Sam and Alex can continue using PAIL to iterate on the game, pursuing completely new directions without losing track of the lessons they learned from earlier prototypes. PAIL's suggested alternatives enable

Sam and Alex to break free of anchoring effects or the "sunk code" fallacy of energy invested in authoring code. Using Pail, the energy invested is in considering alternatives, exploring the design space with potential users, and testing prototypes—epistemic actions resolving unknowns, which serve a useful design purpose whether they resolve positively or negatively.

5 User Study Procedure

We had two goals in our user study: (1) identify possible ways the various built affordances helped / did not help users (though not prove definitively); and (2) identify likely challenges inherent to

5.1 Interview Protocol

We began each interview by showing a demo use case for Pail, with an emphasis on what we expect it to be helpful for: designing interactive applications. We showed how to use the chat functionality, explained the tight integration between the chat agent and the code (as mentioned in §3.3, the chat agent can directly modify the code), and then walked users through the various parts of the design panel, explaining each subsection. With each subsection, we showed examples of entries, complete with the rationales, lists of alternatives, and results of speculative execution, and, finally, demonstrated what happened when the user clicks on the TRY button. We finished our introduction by explaining our research question, answering any questions from participants, and then moving on to the first task.

We encouraged participants to think of Pail not only as a *code generation* tool, but also as a *design* tool. We informed participants that they could ask for broad, high-level goals in the chat panel, such as "Help me design an interactive feature that goes along with this article: [pasted article]" (see §5.2, Task 1 below)—that the chatbot was designed to walk them through a design process, considering target users and the participant's own high-level goals.

Because our primary goal is not to measure the effectiveness of Pail from a typical "HCI system evaluation" perspective, but rather to learn what the point design that Pail represents can tell us about the broader design space of AI-assisted programming, we actively encouraged participants to make use of Pail's affordances, and gave them suggestions on when and how to do so. This allowed us to observe a broader range of interactions and impacts than a more traditional lab study, or which might otherwise only be visible after participants develop an expertise using Pail. Naturally, this choice comes with limitations on what we can thus claim.

5.2 Tasks

Our primary criteria for task selection were: ambitious and ambiguous goals; robustness to common LLM failure modes, such as tracking long (or multiple) code files; and enough design focus that we could reasonably observe a diversity of design approaches among our participants.

We ultimately selected 3 tasks for our participants:

Task 1: Create an interactive feature to go along with an article about the impacts of air conditioning on migration patterns in high-average-temperature areas.

Task 2: Create a game that helps a young child learn how to read or multiply.

Task 3: Create a simulation to show the effects of medical overtesting, e.g., recommending screening for specific rare diseases for more people.

For participants who were parents, we started with task 2, and specifically asked for them to consider the needs of their own child and create a game to address one such need, so that we could observe whether having a very specific

ID	Age	Parent?	Professional Background	Design Experience	Programming Experience	LLM Use
P1	40s	Yes	Design Researcher	Professional	Professional	Infrequent
P2	20s	-	HCI Graduate Student	Professional	Professional	Frequent
P3	30s	-	HCI Researcher	Amateur	Professional	Frequent
P4	30s	-	Software Eng Manager	None	Professional	Infrequent
P5	50s	-	Software Eng Manager	Professional	Professional	Moderate
P6	60s	-	Film & Media Artist	Professional	Amateur	Frequent
P7	20s	-	HCI Graduate Student	Professional	Amateur	Infrequent
P8	40s	Yes	Filmmaker & Academic	Limited	Limited	Infrequent
P9	50s	-	Sound Artist & Lecturer	Professional	Amateur	Infrequent
P10	60s	-	Software Engineer	Amateur	Professional	Moderate
P11	50s	Yes	Software Engineer	Limited	Professional	Moderate

Table 1. Study participants.

target user, who is well-known to the designer, has an impact on the design process. Parents were then asked to engage in task 1, if time permitted. All other participants were asked to engage in Task 1 first, then offered a choice of 2 or 3.

Several participants also requested time to engage in an open-ended exploratory task, in which they could experiment with Pail in service of some personally meaningful goal; we supported this where time allowed.

5.3 Participants

Participants were a mix of 5 academics and 6 professionals, of whom 3 were parents. We recruited participants with varying levels of programming expertise, design experience, and prior LLM use; see Table 1. Our sample size was chosen in line with prior work around formative testing for usability [18, 48]: our goal is to explore the early benefits and likely challenges users encounter when engaged in our task using Pail, and our experience with pilot users suggested that we would find benefits and challenges quite quickly.

Our participant pool is skewed towards, but not exclusively consisting of, professionals and graduate students in design- and STEM-related fields—and it is certainly not representative of the population at large. We are not making claims about how a specific population does or does not engage in specific behavior, but rather seek to identify Pail benefits and forecast future challenges among a population that we expect is disproportionately likely to be early adopters of LLM-based tools.

