Squire: Interactive UI Authoring via Slot QUery Intermediate REpresentations

Alan Leung
Apple
Seattle, WA, USA
alleu@apple.com

Ruijia Cheng
Apple
Seattle, WA, USA
rcheng23@apple.com

Jason Wu
Apple
Seattle, WA, USA
jason_wu8@apple.com

Jeffrey Nichols
Apple
Seattle, WA, USA
jwnichols@apple.com

Titus Barik
Apple
Seattle, WA, USA
tbarik@apple.com

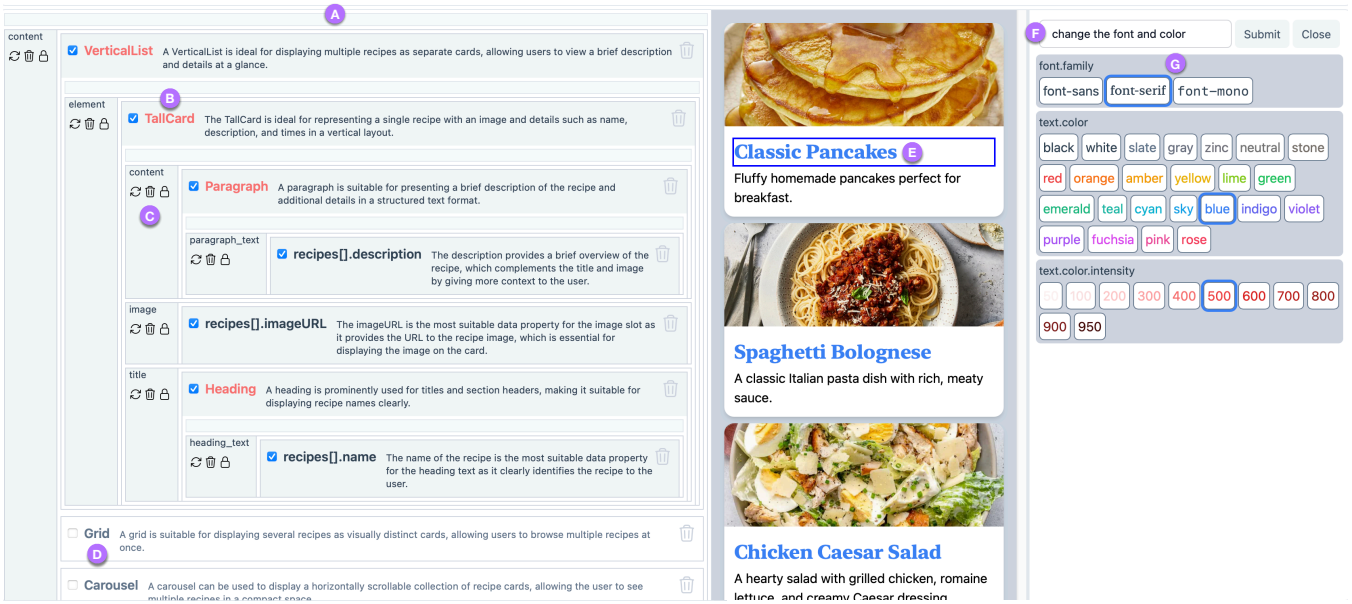


Figure 1: The Squire system for iterative UI prototyping. The Squire system incrementally generates a tree-based intermediate representation (A) that concisely encodes high level architecture as a component hierarchy. Components (B) holds slots (C) that Squire instantiates with the help of an LLM that incrementally updates its context as the hierarchy becomes increasingly detailed. To support design exploration, when Squire determines a component may have multiple compatible alternatives (e.g. list or grid), Squire offers special choice nodes (D) which, when selected, modify the UI to reflect the new alternative. To make fine-grained aesthetic changes (e.g. typography, color), the user selects a target (E) in the live preview and enters a free-form command (F). Squire interprets the command, mutates the targeted component accordingly, then offers ephemeral controls to apply further semantically-related changes (G) to help the developer evaluate design alternatives.

Abstract

Frontend developers create UI prototypes to evaluate alternatives, which is a time-consuming process of repeated iteration and refinement. Generative AI code assistants enable rapid prototyping simply by prompting through a chat interface rather than writing code. However, while this interaction gives developers flexibility since they can write any prompt they wish, it makes it challenging to control what is generated. First, natural language on its own can be ambiguous, making it difficult for developers to precisely communicate their intentions. Second, the model may respond unpredictably, requiring the developer to re-prompt through trial-and-error to repair any undesired changes. To address these weaknesses, we introduce Squire, a system designed for guided prototype exploration and refinement. In Squire, the developer incrementally builds a UI component tree by pointing and clicking on different

UIST '25, Busan, Republic of Korea

ACM ISBN 979-8-4007-2037-6/2025/09
https://doi.org/10.1145/3746059.3747672

alternatives suggested by the system. Additional affordances let the developer refine the appearance of the targeted UI. All interactions are explicitly scoped, with guarantees on what portions of the UI will and will not be mutated. The system is supported by a novel intermediate representation called SquireIR with language support for controlled exploration and refinement. Through a user study where 11 frontend developers used Squire to implement mobile web prototypes, we find that developers effectively explore and iterate on different UI alternatives with high levels of perceived control. Developers additionally scored Squire positively for usability and general satisfaction. Our findings suggest the strong potential for code generation to be controlled in rapid UI prototyping tools by combining chat with explicitly scoped affordances.

CCS Concepts

• **Human-centered computing → Systems and tools for interaction design**; **User interface programming**.

Keywords

UI prototyping, intermediate representations, artificial intelligence

ACM Reference Format:
Alan Leung, Ruijia Cheng, Jason Wu, Jeffrey Nichols, and Titus Barik. 2025. Squire: Interactive UI Authoring via Slot QUery Intermediate REpresentations. In *The 38th Annual ACM Symposium on User Interface Software and Technology (UIST '25), September 28-October 1, 2025, Busan, Republic of Korea.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3746059.3747672

1 Introduction

Building user interface prototypes is a standard method to evaluate design alternatives and elicit feedback. Traditionally, this kind of prototyping involves a time-consuming process requiring multiple iterations of exploration, implementation, and refinement [3, 16] where constructing high-fidelity prototypes is generally the most time-consuming part of the process [23, 29, 30]. Recently, generative AI coding assistants such as ChatGPT [34] and v0 [48] have emerged promising to enable rapid prototyping with high fidelity using only a chat interface. These tools allow designers and developers to explore and refine prototypes simply by prompting language models to perform the heavy lifting of translating natural language requests to executable code, removing much of the manual effort previously required to code prototypes.

Still, while prompt-based interactions may give users substantial latitude in the way they frame their requests, this freedom is a double-edged sword. First, interaction at the level of prompts is a challenge due to the inherent ambiguity of natural language, with several studies echoing the challenge that users often struggle to formulate prompts to convey their intent effectively [2, 46, 54]. Second, the output of generative models can be unpredictable [13, 24, 53]. Actual outputs sometimes contain changes not intended by the user, which leads to slow trial-and-error loops in an attempt to compare different results and converge on a desired outcome [25]. This second problem is exacerbated by the fact that developers end up spending a substantial amount of time reviewing and comparing iterations rather than generating the code itself [46].

We posit that these limitations stem from the fact that free-form prompt interactions lack two desirable properties:

(1) It should be possible to precisely scope change requests to specific aspects of the system being modified. This gives the user confidence that no changes beyond the specified scope have occurred. By contrast, current chat-based programming assistants generally permit the model to modify generated code arbitrarily, with no guarantees that any constraints specified in the prompt will be obeyed.
(2) Alternative outputs should be fast and easy to compare, ideally with minimal friction when navigating between them. By contrast, the predominant chat-based paradigm relies on a linear chat history without branching navigation, so picking between alternatives can be a tedious copy-paste exercise.

In this paper, we present Squire, a high-fidelity UI prototyping tool that imbues its interactions with the above two properties: it gives users explicit controls to scope modification and quickly explore alternatives. In Squire, users start a project by providing a prompt that describes their goals for the UI, along with sample data containing information for Squire to use as a reference. Users then construct UI as a tree of components in top-down fashion by prompting Squire to fill holes representing missing yet expected functionality. In response to this kind of request, Squire generates a list of appropriate alternatives, each scoped specifically to the targeted hole in the unfinished UI. Clicking on each alternative immediately updates a live rendered preview as well as underlying code, making it easy to visualize the differences. The user can also pose targeted requests to modify the appearance of specific areas of the UI, with the guarantee that no code outside the intended scope will be mutated. In response to this kind of request, Squire generates ephemeral controls [6] that allow the user to apply semantically-related changes quickly and without re-prompting. In all cases, the LLM acts as a companion, presenting reasonable choices for the user to evaluate, but leaving the user with agency to accept or reject its suggestions.

The features of Squire are underpinned by SquireIR, a novel domain-specific intermediate representation (IR) for encoding the space of UI alternatives generated by Squire. This representation enables the above interactions through special operators that represent holes needing instantiation (null operators) and explorable alternatives (choice operators), all while remaining automatically convertible to executable code.

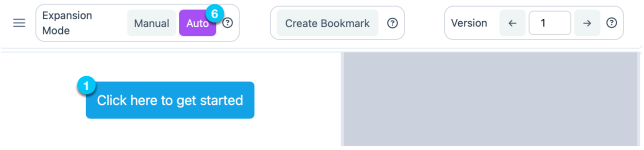
This paper makes the following contributions:

- We develop SquireIR, a domain-specific IR designed for scoped exploration and refinement of high-fidelity prototypes.
- We present Squire, a graphical development environment built atop SquireIR for exploration and iterative refinement, which we instantiate for the domain of mobile web application screens.
- Through data collected from a user study of 11 frontend developers, we find that (1) Squire's interactions encouraged participants to explore frequently, rather than simply use Squire as a code accelerator, (2) participants felt encouraged to take risks when making changes, knowing that the consequences of making atypical decisions could always be undone without friction, (3) participants indicated confidence that Squire matched their intent when making changes, and

(4) participants were generally pleased with the quality of code and visuals generated by the system.

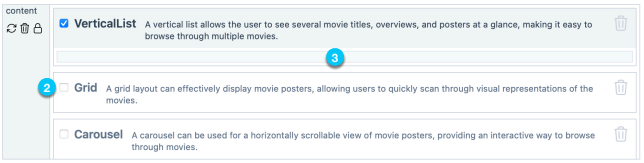
2 Example Squire Usage

Mina is a frontend developer who builds mobile web apps. She uses Squire to prototype an app screen for searching for movies. When Mina first loads Squire, she is greeted with the following interface, which shows an empty canvas with a button for getting started 1.



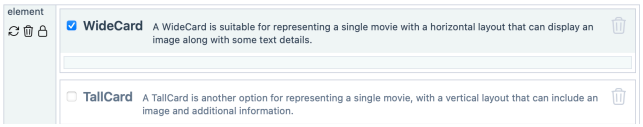
Getting Started: To begin, Squire asks Mina to provide (1) a system prompt describing her overall goal for the screen, and (2) sample data in JSON format that Squire will use as reference data. She enters the following system prompt: "A screen for viewing movies. The screen should allow the user to see several different movies in a scrollable view." Since she intends for her project to use an existing movie database, she extracts some of its contents and provides them as sample data.

Constructing the Component Tree: With preliminaries complete, Mina presses the button 1, and Squire begins processing its first request. After a few seconds, Squire updates the canvas to show a list of choices 2. These choices are proposals for components for the top level of the screen, along with design rationale for each choice. Mina reviews the choices and decides that a `VerticalList` would be most suitable since she agrees with the rationale that it would "allow the user to see several movie titles, overviews, and poster images at a glance."



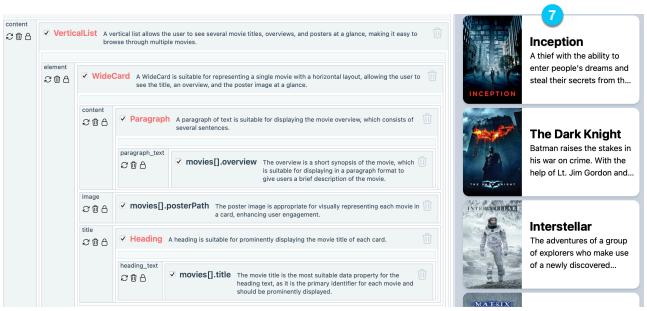
Mina is ready to move on to the next step, so she clicks 3 to open the *slot query dialog* 4. This is Squire's mechanism for asking which child of `VerticalList` Mina would like to start constructing next. Mina selects *element* and augments its textual description to provide more detail about her intended purpose for *element*: "Each element should represent a single movie." 5

As before, Squire processes the query for a few seconds and presents Mina with a new set of choices. This time, the choices represent possible components that could serve as list elements, where each element depicts a single movie. Mina decides she would like to use a `WideCard` component that uses a "horizontal layout that can display an image along with some text details."

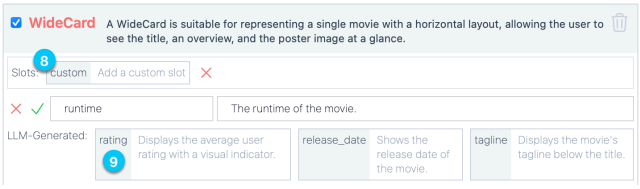


Up until this point, Mina has been selecting components one-by-one, which offers precise control but also slows her down in a situation where she would rather rapidly generate an initial design to refine more carefully afterwards. To do this, Mina switches to Auto Expansion Mode 6 (refer back to the first figure in this section), which tells Squire to make its own independent choices without waiting for feedback after each step. Mina now opens the slot query dialog for `WideCard` and submits a new query for the *title* slot. As shown in the following figure, Squire generates a `Heading` component whose text is the title of a movie, first by choosing the `Heading` component, and second by choosing `movies[].title` to be its *heading_text* slot (`movies[].title` is SquireIR notation for a reference to sample data, which in this specific case references the `title` element of each `movie`).

Mina proceeds to use the same procedure to populate the *image* and *content* slots of the `WideCard` component. At the conclusion of this process, Squire displays the following screen showing the completed component tree as well as a live preview of the rendered design. For each of the *title*, *image*, and *content* slots, Mina only had to make one request–Squire automatically generated the subtree by chaining together the appropriate decisions. Throughout this process, the Preview Pane 7 also continuously updated.



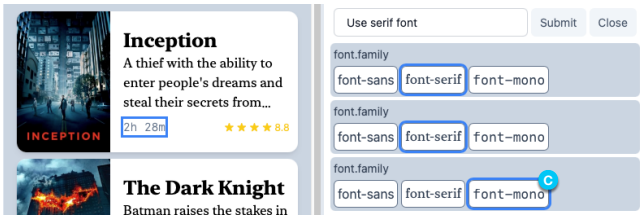
Adding Custom Slots: Mina is satisfied with the initial design, but she notices that each card is missing some important details, such as the movie duration. She would like to add this information to the `WideCard` component, but she sees that there is no existing slot for *runtime*. This is because `WideCard` starts life with a predefined template that includes slots for *title*, *image*, *content*, but no others. While this provides an initial starting point, Mina will need to customize it to meet her unique needs. To do so, Mina selects the *custom* option in `WideCard`'s slot query dialog 8 and submits a query for a new slot named *runtime* with the description "The runtime of the movie."



Squire responds by modifying `WideCard`'s template to add this new slot while keeping consistency with the preceding version of the template, then instantiates it with a caption displaying the movie duration. B

Beneath the slot query dialog, Mina sees that Squire has also auto-generated some suggestions for other possible custom slots. On reviewing the suggestions, Mina realizes that a movie rating would also be a good thing to add. She clicks the *rating* suggestion 9, and Squire modifies the template accordingly to also include a movie rating with star icons A.

Refining Detailed Aesthetics: Mina now decides to focus on detailed aesthetics: she would like to change the typography on each card to use a serif font. To do this, she double clicks a card in the Preview Pane, which opens the Refinement Pane to the right. She enters the command "Use serif font" and submits it. Squire responds by modifying the fonts used, and additionally presents a set of ephemeral controls for tweaking the fonts further: one each for the title, overview, and movie duration. By making different selections in these controls C, Mina is able to apply changes to the targeted components in real time to quickly visualize the differences. Without these controls, Mina would have needed to submit separate requests for each change, increasing the latency of each refinement.



Exploring Further: Satisfied with her first prototype, Mina now decides she would like to investigate a different set of design decisions altogether: what would the screen look like if she chose to use a grid instead of a list to display movies? To do this, she simply selects the `Grid` choice that is sibling to the `VerticalList` component she originally selected. Mina can now use all the same techniques she has already used to quickly produce a new prototype that employs a grid of movies instead. In fact, she can revisit choices throughout the component tree, not just at the top-level—each set of choices corresponds to a different set of interchangeable design decisions.

3 Related Work

We review literature in three related areas: model-based UI systems, generative UI models, and UI design support tools. These represent the three areas brought together in Squire: intermediate representations for UI, generation of UI, and exploration of UI.

Model-Based UI Intermediate Representations. Model-based UI (MBUI) approaches [37] seek to automate the creation of user interfaces by translating from high-level specifications of UI behavior and appearance to concrete user interfaces. Early work such as UIDE [22] and Jade [55] fully automate this process through translation to target implementations, with some later systems such as SUPPLE [15] employing optimization techniques at runtime to customize UI for target contexts. These approaches generally rely on manual authoring of UI specifications, and because much of this work predates contemporary breakthroughs in generative ML models, typically produce UI using deterministic, rule-based techniques.

A central aspect of MBUI systems is that they employ intermediate representations (models) to abstract different aspects of the user interface being developed. Although a variety of MBUI intermediate representations have been proposed for different purposes [4, 36, 47], we focus here on presentation models [27] since they are most related to SquireIR. Early presentation models focused on encoding specific types of user interfaces, such as menus and dialog box layouts [33, 55], with later iterations supporting more complex interfaces including components such as visualizations [27]. Although different systems employed different conventions, a typical feature of presentation models is treatment of UI as a collection of decision rules and constraints defining layout and component compatibility, meant to be refined by the user and translated by the system into a running application. The Humanoid [43] system is particularly relevant in this context as its goal is also to enable rapid UI prototyping. In Humanoid, the user adds increasingly detailed constraints to an initial template specification, with the system able to instantiate UI incrementally from incomplete specifications by assuming reasonable defaults when presentation rules are underspecified. SquireIR is inspired by the notion of incomplete UI specification as a substrate for incremental refinement, but SquireIR takes a conceptually different approach that lifts exploration of alternatives (in addition to refinement) to a first-class consideration—the IR encodes UI alternatives being explored in parallel (via choice nodes), rather than a single instance being linearly refined, which enables efficient navigation between the alternatives.

Generative UI Models. We describe two related areas aiming to specialize generative large language models [17, 35] toward UI generation [39, 44, 50] and evaluation [11, 12].

In the first line of work, researchers have proposed fine-tuning techniques intended to imbue language models with improved performance in UI generation tasks. UICoder [50] is a LLM fine-tuned for iOS UI generation from synthetically generated programs filtered for quality using a combination of code analysis and vision-language models. Other work has focused on more specific aspects of UI generation, such as PosterLlama [39] and LayoutNuwa [44], which both employ specialized fine-tuning procedures over images and HTML to improve layout generation quality. This line of work is complementary to our goals with Squire, as advances in model

capabilities accrue toward improved relevance and aesthetic quality achievable by Squire.

A second line of work has focused on applications of LLMs for UI evaluation to aid design activities. Duan et al. [12] developed a Figma plugin employing GPT-4 to generate heuristic evaluations of UI mockups. Subsequent work produced UICrit [11], a dataset of expert mobile design critiques for the purpose of supporting automated UI quality evaluation. UIClip [49] is a CLIP-based model fine-tuned for quality and relevance assessment given pairs of screenshots and descriptions. While not employed by Squire, we see these efforts as enablers for future opportunity to incorporate automated feedback, either for feedback to the user during prototyping, or as part of an underlying generation pipeline itself to improve generated outputs.

UI Design Support Tools. UI design is an iterative process that involves exploration of alternative versions and refinement of existing designs [7]. Exploring different design directions, especially at early stages, helps avoid the pitfall of fixation [21] and leads to better design outcomes [10], while refinement allows designers to incrementally work towards design goals [9].

Towards those aims, a number of research systems have been developed to support exploration and refinement in UI design. The Scout system [42] helps designers rapidly explore alternative UI layouts by framing layout generation as constraint satisfaction over constraints derived from encoded design principles and designer feedback. The d.note system [18] treats iteration as revision: the user expresses modifications to UI as ink annotations and sketches that the system recognizes and manifests as changes to a UI being live-prototyped. Misty [26] explores UI generation through conceptual blending of existing designs with design inspirations in the form of screenshots. Squire is a synthesis of insights from these different lines of work to combine rapid exploration, targeted refinement/revision, LLM-aided design inspiration, and live-prototyping in a single system. However, unlike Squire, Scout and d.note do not generate code, and Misty uses images rather than text as its input modality.

Looking beyond research systems, we see that commercial startups have also produced LLM-based code assistants meant for UI design, where Claude Artifacts [1] and Vercel v0 [48] are two prominent examples of tools that allow users to provide prompts to generate and live-render UI code. We contrast with these efforts by noting that these tools employ linear chat as the sole mechanism for exploration and refinement, which has the limitations described in §1 that Squire aims to address.

4 The Squire System

In this section, we outline the design motivations of the system (§4.1), describe the SquireIR representation that defines the data structure for encoding user interfaces and their variations (§4.2), then describe the design and architecture of Squire, which employs SquireIR as its underlying data structure (§4.3).

4.1 Design Motivations

The design motivations of Squire are supported by literature in the area of AI developer support tools and a qualitative survey meant to elicit pain points of frontend developers (N=21) at a large technology company when using current generative AI developer assistants.