5.4 Analysis & Evaluation

Participants were instructed to think aloud while engaged in the tasks. We then undertook an exploratory data analysis, transcribing all videos and observing where participants directed their attention, where they made design decisions, and what factors appeared to influence them. We also noted where interviewers intervened. We then compared these observations across participants and categorized their approaches and uses of Pail using a well-established affinity diagramming process [44], and through the use of service blueprints [9] that document participants' behavior and reflections. We elected to use these methods from HCI and service design because we seek to understand broadly how different participants engage in similar tasks with a new technology – one that does not have an existing set of users with pre-established work practices they are already engaged in.

6 Findings

In this section, we report findings from our participants' use of Pail. In particular, we find that (1) participants often engage in rapid-fire repeated iteration that sometimes, but not always, makes use of Pail's affordances—and while in this state, activities span all parts of the "4 D's" of design, from problem discover and definition through solution development and delivery, sometimes within the same action; and (2) focused attention was spread quite thinly while participants used Pail, and between changes to code, new messages from chat, and changes to the design panel contents, awareness of what was happening in Pail was easily lost and costly to regain—and that these costs ultimately influenced where participants directed their attention.

6.1 Use of Pail's Design Support

First, we report on ways in which participants used Pail, drawn from in-interview reflections and post-session analysis of Pail use. Across our open-ended program design tasks, participants were quite varied in their approaches.

A common (9/11) initial stumbling block was choosing the first action: though all participants were shown a "project in progress" within Pail with an overview of its various components (see Fig. 3), most (7/11) were not sure at what level of abstraction to make their first request. Should they think of a solution first, and then request that solution? Should they simply provide a description of the design task (and in the case of article-associated interactive task, the article) directly to the ConversationAgent? Though we consistently provided guidance that Pail could help them brainstorm ideas, too, and that they should not feel the need to wait until they had a concrete idea to start making requests, only four participants began by asking the ConversationAgent about the design task directly.

6.1.1 Abstracting Up & Problem Exploration. Regardless of the nature of the initial request, the ConversationAgent would then respond with questions aligned with the design process described above, asking about DQs design goals, target users and user needs, desired outcomes, and often offering some plausible responses for each. For some participants, this would be the first time they would step back to reckon with these factors explicitly; even participants who had thought about what specifically to design often did so without discussing goals or users, but rather brainstorming solutions directly based on what each solution could provide.

All participants interacted with the initial "design process" phase of the ConversationAgent, and all were steered, to varying degrees, by being prompted with these questions. Participants who started by suggesting a specific point design would often use these questions as an opportunity to think more broadly. Even senior programmers with design expertise found something to consider in ConversationAgent's prompting, expressing thoughts like "this really broke it down in an interesting way, who is your target audience" (P5). Later, P5 would reflect:

> I didn't have an idea of what I wanted to do when you first presented me with this problem, and so putting this in and then kind of exploring the "oh, i see, it came up with regions"—and I thought "that sounds reasonable let's start with regions.

The questions could be an awkward fit for some participant-initiated tasks, however. For example, in tasks were not targeted at user needs, the questions seemed not well-targeted to the participant's goals. P6 wanted to use Pail to make a particular "artistic" sketch they had in mind using `p5.js`, with which they were already familiar. Recalling their reckoning with the questions ConversationAgent posed, P6 later reflected:

> Yeah I guess the design element, meaning the "design" as the [...] medium that you're working with here, and having these different questions, requirements, decisions, useful abstractions and things like that, is pretty interesting *[long pause]* But I like it, I like being turned on my ear, you know, it's good.

Some participants also wanted to start directly with an example, rather than by thinking through user groups and user needs. One participant, before even using PAIL, rationalized this desire by noting that it was easier to iterate from a single design than to come up with one from scratch.

Typically, participants would then consider these questions, responding either directly to the CONVERSATIONAGENT, or scanning the summary of questions and possible answers in the design panel and then exploring a subset of REFLECTIONSAGENT-proposed alternatives through the DESIGNAGENT. Participants varied in how they responded to these questions when they did engage. For example, P11 treated the design questions and proposed answers as a checklist to select user groups and features from, clicking the TRY button on all the options that appealed, upfront, and only then ran the generated program.

Reflection. Recall that one of our major goals with PAIL was to encourage thinking about design goals and questions explicitly—on this count, we succeeded for many participants, but not all. It is clear from our study that some users are likely to want to start directly from a point solution, and only then reconsider design goals, target users, etc.—and future tools should consider how to best serve this population, perhaps by starting from a set of easily-comparable point solutions to reduce anchoring.

6.1.2 A Working Sketch: First Contact & Rapid Iteration. For most participants, the first prototype that instantiates a solution, for a sufficiently-formulated problem statement, is revelatory—exposing a number of mismatches between the participant's understanding of the project and either the current state of the code, or the agents' understanding of the project. For example, P9 expected a set of question and answer cards to be shuffled evenly across the canvas, but found them separated by category into distinct question and answer regions. Two other participants (P5, P7) converged on map views with the CONVERSATIONAGENT, but then saw initial prototypes that didn't include maps *per se*, but rather stylized region diagrams with rectangles and triangles representing world or country regions.