DM1: Make the scope of changes explicit. A challenge when interacting with AI assistants is the open-ended nature of communicating intent through natural language. Prior work [41, 54] has shown that users find difficulty crafting prompts that adequately convey intent, resulting in frustration when models fail to meet expectation. One contributing factor is the existence of *implicit intent*—unspoken details that users elide because writing such detailed prompts is onerous [5]. Survey respondents noted that AI assistants "need hand-holding" and "careful prompting" and even then "don't often go well". Another respondent noted they would "like to see tools that are more opinionated [and] more directly enable an experienced programmer to drive the LLM," pointing towards a desire for explicit control affordances in addition to prompts when interacting with code assistants. In Squire, developers interact with specialized controls bound to specific parts of the UI (components) or styles (ephemeral controls), and the model is constrained accordingly.

DM2: Encourage iteration and exploration. A weakness of existing chat-based AI assistants is the slow iteration loop caused by unreliable output and inconsistency between prompt responses [5, 51, 54]. The resulting friction limits developers' ability to explore alternatives efficiently, with prompt iteration often devolving to trial and error [54]. Survey responses noted that "prompting an LLM is easy, but inefficient," and that "it can definitely slow you down if you rely solely on GenAI to produce code for you" because they end up "spending more time trying to piece together bits of generated code" with copy-paste. Despite this, respondents did find that generative AI assistants helped them ideate different approaches—indeed, several respondents noted the positive aspects of generative AI in providing "new perspective[s] to solve a problem," helping "ideate on how to solve complex problems," or serving as a "thought assistant" to develop different high level designs. Squire treats exploration and iteration as a first-class concern, providing specialized interactions for exploring different UI choices, beyond just prompt revision and tuning, and quickly navigating between different versions explored.

DM3: Facilitate review of generated outputs. Several respondents noted the challenge and tedium of reviewing model outputs for correctness. For example, one respondent noted that while generative models may "generate code quickly… often that time gained is lost again" in reviewing and debugging the output. Another respondent indicated they spent more time "understanding generated code" compared to writing it themselves. This echoes findings from prior work that found users have difficulty reviewing low-level output code for correctness [46, 52], with one developer noting that they "did not understand several parts of the function generated," which caused them "to get rid of the whole function… and start over." Squire provides users with a high-level visual representation to convey major UI components and their relationships, without requiring the developer to review low-level target code on every iteration. This design motivation is complementary to DM2, as an important aspect of exploration is the ability to quickly review variants for comparison.

Template
```
<template id="x-button-template">
 <button><slot name="label"></slot></button>
</template>
```
Description "A button with a label."

Slots (*label*, "The button label", 1)

Template
```
<template id="x-verticallist-template">
 <div class="flex flex-col items-stretch gap-2">
  <slot class="w-full" name="element"></slot>
 </div>
</template>
```
Description "Use a VerticalList to present several homogeneous items in a vertical arrangement."

Slots (*element*, "An element in the list.
Each element is rendered identically.", +)

Figure 2: Component definitions for `Button` (top) and `VerticalList` (bottom).

4.2 SquireIR: Language Definition

In this section, we define the intermediate representation used within Squire as an abstraction over UIs. By *abstraction*, we mean that SquireIR is capable of representing multiple alternative user interfaces within a single program (covered in detail in §4.2.2), along with optional holes indicating portions of the UI that have yet to be generated. To do this, SquireIR hierarchically composes components with *null operators* that represent children that have not yet been explored and *choice operators* that represent decision points between alternative but compatible components. Each component itself consists of a template containing placeholders augmented with contextual metadata.

4.2.1 Component Definition. Each component is an HTML template with zero or more named placeholders for children, called *slots*. A component definition additionally contains a natural language description of the component and natural language descriptions for each slot. More precisely, a component is a tuple $C = (T, N_c, D_c, \sigma)$ where T is an HTML template, N_c is the component name, D_c is a natural language description, and σ is a set of slot definitions. Each slot definition $S = (N_s, D_s, a)$ consists of a slot name N_s, a slot description D_s, and arity $a \in \{1, +\}$ that indicates whether the slot may be instantiated only once (1) or multiple times (+). In Squire, we call the set of all component definitions a *component library*.

Figure 2 shows example component definitions for two components, a `Button` and `VerticalList`, respectively. `Button` is a simple button with a label, while `VerticalList` defines a layout in which child elements are stacked vertically with a small gap between each. Notice that `VerticalList`'s `element` slot has arity +, which specifies that the `element` slot can be instantiated multiple times in a single `VerticalList` instance since the list can contain multiple elements, not just one.

To decide on a set of components with broad coverage, we sampled 60 mobile app screens from UXArchive [45], an online repository of mobile app screen captures, then classified them according to the components they used. We then implemented HTML templates for each component class found. We stopped at 60 screens because additional samples led to diminishing returns and were no longer contributing to the set of new components we classified. In total, the Squire component library contains 35 components (see Table 1 in Appendix A for the list of the components).

Note that SquireIR itself is parametric in its component library and supports extensibility for different design systems or UI component conventions—that is, its component library can be substituted with another without modification to other parts of the system, as long as the components are defined in the format described. Additionally, templates only serve as *starting points* and can be customized further via *custom slot queries*, which we describe in more detail in §4.3.3.

4.2.2 Component Instantiation. A SquireIR program nests component *instances*, where each instance is a component definition paired with a substitution of its slots for children. For example, consider a screen containing a single button labeled "Click me". In SquireIR, we denote this by the program `Screen`[*content* ↦ `Button`[*label* ↦ *"Click me"*]], where `Screen` is the top level of the UI that fills the viewport. In general, a component instance may contain more than one slot, in which case multiple slot mappings may appear within the substitution: `X`[a ↦ `A`, b ↦ `B`, …].

Null operators ∅. Consider the situation when a SquireIR program has only been partially constructed. For example, consider a screen containing a labeled button `Screen`[*content* ↦ `Button`[*label* ↦ ∅]], where we intend to define a label for the button, but it is yet to be defined. The null operator ∅ serves as a promise that Squire will expand the corresponding slot into one or more components (slot expansion is covered in detail in §4.3). For brevity, we equivalently write `Component`[∅] to represent that all slots map to ∅ (e.g. `Button`[∅] is equivalent to `Button`[*label* ↦ ∅]).

Choice operators ⊕. Up until now, what has been described about SquireIR has been a notation for a single UI with holes. However, an important aspect of SquireIR is that it can describe multiple alternatives simultaneously in a single program. This is achieved through *choice operators* that denote there are multiple compatible components that inhabit the same position in the UI hierarchy. For example, `Screen`[*content* ↦ ⊕(`VerticalList`[∅] , `Grid`[∅])] denotes a screen that may either use a vertical list or a grid as its main layout. Choice operators may nest within each other as well. For example, `Screen` [*content* ↦ ⊕(`VerticalList` [*element* ↦ ⊕(`TallCard` [∅] , `WideCard` [∅])] , `Grid` [∅])] abstracts over 3 UIs:

(1) `Screen`[*content* ↦ `VerticalList`[*element* ↦ `TallCard`[∅]]]
(2) `Screen`[*content* ↦ `VerticalList`[*element* ↦ `WideCard`[∅]]]
(3) `Screen`[*content* ↦ `Grid`[∅]]

From top to bottom, these describe (1) a screen with a vertical list of tall cards (a tall card is a card whose contents are arranged vertically), (2) a screen with a vertical list of wide cards (a wide card is a card whose contents are arranged horizontally), and (3) a screen with a grid layout.

At most one operand of a choice operator may be *selected*, which we denote by underlining the selection. For example, `Screen`[*content* ↦ ⊕(<u>`VerticalList`</u>[*element* ↦ ⊕(<u>`TallCard`</u>[∅], `WideCard`[∅])], `Grid`[∅])] translates to `Screen` [*content* ↦ `VerticalList` [*element* ↦ `TallCard`[∅]]] by substituting choice operators for their selected operands. In general, any combination of selections leads to a distinct instantiation.

Datum instances. To represent data access, SquireIR programs include *datum instances* in addition to component instances. Datum instances act as queries into an external store of sample data. For

example, `Image` [*path* ↦ `movies[].posterPath`] represents an image whose asset path is given by `movies[].posterPath`. The translation of this asset path to a concrete request is detailed in §4.3.5.

Instance identifiers. SQUIREIR associates each component instance with a unique identifier to disambiguate it from other instances of the same component definition. Consider the following example annotated with identifiers: $\texttt{Screen}_1$ [*content* ↦ ⊕($\texttt{VerticalList}_2$ [*element* ↦ ⊕($\texttt{TallCard}_3$ [∅] , $\texttt{WideCard}_4$ [∅])] , $\texttt{Grid}_5$ [*element* ↦ $\texttt{WideCard}_6$])]. This representation distinguishes $\texttt{WideCard}_4$ from $\texttt{WideCard}_6$ to denote that each may have individual styling and contents different from the other, despite both being instances of `WideCard`. SQUIREIR always associates instances with identifiers, but for clarity in this paper we may exclude them when clear from context.

4.3 SQUIRE: System Design

We now describe the design and implementation of SQUIRE. Throughout this section, we refer to the SQUIRE subsystems depicted in Figure 3, which illustrates the system components and their relationships.

4.3.1 Project preliminaries. To initialize a project, the developer provides SQUIRE with a project description that contextualizes their intent for the UI being developed. This project description contains (1) a natural language prompt indicating the purpose of the screen to be developed, and (2) JSON-formatted sample data that SQUIRE references to generate data requests in the form of datum instances. For example, referring back to the motivating example from §2, Mina provided the following prompt: "A screen for viewing movies. The screen should allow the user to see several different movies in a scrollable view." Mina also provided the sample data shown in Figure 4 consisting of a JSON sample together with a schema that associates each field in the sample data with a description and type. Together, these form the context used as ingredients of the prompts used by SQUIRE.

4.3.2 Slot expansion. The structure of a SQUIREIR program directly corresponds to its representation in the Visual Editor, which depicts the component tree in the left half of the SQUIRE interface. Recall step ❷ from §2, in which Mina requested different component choices for the top level of the screen. The equivalent SQUIREIR program is $\texttt{Screen}_1$ [*content* ↦ ⊕($\underline{\texttt{VerticalList}_2}$,$\texttt{Grid}_3$,$\texttt{Carousel}_4$)]. That is, choice nodes are rendered as labeled checkboxes in the Visual Editor, where the selected component's checkbox is enabled. In subsequent step ❸, Mina submitted a slot query for the *element* slot of $\texttt{VerticalList}_2$. The Slot Query Engine begins processing the query by inserting a null operator substitution $\underline{\texttt{VerticalList}_2}$ [*element* ↦ ∅], which the Visual Editor displays visually as a progress spinner that indicates that SQUIRE is currently processing a request for more choices for *element* at the position of the null operator. The Slot Query Engine completes the slot query by rewriting the corresponding SQUIREIR fragment to $\underline{\texttt{VerticalList}_2}$ [*element* ↦ ⊕($\texttt{WideCard}_5$, $\texttt{TallCard}_6$)] in a process called *slot expansion*, which results in a choice between `WideCard` and `TallCard` components.