These types of mismatches almost always resulted in a flurry of requests for low-level implementation fixes. Participants typically requested these fixes either directly in natural language from the CONVERSATIONAGENT or by finding a suitable alternative in the design panel and trying it. How long participants spent in this rapid-fire local iteration mode varied, from under a minute (P8) to more than 20 minutes (P7), during which their activities resembled some kind of "flow state" [14], in which participants had complete concentration, could express what they wanted as next steps rapidly, noted an effortlessness to the repeated iteration, and with minimal rumination. Participants' think-aloud would often pause in this state.

In this state, participants would often rapidly shift attention across the code, output, chat, and design panel, looking to make sense of what they were seeing, and for how to communicate desired changes most effectively. Those with greater programming expertise, or whose expertise was a closer match with the task domain, unsurprisingly spent more time looking at the code, but not more time editing it directly—rather, seeking confirmation, in the code, of a "lack of surprises" (P3) to validate that the program was being authored in a way that met participants' expectations. Those with less domain expertise (e.g., no familiarity with p5.js, or limited familiarity with authoring interactive artifacts) often ignored the code entirely past a certain point (7/11 participants).

Because none of the agents were designed (nor inclined) to break users out of this state, it would often continue until either some insurmountable "blocker" would interrupt, such as a bug the ConversationAgent couldn't fix, or the user reached a point where they achieved whatever prototype they had set out to achieve, and were uncertain about what to do next. However interrupted, participants would next take stock of where they were, deciding whether to continue making iterative changes, or reconsider the current approach's suitability towards higher-level goals.

Reflection. This rapid iteration was almost always quite broad, and not limited to any one of the traditional "4 D's" of the Double Diamond: the *discover, define, develop,* and *deliver* phases of design. In fact, the same action might serve multiple goals: validating the defined problem while simultaneously progressing towards prototype development, for example, or revealing some new, unanticipated end-user need. To the extent that we expected to support design processes, it appears that rapid iteration with *interactive prototypes*, rather than with paper sketches or other traditional "discovery" methods in design, enables a metaphorical "superposition" of the problem-formulation and solution-exploration phases of design.

6.1.3 The Design Panel, Alternatives, and Rationales. Nearly all (10/11) participants reported finding the design panel useful. Eight participants appreciated the summarization and tracking affordances that enabled quick scans of the (DQs, Reqs) decisions made so far (and updates to those decisions) when new chat messages began to exceed in length how much participants desired to read. Seven participants expressed appreciation for the reporting of rationales and alternatives. Though no participants explicitly called out appreciation towards the Decs implicit decisions or UAbs useful abstractions components of the design panel, we nonetheless observed several participants (P3, P5, P8, P9, P11) drawing insight from these components. P2, for example, explicitly drew a comparison with their prior ChatGPT experience:

> I did try to use ChatGPT for that and it was...fine. [...] It looked [worse] than the one we just built, and it took me longer. [...] [In ChatGPT] I didn't have the right mechanism to high-level changes at this level of abstraction.

Scanning the alternatives lists (and ultimately selecting an alternative to try) typically emerged after some initial trigger, as when a participant would pause, unsure of what next step to take—in a broader sense than just making the next single decision. For example, at one point P10 recognized that their approach to visualizing one particular set of interrelated values (in their case, temperature and air conditioning usage) wasn't going to work, and they weren't sure where to take the project next. In this and other similar cases, the decisions and lists of alternatives offered a lower-cognitive-load path forward, by allowing participants to *select one of several possible options* rather than *generate a new idea entirely* as a next step, echoing the recognition-over-recall UX design heuristic [45]. Scanning these lists would often result in focus on a single choice, eliciting a reaction like "oh that seems like a good idea" (P4)—or would yield an alternative not directly on the lists, but still inspired by the scan, which participants would then suggest either in the chat or in the alternative lists' open text field titled "Replace with...".

A few participants (P5, P7, P11), in "flow state" (see §6.1.2), preferred to use the design panel's alternatives almost exclusively to guide their exploration, avoiding chat. Asked why, P5 reported that it was easier to answer multiple-choice questions than to repeatedly write messages in chat—treating the lists of alternatives in the DQs design questions section as a sort of "design checklist," a menu from which to select target users, goals, and more.

Reflection. To the extent that we expected the design panel to encourage participants to consider alternatives *prospectively*, our findings here and in the previous subsection suggest we could do better: participants only rarely actively sought out new problem formulations unless prompted either by Pail, by an interviewer, or by a realization that their current design approach was not going to lead to success.

6.2 Stumbles, Mismatches in Participants' Design Processes

Beyond participants' use of Pail's specific affordances, we also observed behavior that bears on the design of future AI assistance for program design; here, we detail those observations.

6.2.1 Re-considering Design ***Problems****.* Though Pail explicitly elicits design considerations from participants at the outset of project construction, there is no explicit support to bring users to *reconsider* design questions and goals during the implementation process. As a result, few participants explicitly reconsidered the highest-level design directions within Pail, at least without explicit prompting from an interviewer. In fact, in the "flow state," many participants would fixate and iterate on small details (e.g., colors, item position, text content) repeatedly. Meanwhile, the ConversationAgent was happy to support this low-level iteration for at least as long as interviewers allowed.