Expansion modes. In Manual Expansion Mode, SQUIRE only ever adds a single null operator to the underlying SQUIREIR program at a time, which means that the user must interactively select the slot for each slot expansion operation. However, it is perfectly valid for a SQUIREIR program to contain multiple null operators—this is the case for Auto Expansion Mode, in which SQUIRE inserts a null operator substitution for every uninstantiated slot in the subtree being expanded. In this case, SQUIRE expands null operators in breadth first search order, and in cases where a newly instantiated child component instance has slots of its own, SQUIRE recursively inserts more null operators for each, and so on. The net result of this progressive expansion is that SQUIRE generates an entire subtree of components based on a single interaction by the user, which is how Mina is able to generate a card structure with minimal interaction in step ❻.

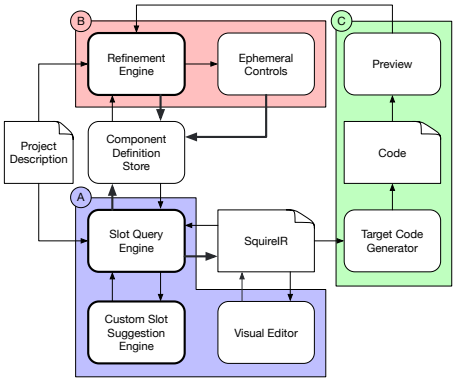


Figure 3: SQUIRE system. The Slot Query Engine modifies components and SQUIREIR in response to slot queries, which can either be manually authored or generated by the Custom Slot Suggestion Engine. The Refinement Engine and Ephemeral Controls modify components in response to refinement queries. Engines making LLM requests are outlined in bold. Bold arrows mean the source modifies the target. The Visual Editor, Refinement Pane, and Preview Pane subsystems are groups A, B, and C, respectively.

Sample
```
{ "movies": [
  { "title": "Inception",
    "posterPath": "static/inception.jpg",
    "overview": "A thief with the ability to enter...",
    "voteAverage": 8.8, ...
  }, ... ] }
```

Schema
```
{ "name": "Movie",
  "description": "A movie",
  "elements": [
    { "name": "title",
      "description": "Movie title",
      "type": "String" },
    { "name": "posterPath",
      "description": "URL to poster image.",
      "type": "String" },
    { "name": "overview",
      "description": "Short synopsis of the movie",
      "type": "String" },
    { "name": "voteAverage",
      "description": "Average user rating",
      "type": "Double" }, ... ] }
```

Figure 4: Sample data together with a schema assigning descriptions and types to fields.

4.3.3 Customization. The system as described so far only permits composition of static templates. However, a system that only supported immutable templates would severely limit the freedom of developers to customize designs for specific use cases. Referring back to our motivating example with Mina, note that it was possible for Mina to create an initial UI with a list of cards with title and overview text, but she later wished to include additional details (e.g. movie rating) or change aesthetics (e.g. typography). This section describes the features that enable these customization capabilities: custom slots queries and the Refinement Pane.

Custom slot queries. Recall from §4.2.1 that every slot in a component definition has two parts: (1) the slot tag itself in the HTML template (e.g. `<slot name="content"></slot>`), and (2) the slot's metadata such as its name and description. The user may perform a *custom slot query* in which they ask Squire to modify a template to include an additional slot (Mina demonstrated this by submitting the custom slot query for a slot named *runtime* in step 8). In response to a custom slot query, the Slot Query Engine first prompts the model to add a `<slot>` tag to the targeted component's template. More specifically, the prompt includes (1) the HTML template of the targeted component and (2) the name and description of the slot to be added. Based on this context, the model infers an appropriate rewrite to the template incorporating the new slot. In a second step, the Slot Query Engine modifies the backing SquireIR program by mapping the new slot to a fresh null operator and performing a slot expansion to instantiate the slot just added with a choice of components. In Mina's case, Squire expands Mina's slot substitution from `WideCard` [*runtime* ↦ ∅, …] to `WideCard` [*runtime* ↦ `Caption` [*caption_text* ↦ `movies[].durationText`], …]. In other words, Squire decides that the movie runtime information should be represented as a caption showing the movie's `durationText` value.

Custom slot query suggestions. To help users ideate possible slots based on context, the Visual Editor auto-populates the slot query dialog with suggested custom slots. The Custom Slot Suggestion Engine achieves this by prompting the model with the component tree outline and sample data, which the model uses to suggest a list of custom slots appropriate for the instance on which the user has opened the slot query dialog.

Template refinement. Squire maintains a database of component definitions called the Component Definition Store that maps instances in the component tree to their underlying component definitions. In this store, each instance maps to its own independent definition by keying off its instance identifier. By doing this, even when multiple instances of a component occur in the component tree, each has its own definition, which allows the user to customize different instances independently with respect to where they lie within the overall user interface.

As the user interacts with the Refinement Pane, Squire mutates the targeted component's template in response to requests for modifications. This refinement process occurs in two phases.

In the first phase, the Refinement Engine asks the model to edit targeted templates according to the user's refinement query. Targets are known based on the coordinates in the Preview Pane where the user has double clicked. In this phase, the model cannot make modifications to any templates that are outside the targeted region of the UI since it does not have access to them.

```
You are a UX designer asked to develop the app structure for an iOS app. Here
is the description of the app: "A screen for viewing movies. The screen should
allow the user to see several different movies in a scrollable view."
---
Screen
└ content: Content of the screen
   └ VerticalList: A vertical list allows the user to see several movie
titles, overviews, and posters at a glance, making it easy to browse through
multiple movies.
      └ element: Each element should represent a single movie. <-- Please
focus here: what should be instantiated here?
---
You have at your disposal the following subcomponents. Which would make sense
to use for the slot named "element" that is highlighted above?
- VerticalList: Use a VerticalList to present several homogeneous items in a
vertical arrangement.
- Button: A button with a label.
- WideCard: Use a WideCard to represent a single item out of many. The contents
are arranged in a horizontal layout with an image on the side.

You have at your disposal the following data properties:
- movies[].title: Movie title. Type: String
- movies[].posterPath: URL to poster image. Type: String
- movies[].overview: Short synopsis of the movie. Type: String
---
Provide a list of 3 answers in the following format. "name" is the name of the
component or data property. "reason" is your rationale for making that choice.
"supporting_data" is a list of data properties that you plan to use as part of
that component.
```json
[{ "name": "Answer 1", "reason": "Explanation 1", "supporting_data": ["Data 1",
"Data 2"] }]
```
```

Figure 5: Example slot expansion prompt derived from step 5 of §2. The prompt has been edited for brevity and presentation. We delineate the preamble, input spec, multiple choice, and output spec via horizontal borders between the sections.

In the second phase, the Refinement Engine asks the model to classify the changes from the first phase into one or more categories drawn from a library of aesthetic changes. Based on this classification, Squire produces ephemeral controls, which are the interactive dialogs that appear in the Refinement Pane for the user to refine template styling via buttons. Squire supports a total of 18 dialog types derived from a common subset of the utility classes from the popular TailwindCSS library, which includes styling for typography, color, border, sizing, and padding. Each dialog is backed by an algorithmic subroutine that parses HTML and rewrites `class` attributes in the targeted template, which provides fast iteration that does not require round trips to the model.

4.3.4 Prompt generation. Squire uses 5 different prompt formats: (1) slot expansion, (2) slot suggestion, (3) custom slot queries, (4) template refinement, and (5) ephemeral controls generation. The prompts are modular and composed of four conceptual parts:

- The **preamble** provides general context for the nature of the request (all formats).
- The **input spec** depicts the UI being operated upon. This can either take the form of a component tree (formats 1-2), or HTML snippets (formats 3-5), depending on the purpose of the prompt.
- The **choice spec** provides the options available for the model to choose (formats 1,2,5).
- The **output spec** specifies the expected output format via few-shot prompting or explicit instructions (all prompts).

Figure 5 depicts an abbreviated slot expansion prompt derived from step 5 of §2. Notice that the preamble includes the project description as general context. The input spec contains a marker "`<- Please focus here:…`" next to the target slot so the model knows which slot to expand. The choice spec lists components and data available, retrieved from the component library and project description, respectively. And finally, the output spec asks the model

to return up to three slot expansion choices along with rationale. The Slot Query Engine parses these choices and inserts them into the SquireIR using a choice node.

For brevity, we omit examples of the other four prompt formats but describe them briefly here. Slot suggestion prompts share a similar structure to slot expansion prompts, but the output spec asks for suggested slot names and descriptions. For custom slot queries, the input spec consists of the HTML template for the targeted component, and the output spec asks the model to edit the template to incorporate the custom slot tag being added. For template refinement prompts, the input spec contains the HTML templates along the path from the root to the target, and the output spec asks the model to edit the targeted templates according to the refinement query. Finally, for ephemeral controls generation, the input spec contains a diff patch before and after executing the template refinement query, the choice spec contains a list of available control types, and the output spec asks the model to return any controls matching the changes from the diff.

4.3.5 SquireIR → target code. The Target Code Generator translates the SquireIR program to code containing HTML, CSS, and JavaScript that makes use of the Web Components API [32], a standard API supported by modern browsers for encapsulating and composing components through the use of custom tags. First, each component definition's template is copied to a `<template>` tag in the target code. Squire then generates a `<script>` tag containing calls to the Web Component API's `customElements.define()` function that register each template with a corresponding `HTMLElement` subclass, one for each component definition used by the SquireIR program being translated. Note that this JavaScript is not generated by the model, as it is mechanical boilerplate that simply registers the name of the component and its template with the browser runtime. Finally, the component tree is translated to nested custom element tags that mirror the syntactic structure of the component tree. Again, this translation does not require use of a generative model, as the SquireIR program already precisely defines the recursive structure of the UI which can be translated mechanically to nested custom elements, and HTML provides standard functionality for instantiating custom elements and their children [31]. Figure 6 depicts the nested custom element structure generated by this translation process, where template definitions occur before their nested instantiations. Note how multiple `x-widecard-14` instances have been generated, each corresponding to a different movie—Squire infers this repetition based on the + arity defined for the `VerticalList`'s *element* slot.

Datum instance translation. Recall from §4.2.2 that SquireIR uses datum instances to represent external data requests. Datum instances perform asynchronous GET requests that retrieve sample data contents from the Squire backend. For example, the SquireIR fragment `Paragraph` [*paragraph_text* ↦ `movies[].overview`] indicates that the paragraph text should be the movie overview. This becomes the following HTML fragment after translation (excerpted from Figure 6): `<x-paragraph-23 slot="content"><x-datum value="movies[0].overview" slot="paragraph_text"></x-datum> </x-paragraph-23>`. The `x-datum` custom element executes a query to REST endpoint `/datum/movies[0].overview` on the Squire server

```
<template id="x-screen-8-template">
  <div class="p-2 flex flex-col h-full w-full">
    <slot class="w-full" name="content"></slot>
  </div>
</template>
<template id="x-verticallist-11-template">
  <div class="p-2 flex flex-col items-stretch gap-2 h-full w-full">
    <slot class="w-full" name="element"></slot>
  </div>
</template>
...
<x-screen-8>
  <x-verticallist-11 slot="content">
    <x-widecard-14 slot="element">
      <x-paragraph-23 slot="content">
        <x-datum value="movies[0].overview" slot="paragraph_text"></x-datum>
      </x-paragraph-23>
      ...
    </x-widecard-14>
    <x-widecard-14 slot="element">
      <x-paragraph-23 slot="content">
        <x-datum value="movies[1].overview" slot="paragraph_text"></x-datum>
      </x-paragraph-23>
      ...
    </x-widecard-14>
    ...
</x-screen-8>
```

Figure 6: Example target code generated by Squire. Template definitions appear above their nested instantiations.

and substitutes the response into the *paragraph_text* slot in its parent component `Paragraph`.