But it was *emphatically not the case* that participants remained fixed in their beliefs about user needs and which is the right problem to solve—they simply did not reconsider their design goals *explicitly in conversation* with the ConversationAgent. Instead, these realizations would come from specific prototypes that yielded specific forms of insight, such as whether particular problem formulation could be compellingly addressed. For example, P10 at one point reflected: can an end-user actually be convinced of a causal relationship between air conditioning and climate change through a bar chart—or should the interactive feature instead focus on a different climate-related relationship?

6.2.2 Content Generation and Interface Overwhelm. Nearly every participant at some point commented on the overwhelm generated by Pail. There is too much data, it is being generated too fast, it's too hard to look at everything, and it's not clear what one should be looking at. As a result, participants reported, much effort was spent figuring out where to look and how to evaluate changes. Running the project was almost always the top choice, but that did not always work. Sometimes, program behavior was not trivial to reproduce; bugs prevented visual output; or the program simply could not be run because the ConversationAgent was in the process of updating it, which could take longer than a minute for large changes. While waiting, participants often scanned the design panel, or the "diff" view of the code, to understand the scope of recent changes or consider what steps to take after the code updates have completed.

In P5's words: "Like, it's asking me so much in here, I'm not gonna read it every time."

6.2.3 Application of Reflection and Design Agents. Third, participants did not appear to make substantial use of the rationales the ReflectionsAgent and DesignAgent provided for why certain decisions were made or what trade-offs would result form selecting a particular alternative. In part this appeared to be because participants found it *easier to simply try an alternative* than to consider whether the provided rationale was valid and relevant—a version of "show, don't tell." [55]

6.2.4 Attention, Expertise, and the Cost of Awareness. As the contents of the code, chat, and design panes updated "automatically" through agent updates, staying on top of the latest updates to any particular pane required substantial attention. Once lapsed, this "awareness" was costly to regain. The exact cost depended on expertise, all else being equal. For example, senior programmers rapidly lost awareness of the code, and their expertise helped them regain it quickly when needed. The cost of regaining awareness depended on participants' choices for where to direct attention. If the code felt "hopeless" (P1), or "unfamiliar" (P5), regaining awareness became a priority only when a participant encountered a bug or issue. P5 described the experience of clicking TRY and watching the code change in response, in the following way:

> Each time I click on something here [in the design panel] [...] I'm like "Ah! What part of this is important?"

Both domain and programming expertise played a role in mitigating those costs, and thus in how effectively participants could stay on top of changes to code and design. This effect manifested in a few ways: first, domain expertise helped participants more easily recognize the overall "shape" of code components as they came in, making it easier to stay on top of changes with lower self-reported cognitive demands. For example, one participant with data science expertise (P3) could recognize the boilerplate data formatting of sample data as it was generated, but found code for a simulation harder to stay on top of—while another participant with creative coding expertise (P7) found `p5.js` code straightforward to retain awareness of.

We observed how participants expecting certain code to come from chat or design panel requests watched as that code streamed in from the ConversationAgent and then engaged in reflection-in-action [49], expressing surprise (or dismay) when these results deviated from expectations.This behavior echoes the observations by Barke et al. [7] of experienced programmers using GitHub Copilot.

Lastly, some participants (P6, P7, P8) rapidly formed a clear, persistent vision for at least one requested task—and rarely, if ever, found themselves actively seeking design alternatives, feedback, or even an understanding of implicit decisions while in the "flow state" of trying to achieve that vision. In the case of P8, the interviewer switched one task's development context from Pail to Anthropic's *Claude* AI system, which could generate and run code that was more aligned with the participant's vision at a more rapid pace than Pail. This approach was an attempt to understand whether P8 would reach a "saturation point" where they were satisfied that their vision was achieved, and design support might be welcomed. Though a saturation point was reached, the desired design support was explicitly limited to "I'm not really interested in what the system might tell me, the only thing I'd want to do is try it with [my child]" (P8).

7 Discussion

Our goal in conducting this work is in identifying the next set of challenges the HCI community is likely to face when it comes to helping programmers and other technologists design working programs with AI assistance. Based on our experience designing Pail and the evaluation results, we identify open challenges for LLM-aided program design around(1) how to define design goals and problems and (2) how to manage the trade-offs between providing *more complete* information and providing *more relevant* information from the very large set of information that LLMs and other AI systems can inexpensively and rapidly deliver.

7.1 Defining Goals, Exploring Problem Spaces

In order for LLMs to effectively support program design, they must identify steps along the pathway towards a user's ultimate design goal that match a user's mental model of the problem space. However, as we saw in our evaluation, what steps are meaningful and how important they are depends on where users begin. If users have a point design to begin with, the appropraite next step may be to extrapolate key features before exploring alternatives. In contrast, if users do not know where to begin, a the right next step may be to name a design dimension before exploring concrete instances of it. Indeed, these approaches are characteristic of the double diamond of design as reflected in Figure 1.

But though the double diamond implies a certain linearity, we found that Pail's ability to rapidly generate code along many different directions enabled a very nonlinear approach to design. Participants would rapidly shift between exploring possible solutions and recognizing that their problem formulations may not have been addressable. For LLM-aided tools to be effective in helping users with design, they should facilitate *and gain awareness of* this rapid movement between extrapolation and concretization that is enabled by their code generation capabilities.

7.2 Managing "More Information" *vs.* "Better Information"

Here, our observations of overwhelm strongly reinforce the need for future systems to be designed with an understanding of what feedback is useful, when, and through what mode of delivery. This is not a new problem. Horvitz identified that a major challenge in automation is the selection of an "ideal action in light of costs, benefits, and uncertainties" [24] as early as 1999.

While introducing design-related "knobs" in Pail helped users consider their approach, Pail sometimes overwhelmed designers with too much information with a low signal-to-noise ratio. This risks that users will ignore the feedback, even when structured to support design. Ultimately, making sense of the large amount of *potential* information generated by LLM-based agents and other cognitive tools will require a new layer of interaction between human users and the underlying agents producing these insights. It's not clear who will wield that power.

Li et al. [35] have argued that tool designers wield a lot of power to shape thought and practices in the domain of creativity support tools. Prior work (e.g., [58]) suggests that LLM-assisted techniques like dynamic grounding can return some of the power to users' hands by allowing tools to *adapt to where humans are*, rather than forcing humans to adapt their ways of thinking and practices to the tools' capabilities as defined by their designers.

Our Pail experiences, however, raise the concern that we are on the brink of handing a lot of power to the IDEs, LLMs, and prompt developers building the next generation of tools, because each new LLM-based design affordances demands attention, and showing them all at once will require a major learning curve. Who directs attention, if not the system? We may be moving from a formal system of rules and practices that are at least discoverable and interpretable, to an analog world of prompt-driven components whose very behavior is both inherently unpredictable and also dependent on the model it happens to be executed against. Our experiences with attention overload in other realms (e.g., social media) suggest that this future may be less empowering for users and developers, rather than more.

8 Limitations and Future Work

There are two key limitations of this work that future work should address: the generalizability of Pail and transitions across levels of abstraction when programming with LLMs.

First, it bears mentioning that Pail isn't intended for for every programming task, nor for every programmer. Though Pail helped get participants started regardless of design background, *some* level of design process literacy is required to make effective use of Pail. Design process literacy can be taught, of course—or it can be designed into tools like Pail. For example, recall that participants were most willing to engage with ConversationAgent's questions about the end-user and their needs when the agent was *not* simultaneously producing changes to the program that a user could be testing—providing code in this context was actively counterproductive. This observation points to a compelling, but also concerning, mechanism by which a tool like Pail could encourage more consideration of various parts of the design process: hold the project hostage by simply refusing to produce any code until the user has considered what the tool wants the user to consider.

Second, regardless of approach, designers need to operate across abstraction levels, knowing when to pop up to a high level from the weeds, and when to deep dive towards a point solution in order to better understand a particular neighborhood within the design space. Pail currently supports only two levels within its hierarchy of abstraction: code and design-related concerns. As recently noted by Vaithilingam et al [58], design decisions happen in a fractal pattern, with many decisions across many levels of abstration. Future LLM-aided programming tools should support hierarchical levels of abstraction to provide greater control over and understanding of generated code, as well as more targeted legibility for the immediate subtask at hand.

Supporting users identify design problems and solutions, as they handle an information overload and transition across abstraction levels, requires we tool designers to reckon with several major open questions:

(1) How much agency or initiative should automated systems be given to set direction? It's one thing to synthesize some code for a user to evaluate—it's another entirely to control when a user considers their high-level goals for a project instead of staying lost in the weeds, or to scope out a set of potentially-anchoring alternative points in the design space.
(2) If indeed a major emerging challenge is managing *attention* in the face of *too much data* and *unknown signal-to-noise ratios* among that data, what can be done to address this?

9 Conclusion

Through this work, we explored the implications of LLM-aided program design, focused on support for problem formulation and assessing solution suitability. Pail, our design probe, encourages developers to follow a user-centered design process and tracks requirements discovered and decisions made through prototyping. Through our user study, we found some evidence that this kind of assistance can be helpful in broadly considering the program design space, but also uncover a set of challenges around managing attention and maintaining awareness of program updates, pointing to broader questions and trade-offs across generating and sharing information, ensuring information is relevant to users, and balancing agency between users and their new generation of tools.

10 Disclosure

The authors used ChatGPT for minor copyediting tasks.