4.3.6 Implementation Details. Squire is implemented as a single page web application, with the frontend implemented in TypeScript and the backend implemented in Python with OpenAI's `gpt-4o` model as the underlying language model servicing Squire requests. The Preview Pane renders the target code generated by Squire in an `<iframe>` tag, which refreshes on each modification to the SquireIR program and Component Definition Store.

5 User Study

Participation. We recruited 11 frontend developers (P1-P11) by posting study invitations to internal communication channels at a large technology company. Participants self-reported an average of 10 years of experience as web frontend developers, with the majority reporting having used generative AI tools in their work, although this was not a specific requirement for participating in the study (6/11 reported using generative AI tools one year or more, 3/11 reported using generative AI for several months, and 2/11 reported having never used generative AI professionally, although subsequent conversations revealed they had explored generative AI in personal settings). As part of the recruitment process, participants were also informed they would receive $15 meal vouchers for their participation. The total study time was 1.5 hours.

Onboarding (30 minutes). To begin, participants were asked to watch a pre-recorded tutorial video demonstrating the different features of Squire. The tutorial included intermittent pauses that asked participants to exercise a feature of Squire to reinforce the learning material.

Open-Ended Tasks (40 minutes). After the tutorial, participants were then given two 20-minute open-ended tasks using Squire during which they were asked to think aloud. The first task asked participants to create a scrollable view for movies—this was the same expository task performed by Mina in §2. The second task

asked participants to create a screen for viewing the details of a hiking trail that might be shown in an app that helps people discover hiking trails. We chose these tasks because they reflected realistic scenarios—each task had analogues in the set of real-world screens we had sampled when designing the component library (§4.2.1)—were described broadly enough to allow room for interpretation, and came from different domains to encourage variation between the two tasks. We used `gpt-4o` to synthesize this sample data by (1) prompting the model to generate a schema from the screen description, (2) prompting the model to generate data that matches the schema, and (3) manually cleaning the output by fixing errors such as broken links.

Post-Interview (20 minutes). After participants completed their tasks, we engaged with them in semi-structured interviews. The interview questions asked participants for overall sentiment, positive and negative impressions of individual features, desires for improvements, comparison with other generative tools they had used, and impressions on how Squire might fit into their existing workflows. Finally, participants completed a survey containing several Likert scale questions about their experience with Squire.

Data collection. Each session's video and audio were recorded and transcribed for qualitative analysis. We additionally collected system traces of every interaction by the participant during their sessions, which was implemented via instrumentation calls that emitted to a log file the timestamps and metadata for every action performed by the participant.

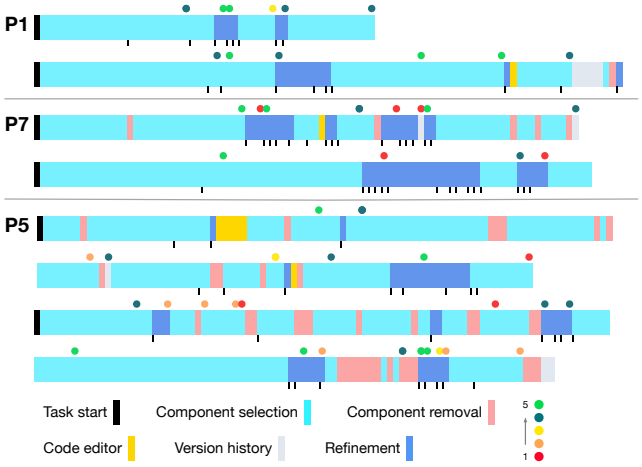


Figure 7: Interaction traces. Interaction traces for three styles of Squire usage. Colored circles above each trace represent subjective ratings. Manually-authored chat interactions are denoted by black ticks. P1 preferred building up the component tree during exploration, pruning unused choices at the end. P7 preferred frequent use of the Refinement Pane. P5 performed the most actions, frequently interspersing additions, removals, and refinement operations.

6 Results

We begin by presenting quantitative findings derived from statistics over interaction traces and quantitative survey responses. We then present qualitative findings derived from interview transcripts and think-aloud data collected while participants performed tasks.

6.1 Quantitative Results

We conducted a quantitative analysis over interaction traces that were automatically recorded while participants interacted with Squire during their open-ended tasks.

6.1.1 Exploration activities. First, we compute summary statistics over the number of actions performed to understand the degree to which participants made use of the different exploration features of Squire. Per task, participants averaged 30.8 component tree additions and only 3.8 component tree deletions. The difference between these two indicates that participants generally preferred to keep the results of different explorations in their component tree rather than discarding them, which we postulate is due to the ease with which Squire allowed them to deselect and navigate between alternatives (DM2).

Participants had access to a code editor for performing manual edits to templates as well as a version history that allowed them to bookmark versions and undo or redo them, but both were used rarely and accounted for only 39 and 35 actions across all participants, respectively. As neither feature is novel or essential to the operation of Squire, we surmise that participants simply did not feel the need to use them.

Examining interactions with the slot query mechanism in detail, we observed that participants averaged 6.9 custom slot queries per task, of which the majority (55%) came from LLM-generated slot suggestions requiring no further revision by the participant, while the remainder were manually authored by the participant. From this we infer that participants felt the generated suggestions were often relevant and met their expectations without any modification. This result is consistent with qualitative feedback that participants felt suggestions predicted their intent well (§6.2.2).

Examining Refinement Pane interactions in detail, participants averaged 14.7 refinement requests per task, of which slightly less than half (44%) of refinements were achieved by making selections with ephemeral controls. The remainder were achieved by directly prompting in the Refinement Pane. On reviewing recordings, a typical pattern we observed was that participants would provide an initial refinement query that coarsely represented their intent, then used subsequent ephemeral controls to evaluate different options to fine-tune the result.

To further understand the nature of the refinements participants requested, we collected all refinement prompts, classified them into categories, and tabulated the number of occurrences in each category. The three most frequently occurring categories of prompts were changes to layout (30%), color (21%), and icon appearance (16%), where layout refinements asked to rearrange elements on the screen (e.g. "make this into a single row"), color refinements asked to modify the color of elements (e.g. "lighten the button background color"), and icon refinements asked to either add or change the appearance of an icon (e.g. "change the icon to a conversation bubble"). Additionally, 13% of refinements were of a very general nature, asking the system to help with ideation ("how about some new fonts"), or to make broad subjective leaps ("make it pop"). In this way, refinement seemingly served two different purposes: first, to make directed edits when participants had specific edits in mind,

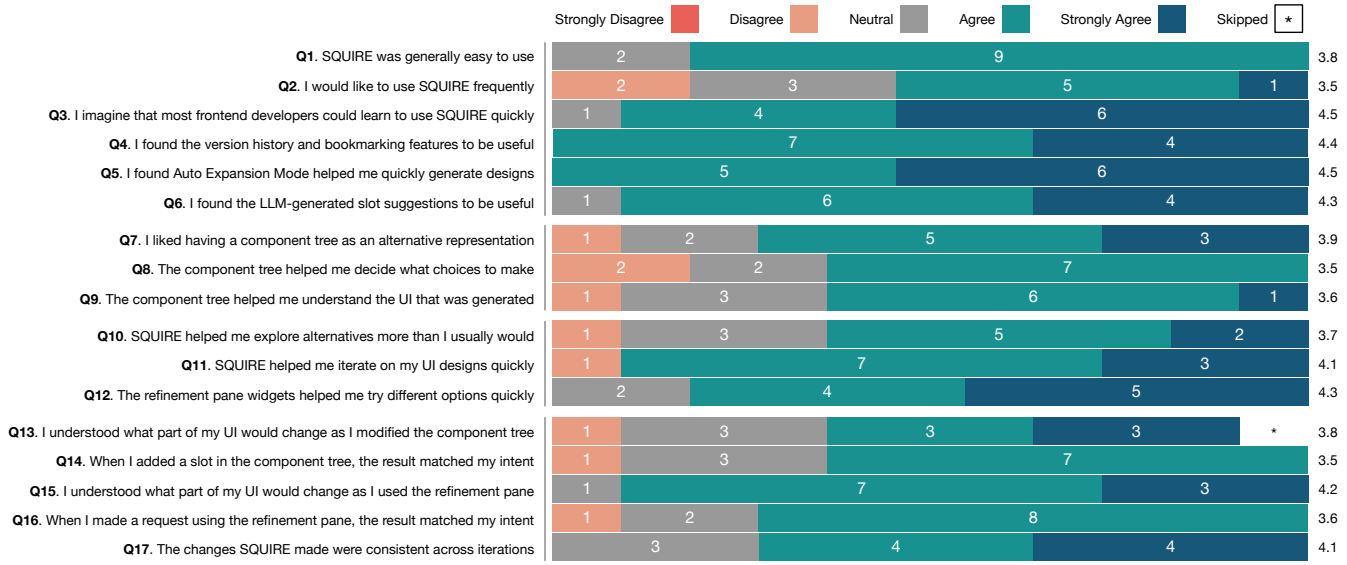


Figure 8: Post-session questionnaire results. Questions are grouped into 4 focus areas from top to bottom: general sentiment, component tree, exploration, and scoped interaction. Different bars indicate responses per rating, with averages on the right. Participants expressed generally positive sentiment towards all features of Squire. One participant did not answer Q13, so we only collected 10/11 responses.

and second, to come up with fresh ideas without the intention of a specific appearance or goal.

6.1.2 Self-reported quality ratings. As participants performed their tasks, they self-reported numerical ratings (1=worst, 5=best) to judge the result of their most recent interaction at that point (this appeared in the Squire UI as a set of rating buttons in the upper right corner, which when pressed would save the rating to the system trace). Note that because participants performed so many actions over the course of a session, they were not expected to provide ratings for every action, but rather periodically as they saw fit.

Overall, participants rated 3 or above in 72% of cases, and rated 5 the most frequently of all (32%). However, in 28% of cases, participants rated less than 3. Many of these low ratings happened immediately after a change had failed to meet the participant's expectations of aesthetic quality. In the case of component tree modifications, this often corresponded to a component choice that the participant disagreed with. In the case of template refinements in the Refinement Pane, this corresponded to cases where the refinement modified the template in a way that failed to meet the participant's quality bar, or otherwise failed to interpret and execute the user's query appropriately. While participants were generally positive about the results they were able to achieve with Squire, participants also encountered limitations with respect to model output quality that could be addressed with further advances in model design knowledge [49, 50].