References

[1] 2024. Artifacts are now generally available. https://www.anthropic.com/news/artifacts
[2] 2024. What is User Centered Design (UCD)? https://www.interaction-design.org/literature/topics/user-centered-design
[3] Shm Garanganao Almeda, J.D. Zamfirescu-Pereira, Kyu Won Kim, Pradeep Mani Rathnam, and Bjoern Hartmann. 2024. Prompting for Discovery: Flexible Sense-Making for AI Art-Making with Dreamsheets. In *Proceedings of the CHI Conference on Human Factors in Computing Systems (CHI '24)*. Association for Computing Machinery, New York, NY, USA, 1–17. https://doi.org/10.1145/3613904.3642858
[4] Barrett R Anderson, Jash Hemant Shah, and Max Kreminski. 2024. Homogenization Effects of Large Language Models on Human Creative Ideation. In *Creativity and Cognition*. ACM, Chicago IL USA, 413–425. https://doi.org/10.1145/3635636.3656204
[5] Tyler Angert, Miroslav Suzara, Jenny Han, Christopher Pondoc, and Hariharan Subramonyam. 2023. Spellburst: A Node-based Interface for Exploratory Creative Coding with Natural Language Prompts. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology (UIST '23)*. Association for Computing Machinery, New York, NY, USA, 1–22. https://doi.org/10.1145/3586183.3606719
[6] Ian Arawjo, Priyan Vaithilingam, Martin Wattenberg, and Elena Glassman. 2023. ChainForge: An open-source visual programming environment for prompt engineering. 1–3.
[7] 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 (2023), 85–111. Publisher: ACM New York, NY, USA.
[8] Shaon Barman, Sarah Chasins, Rastislav Bodik, and Sumit Gulwani. 2016. Ringer: web automation by demonstration. 748–764.

[9] Mary Jo Bitner, Amy L Ostrom, and Felicia N Morgan. 2008. Service blueprinting: a practical technique for service innovation. *California management review* 50, 3 (2008), 66–94. Publisher: SAGE Publications Sage CA: Los Angeles, CA.

[10] Stephen Brade, Bryan Wang, Mauricio Sousa, Sageev Oore, and Tovi Grossman. 2023. Promptify: Text-to-Image Generation through Interactive Prompt Exploration with Large Language Models. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology (UIST '23)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3586183.3606725

[11] Bill Buxton. 2010. *Sketching user experiences: getting the design right and the right design.* Morgan kaufmann.

[12] 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. https://doi.org/10.48550/arXiv.2107.03374 arXiv:2107.03374 [cs] version: 2.

[13] Yan Chen, Steve Oney, and Walter S. Lasecki. 2016. Towards Providing On-Demand Expert Support for Software Developers. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems (CHI '16)*. Association for Computing Machinery, New York, NY, USA, 3192–3203. https://doi.org/10.1145/2858036.2858512

[14] Mihaly Czikszentmihalyi. 1990. *Flow: The psychology of optimal experience.* New York: Harper & Row.

[15] Paul Denny, Viraj Kumar, and Nasser Giacaman. 2023. Conversing with copilot: Exploring prompt engineering for solving cs1 problems using natural language. 1136–1142.

[16] Giulia Di Fede, Davide Rocchesso, Steven P Dow, and Salvatore Andolina. 2022. The Idea Machine: LLM-based Expansion, Rewriting, Combination, and Suggestion of Ideas. In *Creativity and Cognition*. 623–627.

[17] Steven P. Dow, Alana Glassco, Jonathan Kass, Melissa Schwarz, Daniel L. Schwartz, and Scott R. Klemmer. 2010. Parallel prototyping leads to better design results, more divergence, and increased self-efficacy. *ACM Transactions on Computer-Human Interaction* 17, 4 (Dec. 2010), 1–24. https://doi.org/10.1145/1879831.1879836

[18] Laura Faulkner. 2003. Beyond the five-user assumption: Benefits of increased sample sizes in usability testing. *Behavior Research Methods, Instruments, & Computers* 35 (2003), 379–383. Publisher: Springer.

[19] GitHub. 2021. GitHub Copilot • Your AI Pair Programmer. http://github.com/features/copilot

[20] Ken Gu, Madeleine Grunde-McLaughlin, Andrew McNutt, Jeffrey Heer, and Tim Althoff. 2024. How Do Data Analysts Respond to AI Assistance? A Wizard-of-Oz Study. In *Proceedings of the CHI Conference on Human Factors in Computing Systems (CHI '24)*. Association for Computing Machinery, New York, NY, USA, 1–22. https://doi.org/10.1145/3613904.3641891

[21] Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. *ACM Sigplan Notices* 46, 1 (2011), 317–330. Publisher: ACM New York, NY, USA.

[22] Björn Hartmann, Loren Yu, Abel Allison, Yeonsoo Yang, and Scott R. Klemmer. 2008. Design as Exploration: Creating Interface Alternatives through Parallel Authoring and Runtime Tuning. In *Proceedings of the 21st Annual ACM Symposium on User Interface Software and Technology (UIST '08)*. Association for Computing Machinery, New York, NY, USA, 91–100. https://doi.org/10.1145/1449715.1449732 event-place: Monterey, CA, USA.

[23] Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. 2023. MetaGPT: Meta Programming for A Multi-Agent Collaborative Framework. https://openreview.net/forum?id=VtmBAGCN7o

[24] Eric Horvitz. 1999. Principles of mixed-initiative user interfaces. In *Proceedings of the SIGCHI conference on Human factors in computing systems the CHI is the limit - CHI '99*. ACM Press, Pittsburgh, Pennsylvania, United States, 159–166. https://doi.org/10.1145/302979.303030

[25] Daniel Jackson. 2021. *The Essence of Software: Why Concepts Matter for Great Design.* Princeton University Press.

[26] Dhanya Jayagopal, Justin Lubin, and Sarah E Chasins. 2022. Exploring the learnability of program synthesizers by novice programmers. 1–15.

[27] Majeed Kazemitabaar, Jack Williams, Ian Drosos, Tovi Grossman, Austin Henley, Carina Negreanu, and Advait Sarkar. 2024. Improving Steering and Verification in AI-Assisted Data Analysis with Interactive Task Decomposition. https://doi.org/10.1145/3654777.3676345 arXiv:2407.02651 [cs].

[28] Mary Beth Kery. 2017. Designing Effective History Support for Exploratory Programming Data Work. (2017).

[29] 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 (CHI '17)*. Association for Computing Machinery, New York, NY, USA, 1265–1276. https://doi.org/10.1145/3025453.3025626

[30] Mary Beth Kery, Bonnie E. John, Patrick O'Flaherty, Amber Horvath, and Brad A. Myers. 2019. Towards Effective Foraging by Data Scientists to Find Past Analysis Choices. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. ACM, Glasgow Scotland Uk, 1–13. https://doi.org/10.1145/3290605.3300322

[31] Pranav Khadpe, Ranjay Krishna, Li Fei-Fei, Jeffrey T Hancock, and Michael S Bernstein. 2020. Conceptual metaphors impact perceptions of human-ai collaboration. *Proceedings of the ACM on Human-Computer Interaction* 4, CSCW2 (2020), 1–26. Publisher: ACM New York, NY, USA.

[32] Michelle S. Lam, Janice Teoh, James A. Landay, Jeffrey Heer, and Michael S. Bernstein. 2024. Concept Induction: Analyzing Unstructured Text with High-Level Concepts Using LLooM. In *Proceedings of the CHI Conference on Human Factors in Computing Systems (CHI '24)*. Association for