6.1.3 Usage styles. Although Squire provides a more guided experience than chat-only tools, we inspected visualizations of interaction traces to identify the degree to which Squire might still encourage a diversity of usage styles. Figure 7 shows visualizations of interaction traces that exhibit the three distinct patterns of Squire usage we identified, roughly in order of increasingly frequent activity from top (P1) to bottom (P5). P1 generally preferred building out the component tree to explore different options and only deleted nodes once at the end of the task to prune unused choices, which is consistent with the finding that deletion was generally infrequent (§6.1.1). P7 preferred more frequent use of the Refinement Pane, submitting 2.6 times as many refinement requests as P1 (42 vs 16). P5 submitted the most actions of all participants, executing 1.9 times more actions than the median participant (480 vs 245), with frequent additions, removals, and refinement operations interspersed.

6.1.4 Questionnaire responses. Figure 8 shows a breakdown of responses to the post-session questionnaire, which asked participants to indicate level of agreement with statements meant to elicit subjective ratings of various aspects of Squire. Participants were also asked to give optional justification for their answers if they wished. The questions span four major themes, shown grouped from top to bottom in Figure 8: general sentiment about Squire and its individual features (Q1-Q6), effectiveness of the component tree as a representation to aid comprehension (Q7-Q9), effectiveness of Squire as an exploration aid (Q10-Q12), and effectiveness of scoped interactions (Q13-Q17).

General sentiment (Q1-Q6): Participants felt that Squire was easy to learn (Q3) and use (Q1), expressing positive sentiment towards all features of Squire (Q4-Q6): 5/6 of the questions in this group received at least 9 ratings of Agree or Strongly Agree. However, one question elicited more mixed responses: when asked whether they would like to use Squire frequently (Q2), 2 participants disagreed and provided explanations in their questionnaire responses. One participant felt that while Squire was well-suited for creative

exploration, "there [were] people on the team who handle this responsibility" for them already (P5). Another participant felt that the experience "felt alien compared to [their] normal flow for prototyping or scaffolding a mobile UI" (P6), suggesting that lack of familiarity may have been a barrier to adoption that might have been overcome with more experience using the tool.

Component tree as a comprehension aid (Q7-Q9): Participants were also generally positive regarding the visualization of the component tree (DM3) as an aid to comprehension (Q7), both when deciding which choices to make (Q8), and understanding the resulting generated UI (Q9). One participant who disagreed with Q8 volunteered that the visualization became difficult to navigate as the tree became too large (P6), suggesting room for usability improvements, such as zoom or expand/collapse options to better aid focus on subsets of the tree. Additionally, some participants who answered Q8 positively offered improvements to increase visual clarity and density of the component tree further: removing excess horizontal indentation (P9), making textual descriptions less prominent (P4, P5), and adding color coding to better portray visual correspondence with the preview (P9).

Squire as an exploration aid (Q10-Q12): The third group of questions (Q10-Q12) asked participants to indicate the degree to which Squire helped them explore and iterate (DM2). While all questions elicited high ratings, participants were particularly positive about the ephemeral controls (Q12), with 5 participants indicating strong agreement. On reviewing responses, we found that participants praised this feature for its accuracy (P1) and the way it encouraged quick tinkering (P11). We will have more to say on this point in §6.2.4.

Scoped interaction (Q13-Q17): The last group of questions elicited participants' sentiment on whether the scoped nature of component tree and Refinement Pane interations were effective at conveying changes (Q13,Q15), and whether those changes actually satisfied their intents (Q14,Q16). In both cases, the majority of participants expressed positive sentiment, indicating that participants generally felt that Squire gave them a sense of control over changes (DM1). We elaborate on this point further in §6.2.4.

6.2 Qualitative Results

We conducted a lightweight thematic analysis of transcribed study recordings to identify common themes.

6.2.1 Envisioning design using sample data. Participants noted that starting from sample data was a natural and effective approach for designing user interfaces, as "a lot of times we are constrained by what data is available to us so if you want to develop a new view… [you have to think] data first and build the views around it" (P8). By starting with a data-first approach, Squire aided the process of coming up with possible designs, as it made it "easier to envision" possibilities for rendering data (P6). Participants also observed that the explicit representation of data in Squire gave them more control over the decision-making process than chat-style assistants: "instead of the model generating the data and then making decisions based on that, at least you have time to intervene" and guide the model towards intended behavior (P10). One participant enjoyed the data-first approach enough that they proposed as a future enhancement an interactive flow for guiding users towards authoring project descriptions: "Maybe prompting the user to say, hey, what do you want to create? Cool. Well, can you provide me some data? Cool. Or maybe this is missing or something like that" (P10).

6.2.2 Proactive prediction of design possibilities. In contrast to most chat-based generative tools where the AI reactively responds to user requests, several participants expressed that Squire would seemingly anticipate their intent and suggest custom slots that they had not yet verbalized: "So it's doing a pretty good job of anticipating the kinds of things that I might want to put on, and it adds more as I go, which is nice." (P3) Some participants indicated Squire was so effective at suggesting custom slots that it was "kind of reading my mind" and seemingly clairvoyant in the way it "worked like magic" (P9). In addition to predicting the participants' non-verbalized intent, Squire sometimes offered suggestions that led users to explore possibilities they had not yet even considered (DM2), thus acting as a creative aid: "Often times it would suggest components I hadn't yet thought of" (P5).

Mirroring their feedback toward slot suggestions, participants stated that the Refinement Pane should have also incorporated more proactive suggestions: "when you double click something, it [should] sort of maybe already present you with some options on like what you'd want to do" (P2) and "similar to how you give the suggestions for slots, you [should] give some suggestions for instructions as well" (P8).

6.2.3 Accelerating iteration and exploration rather than automating design. Participants noted that current generation AI assistants like ChatGPT or v0 seemed to be good for automating the process of generating project skeletons, but made it hard to iterate further on results, with one participant noting "they'll just generate a whole application and then come up with some sort of a UI." Further iteration through additional prompting was a struggle when attempting to achieve more targeted refinement: "after they generate you could make smaller modifications. But sometimes it struggles to do that" (P10).

In comparison, participants enjoyed that Squire offered a different, more iterative and exploratory interaction (DM2): "[tools like ChatGPT] create the skeleton in a way is how I view it, whereas here it feels like very much a visual exploration" (P5). One participant pointed out that Squire even felt like an improvement over visual mockup tools like Sketch "because it would actually help prototype different layouts so much quicker… it's a step above like regular Sketch type stuff" (P1) because it creates high-fidelity designs through code.

As a counterpoint, however, one participant noted that Squire seemed to serve better as a complement, rather than a replacement: "I think Squire was better at exploration of different visual options and interactively modifying styles while others feel more suited for spinning up web applications and implementing features" (P5). Indeed, some participants noted that although they may have been satisfied visually, there was remaining concern that the output was not in their preferred implementation framework: "many teams are tightly bound to certain frontend libraries like React/Angular and CSS libraries; not sure how well a tool like this can integrate into an application that is built with these other libraries" (P1).

6.2.4 Building confidence in making changes through control. Participants noted that they felt confident when making changes because Squire provided explicit controls (DM1): "Controlled yeah more controlled, like you're more sure what's gonna happen. I think that this is a better approach to do it…controlled to a particular region or place to implement is a very nice way, I think, to be very confident" (P8). Another participant noted that the ability to precisely change elements of the component tree felt like missing functionality in other tools: "Now that I use it more, I think that's what['s] missing in tools like v0.dev or things I use, because it's very hard to just like modify a part of the whole component tree" (P8). This point was reinforced by another participant who noted that tools like ChatGPT provide no special features for making modifications other than additional prompting, whereas with Squire "you have more control of making those modifications afterwards" (P10). Another participant felt that it increased their confidence that Squire would do what they asked: "Squire generally didn't go off the rails like most other generative AI tools I've used or seen" (P11).

As a result, Squire seemingly encouraged participants to tinker frequently without worrying about implementing and undoing unfruitful explorations (DM2). For example, one participant mentioned that "being able to quickly choose the color like this, like just tinker with all that, [is] super easy" whereas "if you're just dealing with raw code, as I said, like it's not as quick to be able to do this sort of tinkering" (P11). In addition to reducing the friction to make changes, another participant observed that the tool seemed to encourage exploration of new ideas by removing risk: "I could take that risk anytime to like explore new design types…It's like a fail-fast fashion where I can change fast [without] rewriting the whole thing again" (P9). This sense of risk-free exploration also provided the participants a sense of freedom: "It gives you that autonomy like, ok, there are other ways how you can show this UI and you can risk with it" (P9).

Still, while some preferred the guided nature of Squire, one participant did mention wishing for a way to escape into a more free-form prompting interaction: "I think it's a little bit more guided experience where like, for example, like if I interact with an LLM, it's mostly through their prompt. This type of interaction is new. I haven't worked with it before, where you have a little bit more, you have more knobs and like a tree structure to be able to work with. But I kept on, it looks like I kept on still trying to interact with the LLM by selecting elements on the Preview Pane and then instructing what to do" (P10).

Study Limitations. User study participants had no prior experience with Squire and were given only 20 minutes per task. This limited our ability to evaluate Squire's capacity to scale to larger and more complex screens. While a comprehensive evaluation of scalability is the subject of future work, we refer to Figure 9 in the Appendix for a sample of screens that qualitatively convey the variety of screens possible with Squire.

A controlled user study is a low-stakes environment in which decisions have few real-world consequences, so participants may have been less demanding when accepting Squire's suggestions than they might have been in their day-to-day work. A future in-situ study could evaluate whether a more natural setting affects developers' experience with Squire.

7 Discussion and Future Work

We now discuss the broader implications of our findings and potential future directions for rapid prototyping tools.

Combining scoped and unscoped affordances. The developers in our user study reported confidence that Squire would do as intended when they made changes, indicating that the explicitly scoped interactions (slot queries and refinement queries) were successful in helping developers intuitively understand the region of the UI and the aesthetic aspects of the UI (in the case of refinement queries) that would be changed. Still, while these findings affirmed the value of scoped affordances, we note that prompting for directed changes is only one way in which developers use generative assistants. Indeed, we saw in our user study that participants sometimes wanted to riff with the model and see what happened when providing very general and ambiguous queries ("make it pop") in the hopes of being inspired serendipitously. This points to a need for future rapid prototyping tools to combine both scoped and unscoped interactions that serve different purposes—one for controlled exploration through guided modification, and another for radical inspiration through larger leaps—where a future challenge will be to develop representations that are compatible with both.

Combining direct manipulation and conversational interfaces. In contrast to direct manipulation design tools [14, 40], Squire employs a predominantly conversational interface. However, conversational and direct manipulation approaches need not be mutually exclusive and offer different advantages: natural language is expressive but potentially ambiguous, while direct manipulation is precise but limited to specific gestures. Future work could investigate how Squire could be extended to support some forms of direct manipulation. For example, some participants (P2,P4,P5,P8) expressed a desire for drag-and-drop to rearrange components, a feature that could potentially be implemented via programmatic tree-rewriting. For direct manipulations where the users' goals are more complex, programming-by-demonstration [8] may be a promising direction to infer intent, as it has been applied successfully in the domains of graphic design [19] and visualization [38].

Grounding creation in known requirements. In Squire, developers provide initial context in the form of project descriptions that include sample data. Our user study participants enjoyed this aspect of the tool, indicating that it fits naturally in cases where project requirements impose constraints on what data is available. However, it is still a limitation of Squire that developers must pay the cost to provide sample data up front. In cases where this data is already available, this cost may be minimal, but in cases where it is not, developers must spend time to create it themselves. For our user study, we synthesized this sample data using an LLM (`gpt-4o`) and estimate that each required less than half an hour of effort to create, which is a small but still non-negligible amount of effort. Future work should investigate ways to integrate synthetic data generation to reduce the burden on the user.

While Squire employs generative models, we note that classical approaches to model-based UI have previously investigated deterministic approaches for the related task of component selection from specifications [55]. Future work may combine deterministic and probabilistic techniques into one system, where deterministic rules are used for fast matching from data to compatible elements,

while probabilistic models may be used to make higher-level decisions that require more sophisticated design and world knowledge.

Thinking further, we note that providing sample data is only one instance of a broader class of potential techniques to support grounded authoring of UI (i.e. techniques to steer models toward a more constrained class of intended outputs). One promising direction was suggested by two participants (P2,P8), who noted that it would be useful for them to provide predefined stylesheets, branding guidelines, or other design patterns as a way to tailor outputs to their particular team's needs. Future work should explore such input modalities that both fit within developers' existing practices and are lightweight enough for developers so as not to be onerous to produce. Specialized tools for generating this form of context may be another promising area of inquiry.

Connection to overview+detail interfaces. Squire's user interface adopts a variant of the overview+detail pattern [20], with the component tree and preview providing overview and detail, respectively. In Squire, the overview granularity is not dynamically adjustable, which limits users' ability to focus on subtrees within more complex component trees: recall that one participant (P6) suggested adding zoom or expand/collapse to aid focus (§6.1.4). Recent work on malleable overview-detail interfaces [28] points to the benefits of dynamically tunable overviews, of which zoom and expand/collapse are only two possibilities. Future work should investigate how these interactions might aid exploration in complex designs, while also evaluating potential trade-offs around increased context-switching and reduced spatial awareness.

Productionizing prototypes. Some participants (P1,P6,P10,P11) voiced concern whether it would be difficult to integrate what they had built in Squire into their existing codebases, as development teams' technology stacks vary in their choice of third-party libraries and UI frameworks. For the purpose of this paper, we implemented Squire to support standard HTML, CSS, and JavaScript to minimize dependencies and reduce necessary domain knowledge for users. However, the conceptual framework proposed in this paper remains theoretically compatible with feature-rich frameworks like React, Vue.js, or Svelte, as they are still fundamentally based on nesting of components with encapsulated behavior and styles, albeit with their own unique engineering practicalities.

Squire makes a simplifying assumption to fetch backend data using JSONPath queries serviced by the Squire backend itself. This is one area where the resulting code would need further editing to be integrated into an existing codebase for further development, either by supporting the JSONPath query interface, or by replacing datum instances with another data access mechanism. Squire could be extended with support for external REST APIs or other backend request conventions by supplementing the data schema with external resource URLs and revising the datum instance implementation to query to those external resources instead.

Going beyond visual exploration. Squire is a visual prototyping tool—it only generates one screen at a time and does not generate interaction-specific code, such as transitions, animations, or navigation to other screens. Based on the results of our user studies, we have found that developers still find the supported functionality highly useful, but future work should investigate support for multi-screen flows and dynamic behavior, along with corresponding extension to provide scoped exploration across those new dimensions.

8 Conclusion

We present a system for user interface development called Squire designed for incremental and interactive prototyping of mobile web user interfaces. As the central concept in Squire, we propose a domain-specific representation called SquireIR for encoding UI structure as a high level composition of components that can be explored and refined through slot queries and template refinement. We report a user study with experienced frontend developers that demonstrates Squire's ability to support exploration, refinement, and comprehension of generated UI. Our findings reveal future opportunities for tools that incorporate scoped UI exploration and refinement.

References

[1] Anthropic. 2025. Claude Artifacts. https://support.anthropic.com/en/articles/9487310-what-are-artifacts-and-how-do-i-use-them. Accessed: 2025-04-04.

[2] Christian Bird, Denae Ford, Thomas Zimmermann, Nicole Forsgren, Eirini Kalliamvakou, Travis Lowdermilk, and Idan Gazit. 2023. Taking Flight with Copilot: Early insights and opportunities of AI-powered pair-programming tools. *Queue* 20, 6 (Jan. 2023), 35–57.

[3] Bill Buxton. 2007. *Sketching User Experiences: Getting the Design Right and the Right Design.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[4] Gaëlle Calvary, Joëlle Coutaz, Laurent Bouillon, M. Florins, Quentin Limbourg, L. Marucci, Fabio Paterno, C. Santoro, N. Souchon, David Thevenin, and Jean Vanderdonckt. 2002. The CAMELEON reference framework.

[5] Xiang 'Anthony' Chen, Tiffany Knearem, and Yang Li. 2025. The GenUI Study: Exploring the Design of Generative UI Tools to Support UX Practitioners and Beyond. In *Proceedings of the 2025 ACM Designing Interactive Systems Conference (DIS '25)*. Association for Computing Machinery, New York, NY, USA, 1179–1196.

[6] Ruijia Cheng, Titus Barik, Alan Leung, Fred Hohman, and Jeffrey Nichols. 2024. BISCUIT: Scaffolding LLM-Generated Code with Ephemeral UIs in Computational Notebooks. In *2024 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 13–23.

[7] Design Council. 2004. Framework for Innovation. https://www.designcouncil.org.uk/our-resources/framework-for-innovation/

[8] Allen Cypher. 1993. *Watch What I Do.* MIT Press.

[9] Steven P Dow, Alana Glassco, Johnathan Kass, Melissa Schwarz, and Scott R. Klemmer. 2009. *The Effect of Parallel Prototyping on Design Performance, Learning, and Self-Efficacy.* Technical Report. Stanford University.

[10] Steven P. Dow, Alana Glassco, Jonathan Kass, Melissa Schwarz, Daniel L. Schwartz, and Scott R. Klemmer. 2011. Parallel prototyping leads to better design results, more divergence, and increased self-efficacy. *ACM Trans. Comput.-Hum. Interact.* 17, 4 (Dec. 2011).

[11] Peitong Duan, Chin-Yi Cheng, Gang Li, Bjoern Hartmann, and Yang Li. 2024. UICrit: Enhancing Automated Design Evaluation with a UI Critique Dataset. In *Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology (UIST '24)*. Association for Computing Machinery, New York, NY, USA.

[12] Peitong Duan, Jeremy Warner, Yang Li, and Bjoern Hartmann. 2024. Generating Automatic Feedback on UI Mockups with Large Language Models. In *Proceedings of the CHI Conference on Human Factors in Computing Systems (CHI '24)*. Association for Computing Machinery, New York, NY, USA.

[13] Yanai Elazar, Nora Kassner, Shauli Ravfogel, Abhilasha Ravichander, Eduard Hovy, Hinrich Schütze, and Yoav Goldberg. 2021. Measuring and Improving Consistency in Pretrained Language Models. *Transactions of the Association for Computational Linguistics* 9 (2021), 1012–1031.

[14] Figma. 2025. Figma. https://figma.com. Accessed: 2025-07-03.

[15] Krzysztof Gajos and Daniel S. Weld. 2004. SUPPLE: automatically generating user interfaces. In *Proceedings of the 9th International Conference on Intelligent User Interfaces (IUI '04)*. Association for Computing Machinery, New York, NY, USA, 93–100.

[16] John D. Gould and Clayton Lewis. 1985. Designing for usability: key principles and what designers think. *Commun. ACM* 28, 3 (March 1985), 300–311.

[17] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. 2024. The Llama 3 Herd of Models. *arXiv preprint arXiv:2407.21783* (2024).

[18] Björn Hartmann, Sean Follmer, Antonio Ricciardi, Timothy Cardenas, and Scott R. Klemmer. 2010. d.note: revising user interfaces through change tracking, annotations, and alternatives. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '10)*. Association for Computing Machinery, New York, NY, USA, 493–502.

[19] Brian Hempel, Justin Lubin, and Ravi Chugh. 2019. Sketch-n-Sketch: Output-Directed Programming for SVG. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology (UIST '19)*. Association for Computing Machinery, New York, NY, USA, 281–292.

[20] Kasper Hornbæk and Erik Frøkjær. 2001. Reading of electronic documents: the usability of linear, fisheye, and overview+detail interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '01)*. Association for Computing Machinery, New York, NY, USA, 293–300.

[21] David G Jansson and Steven M Smith. 1991. Design fixation. *Design studies* 12, 1 (1991), 3–11.

[22] Won Chul Kim and James D. Foley. 1993. Providing high-level control and expert assistance in the user interface presentation design. In *Proceedings of the INTERACT '93 and CHI '93 Conference on Human Factors in Computing Systems (CHI '93)*. Association for Computing Machinery, New York, NY, USA, 430–437.

[23] Kristian Kolthoff, Felix Kretzer, Lennart Fiebig, Christian Bartelt, Alexander Maedche, and Simone Paolo Ponzetto. 2024. Zero-Shot Prompting Approaches for LLM-based Graphical User Interface Generation. *arXiv preprint arXiv:2412.11328* (12 2024).

[24] Yoonjoo Lee, Kihoon Son, Tae Soo Kim, Jisu Kim, John Joon Young Chung, Eytan Adar, and Juho Kim. 2024. One vs. Many: Comprehending Accurate Information from Multiple Erroneous and Inconsistent AI Generations. In *Proceedings of the 2024 ACM Conference on Fairness, Accountability, and Transparency (FAccT '24)*. Association for Computing Machinery, New York, NY, USA, 2518–2531.

[25] 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.

[26] Yuwen Lu, Alan Leung, Amanda Swearngin, Jeffrey Nichols, and Titus Barik. 2025. Misty: UI Prototyping Through Interactive Conceptual Blending. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems (CHI '25)*. Association for Computing Machinery, New York, NY, USA.

[27] Ping Luo, Pedro Szekely, and Robert Neches. 1993. Management of interface design in humanoid. In *Proceedings of the INTERACT '93 and CHI '93 Conference on Human Factors in Computing Systems (CHI '93)*. Association for Computing Machinery, New York, NY, USA, 107–114.

[28] Bryan Min, Allen Chen, Yining Cao, and Haijun Xia. 2025. Malleable Overview-Detail Interfaces. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems (CHI '25)*. Association for Computing Machinery, New York, NY, USA.

[29] Kevin Moran, Carlos Bernal-Cárdenas, Michael Curcio, Richard Bonett, and Denys Poshyvanyk. 2020. Machine Learning-Based Prototyping of Graphical User Interfaces for Mobile Apps. *IEEE Transactions on Software Engineering* 46, 2 (2020), 196–221.

[30] Kevin Moran, Boyang Li, Carlos Bernal-Cárdenas, Dan Jelf, and Denys Poshyvanyk. 2018. Automated reporting of GUI design violations for mobile apps. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 165–175.