Computing Machinery, New York, NY, USA, 1–28. https://doi.org/10.1145/3613904.3642830
[33] James A. Landay and Brad A. Myers. 1995. Interactive sketching for the early stages of user interface design. In *Proceedings of the SIGCHI conference on Human factors in computing systems - CHI '95*. ACM Press, Denver, Colorado, United States, 43–50. https://doi.org/10.1145/223904.223910
[34] Tomas Lawton, Kazjon Grace, and Francisco J Ibarrola. 2023. When is a Tool a Tool? User Perceptions of System Agency in Human–AI Co-Creative Drawing. In *Proceedings of the 2023 ACM Designing Interactive Systems Conference*. 1978–1996.
[35] Jingyi Li, Eric Rawn, Jacob Ritchie, Jasper Tran O'Leary, and Sean Follmer. 2023. Beyond the Artifact: Power as a Lens for Creativity Support Tools. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology (UIST '23)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3586183.3606831
[36] Toby Jia-Jun Li, Amos Azaria, and Brad A Myers. 2017. SUGILITE: creating multimodal smartphone automation by demonstration. In *Proceedings of the 2017 CHI conference on human factors in computing systems*. 6038–6049.
[37] Geoffrey Litt. 2024. Patchwork lab notebook: Version control for everything. https://www.inkandswitch.com/patchwork/notebook/
[38] Michael Xieyang Liu, Advait Sarkar, Carina Negreanu, Benjamin Zorn, Jack Williams, Neil Toronto, and Andrew D Gordon. 2023. "What it wants me to say": Bridging the abstraction gap between end-user programmers and code-generating large language models. 1–31.
[39] Vivian Liu and Lydia B Chilton. 2022. Design Guidelines for Prompt Engineering Text-to-Image Generative Models. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems (CHI '22)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3491102.3501825
[40] Justin Lubin, Nick Collins, Cyrus Omar, and Ravi Chugh. 2019. Program Synthesis with Live Bidirectional Evaluation. *arXiv preprint arXiv:1911.00583* (2019).
[41] Aran Lunzer and Kasper Hornbæk. 2008. Subjunctive Interfaces: Extending Applications to Support Parallel Setup, Viewing and Control of Alternative Scenarios. *ACM Trans. Comput.-Hum. Interact.* 14, 4 (Jan. 2008). https://doi.org/10.1145/1314683.1314685 Place: New York, NY, USA Publisher: Association for Computing Machinery.
[42] J. Marks, B. Andalman, P. A. Beardsley, W. Freeman, S. Gibson, J. Hodgins, T. Kang, B. Mirtich, H. Pfister, W. Ruml, K. Ryall, J. Seims, and S. Shieber. 1997. Design Galleries: A General Approach to Setting Parameters for Computer Graphics and Animation. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '97)*. ACM Press/Addison-Wesley Publishing Co., USA, 389–400. https://doi.org/10.1145/258734.258887
[43] Andrew M. McNutt, Chenglong Wang, Robert A. DeLine, and Steven M. Drucker. 2023. On the Design of AI-powered Code Assistants for Notebooks. http://arxiv.org/abs/2301.11178 arXiv:2301.11178 [cs].
[44] Bill Moggridge. 2006. *Designing Interactions*. The MIT Press.
[45] Jakob Nielsen. [n. d.]. 10 Usability Heuristics for User Interface Design. https://www.nngroup.com/articles/ten-usability-heuristics/
[46] James Prather, Brent N Reeves, Paul Denny, Brett A Becker, Juho Leinonen, Andrew Luxton-Reilly, Garrett Powell, James Finnie-Ansley, and Eddie Antonio Santos. 2023. "It's Weird That it Knows What I Want": Usability and Interactions with Copilot for Novice Programmers. *ACM Transactions on Computer-Human Interaction* 31, 1 (2023), 1–31. Publisher: ACM New York, NY.
[47] Eric Rawn, Jingyi Li, Eric Paulos, and Sarah Chasins. 2023. Understanding Version Control as Material Interaction with Quickpose. In *Proceedings of the SIGCHI conference on human factors in computing systems*.
[48] Jeff Sauro and James R Lewis. 2016. *Quantifying the user experience: Practical statistics for user research*. Morgan Kaufmann.
[49] Donald A Schon. 1984. *The Reflective Practitioner: How Professionals Think In Action*. Vol. 5126. Basic Books.
[50] Shreya Shankar, J. D. Zamfirescu-Pereira, Björn Hartmann, Aditya G. Parameswaran, and Ian Arawjo. 2024. Who Validates the Validators? Aligning LLM-Assisted Evaluation of LLM Outputs with Human Preferences. https://doi.org/10.48550/arXiv.2404.12272 arXiv:2404.12272 [cs].
[51] Chenglei Si, Yanzhe Zhang, Zhengyuan Yang, Ruibo Liu, and Diyi Yang. 2024. Design2Code: How Far Are We From Automating Front-End Engineering? http://arxiv.org/abs/2403.03163 arXiv:2403.03163 [cs].
[52] Hari Subramonyam, Roy Pea, Christopher Pondoc, Maneesh Agrawala, and Colleen Seifert. 2024. Bridging the Gulf of Envisioning: Cognitive Challenges in Prompt Based Interactions with LLMs. 1–19.
[53] Sangho Suh, Bryan Min, Srishti Palani, and Haijun Xia. 2023. Sensecape: Enabling Multilevel Exploration and Sensemaking with Large Language Models. *arXiv preprint arXiv:2305.11483* (2023).
[54] Theodore R. Sumers, Shunyu Yao, Karthik Narasimhan, and Thomas L. Griffiths. 2024. Cognitive Architectures for Language Agents. https://doi.org/10.48550/arXiv.2309.02427 arXiv:2309.02427 [cs].
[55] M. Swan. 1927. *How You Can Write Plays: A Practical Guide-book*. S. French. https://books.google.com/books?id=UyYrAAAAIAAJ
[56] Michael Terry, Chinmay Kulkarni, Martin Wattenberg, Lucas Dixon, and Meredith Ringel Morris. 2023. AI Alignment in the Design of Interactive AI: Specification Alignment, Process Alignment, and Evaluation Support. https://doi.org/10.48550/arXiv.2311.00710 arXiv:2311.00710 [cs].
[57] Michael Terry, Elizabeth D. Mynatt, Kumiyo Nakakoji, and Yasuhiro Yamamoto. 2004. Variation in element and action: supporting simultaneous development of alternative solutions. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, Vienna Austria, 711–718. https://doi.org/10.1145/985692.985782
[58] Priyan Vaithilingam, Ian Arawjo, and Elena L. Glassman. 2024. Imagining a Future of Designing with AI: Dynamic Grounding, Constructive Negotiation, and Sustainable Motivation. In *Designing Interactive Systems Conference*. ACM, IT University of Copenhagen Denmark, 289–300. https://doi.org/10.1145/3643834.3661525

[59] Tongshuang Wu, Michael Terry, and Carrie J Cai. 2022. AI Chains: Transparent and Controllable Human-AI Interaction by Chaining Large Language Model Prompts. In *Proceedings of the 2022 CHI conference on human factors in computing systems*.

[60] J.D. Zamfirescu-Pereira, Heather Wei, Amy Xiao, Kitty Gu, Grace Jung, Matthew G Lee, Bjoern Hartmann, and Qian Yang. 2023. Herding AI cats: Lessons from designing a chatbot by prompting GPT-3. In *Proceedings of the 2023 ACM Designing Interactive Systems Conference*. 2206–2220.

[61] J.D. Zamfirescu-Pereira, Richmond Y Wong, Bjoern Hartmann, and Qian Yang. 2023. Why Johnny can't prompt: how non-AI experts try (and fail) to design LLM prompts. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. 1–21.

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009