[31] Mozilla. 2025. Using shadow DOM. https://developer.mozilla.org/en-US/docs/Web/API/Web_components/Using_shadow_DOM. Accessed: 2025-03-10.

[32] Mozilla. 2025. Web Components. https://developer.mozilla.org/en-US/docs/Web/API/Web_components. Accessed: 2025-03-10.

[33] D. R. Olsen. 1989. A programming language basis for user interface. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '89)*. Association for Computing Machinery, New York, NY, USA, 171–176.

[34] OpenAI. 2025. ChatGPT. https://chat.openai.com/chat. Accessed: 2025-03-31.

[35] OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. GPT-4 Technical Report. *arXiv preprint arXiv:2303.08774* (03 2023).

[36] A.R. Puerta. 1997. A model-based interface development environment. *IEEE Software* 14, 4 (1997), 40–47.

[37] Angel Puerta, Michael Micheletti, and Alan Mak. 2005. The UI pilot: a model-based tool to guide early interface design. In *Proceedings of the 10th International Conference on Intelligent User Interfaces (IUI '05)*. Association for Computing Machinery, New York, NY, USA, 215–222.

[38] Arvind Satyanarayan, Bongshin Lee, Donghao Ren, Jeffrey Heer, John Stasko, John Thompson, Matthew Brehmer, and Zhicheng Liu. 2020. Critical Reflections on Visualization Authoring Systems. *IEEE Transactions on Visualization and Computer Graphics* 26, 1 (2020), 461–471.

[39] Jaejung Seol, Seojun Kim, and Jaejun Yoo. 2025. PosterLlama: Bridging Design Ability of Language Model to Content-Aware Layout Generation. In *Computer Vision – ECCV 2024*, Aleš Leonardis, Elisa Ricci, Stefan Roth, Olga Russakovsky, Torsten Sattler, and Gül Varol (Eds.). Springer Nature Switzerland, Cham, 451–468.

[40] Sketch. 2025. Sketch. https://sketch.com. Accessed: 2025-07-03.

[41] 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. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems (CHI '24)*. Association for Computing Machinery, New York, NY, USA.

[42] Amanda Swearngin, Chenglong Wang, Alannah Oleson, James Fogarty, and Amy J. Ko. 2020. Scout: Rapid Exploration of Interface Layout Alternatives through High-Level Design Constraints. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems (CHI '20)*. Association for Computing Machinery, New York, NY, USA, 1–13.

[43] Pedro Szekely, Ping Luo, and Robert Neches. 1992. Facilitating the exploration of interface design alternatives: the HUMANOID model of interface design. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '92)*. Association for Computing Machinery, New York, NY, USA, 507–515.

[44] Zecheng Tang, Chenfei Wu, Juntao Li, and Nan Duan. 2024. LayoutNUWA: Revealing the Hidden Layout Expertise of Large Language Models. In *The Twelfth International Conference on Learning Representations*.

[45] UXArchive. 2025. UXArchive. https://uxarchive.com. Accessed: 2025-04-09.

[46] 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 *Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems (CHI EA '22)*. Association for Computing Machinery, New York, NY, USA.

[47] Jean M. Vanderdonckt and François Bodart. 1993. Encapsulating knowledge for intelligent automatic interaction objects selection. In *Proceedings of the INTERACT '93 and CHI '93 Conference on Human Factors in Computing Systems (CHI '93)*. Association for Computing Machinery, New York, NY, USA, 424–429.

[48] Vercel. 2025. Vercel v0. https://v0.dev. Accessed: 2025-03-31.

[49] Jason Wu, Yi-Hao Peng, Xin Yue Amanda Li, Amanda Swearngin, Jeffrey P Bigham, and Jeffrey Nichols. 2024. UIClip: A Data-driven Model for Assessing User Interface Design. In *Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology (UIST '24)*. Association for Computing Machinery, New York, NY, USA.

[50] Jason Wu, Eldon Schoop, Alan Leung, Titus Barik, Jeffrey Bigham, and Jeffrey Nichols. 2024. UICoder: Finetuning Large Language Models to Generate User Interface Code through Automated Feedback. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, Kevin Duh, Helena Gomez, and Steven Bethard (Eds.). Association for Computational Linguistics, Mexico City, Mexico, 7511–7525.

[51] Mingyue Yuan, Jieshan Chen, Yongquan Hu, Sidong Feng, Mulong Xie, Gelareh Mohammadi, Zhenchang Xing, and Aaron Quigley. 2024. Towards Human-AI Synergy in UI Design: Enhancing Multi-Agent Based UI Generation with Intent Clarification and Alignment. *arXiv preprint arXiv:2412.20071* (2024).

[52] J.D. Zamfirescu-Pereira, Eunice Jun, Michael Terry, Qian Yang, and Bjoern Hartmann. 2025. Beyond Code Generation: LLM-supported Exploration of the Program Design Space. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems (CHI '25)*. Association for Computing Machinery, New York, NY, USA, 1–17.

[53] 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 (DIS '23)*. Association for Computing Machinery, New York, NY, USA, 2206–2220.

[54] 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 (CHI '23)*. Association for Computing Machinery, New York, NY, USA.

[55] Brad Vander Zanden and Brad A. Myers. 1990. Automatic, look-and-feel independent dialog creation for graphical user interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '90)*. Association for Computing Machinery, New York, NY, USA, 27–34.

A Appendix

Table 1: The Squire component library.

Component	Description
Button	A button with a label.
Caption	A caption for describing a corresponding image or other media.
Carousel	A horizontally scrollable collection of images or cards.
Checkmark	A check mark icon for indicating success or acceptance based on a boolean value.
CircularImage	A single image with a circular form factor.
Date	A date formatted to be human-readable.
Form	A form for collecting user input.
Grid	Use a Grid to show several items of the same type to the user. Generally, a grid item is a single image or visually distinctive element without much text. The user may click on it to view more details about it.
Heading	A heading that shows text prominently. A good choice for titles and other section headers.
Hyperlink	A link to another page or resource.
Icon	An icon.
InlineList	Use an InlineList to present several items of the same type to the user in a horizontal arrangement.
Literal	A hard-coded string that never changes.
Login	A login form.
Navigation	A navbar on the side with links to other pages within the app.
Option	A selectable option amongst many.
OptionGroup	A group of selectable options. Only one may be selected at a time.
Panel	A panel is a visually distinct group of elements that are displayed together. They are surrounded by a border and padding to distinguish them from surrounding elements.
Paragraph	A paragraph of text. A good choice for text content that consists of several sentences.
Password	A password input field.
PlayControls	Play controls for video or audio player.
ProfilePage	A page for viewing a user profile and associated settings.
ProgressBar	A progress bar indicator. This is a horizontal bar that is filled between 0 and 100.
ProgressRing	A circular progress indicator. Given a numeric value between 0 and 100, this element displays a ring with a percentage of the ring filled in.
Score	A badge for showing numerical scores, where higher is better. Useful for visualizing quantitative values.
Screen	The top level of an individual screen.
SearchBar	A search box for finding results.
SquareImage	A single image with a square form factor.
TabBar	If there are several distinct views in the program, then use a tab bar to separate them into separate tabs that can be selected by the user.
TallCard	Use a TallCard to represent a single item out of many. The contents are arranged in a vertical layout with an image at the top.
TextInput	A text input field.
Toggle	A setting toggle that can be either on or off based on its boolean value.
VerticalList	Use a VerticalList to present several homogeneous items in a vertical arrangement.
Video	A video player.
WideCard	Use a WideCard to represent a single item out of many. The contents are arranged in a horizontal layout with an image on the side.

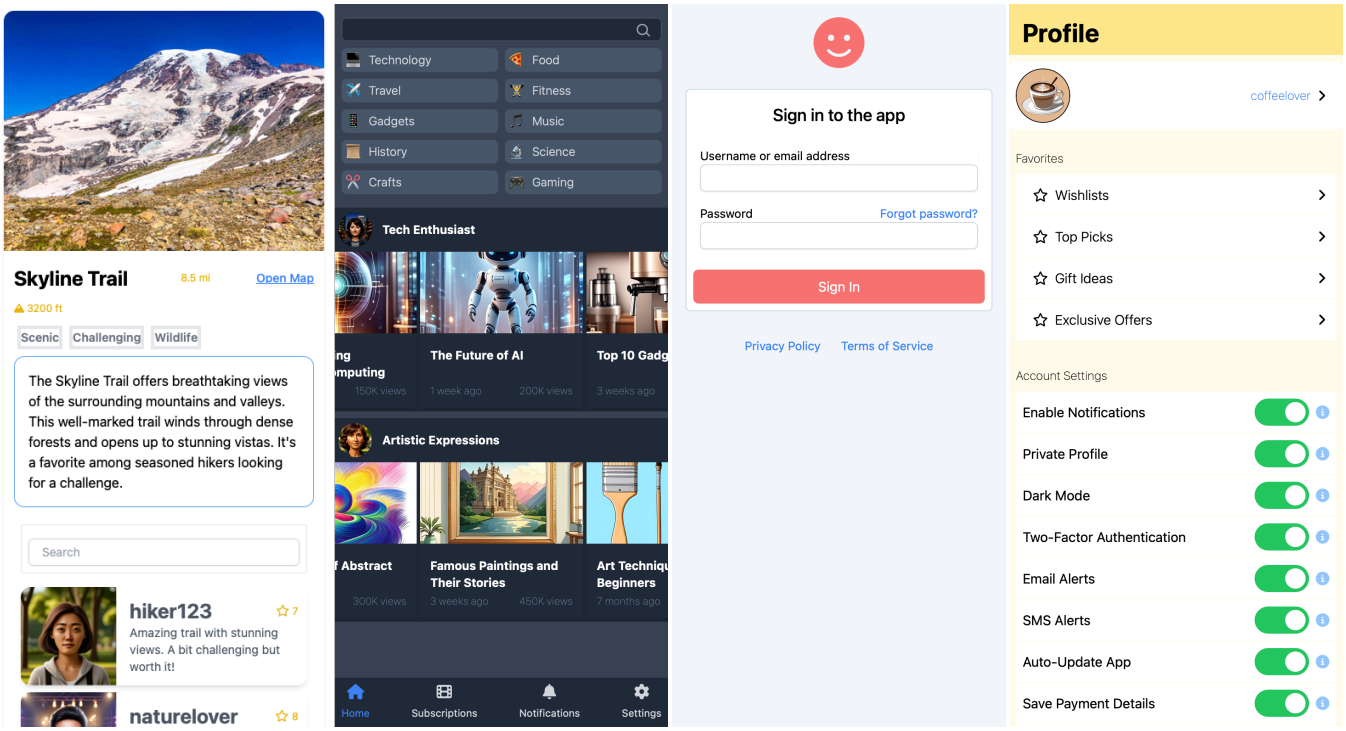


Figure 9: Sample app screens created with SQUIRE. These screens depict the following use cases: hiking trail details, social video streaming, app log in, and profile preferences (left-to-right). The first was created by a participant (P7) during the user study, while the rest were created by the authors to demonstrate a broader variety of screens beyond the user study setting.