Generative Trigger-Action Programming with Ply

Timothy J. Aveni
Electrical Engineering and Computer Sciences
University of California, Berkeley
Berkeley, California, USA
tja@berkeley.edu

Hila Mor
Electrical Engineering and Computer Sciences
University of California, Berkeley
Berkeley, California, USA
hilamor@berkeley.edu

Armando Fox
Electrical Engineering and Computer Sciences
University of California, Berkeley
Berkeley, California, USA
fox@cs.berkeley.edu

Björn Hartmann
Electrical Engineering and Computer Sciences
University of California, Berkeley
Berkeley, California, USA
bjoern@eecs.berkeley.edu

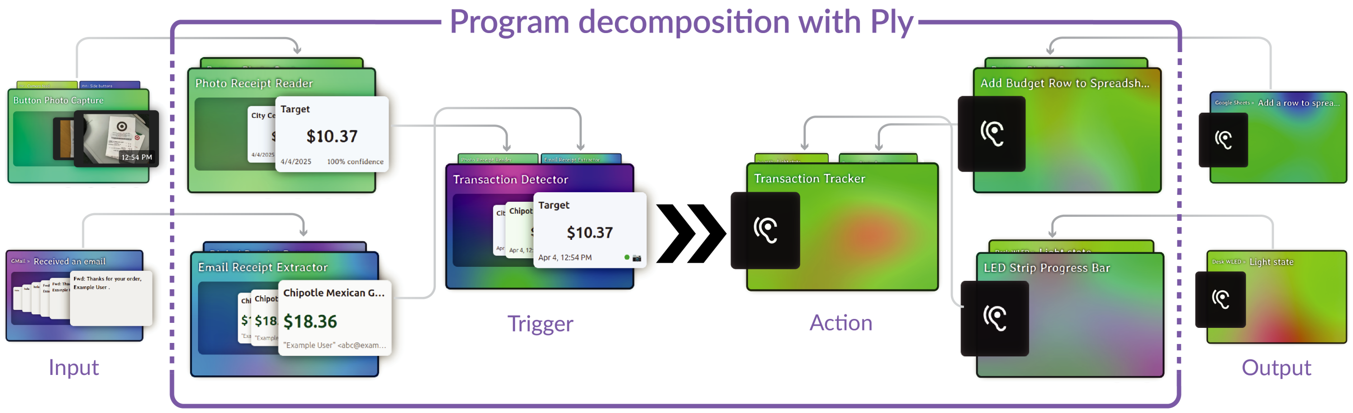


Figure 1: This trigger-action program, built in Ply, tracks personal spending when new transactions are entered into the program. By building up components based on camera and email inputs, the user creates a high-level "transaction detector" using Ply's code generation features. Similarly, the action taken when a new transaction is detected is a composition of a spreadsheet ledger and an LED strip that progressively lights up as the total tracked amount increases.

Abstract

Trigger-action programming has been a success in end-user programming. Traditionally, the simplicity of links between triggers and actions limits the expressivity of such systems. LLM-based code generation promises to enable users to specify more complex behavior in natural language. However, users need appropriate ways to understand and control this added expressive power. We introduce *Ply*, a system that tackles this challenge through the following techniques: (1) a *layer* abstraction that enables decomposition into smaller building blocks; (2) generated visualizations at each layer boundary to enable interrogation of the behavior of generated code; and (3) generated customizable parameters, with associated configuration UIs, to allow users to tune each layer's behavior. We offer a technical evaluation of Ply, demonstrating how custom programs can be authored and tested incrementally using this layered trigger-action technique. Additionally, we describe a first-use study with seven participants, demonstrating where and how Ply's generative features can affect how users build programs.

CCS Concepts

• **Human-centered computing** → **Interactive systems and tools**; **Natural language interfaces**; *Visualization systems and tools*.

Keywords

trigger-action programming, programming systems, code generation, LLMs, software architecture

ACM Reference Format:
Timothy J. Aveni, Hila Mor, Armando Fox, and Björn Hartmann. 2025. Generative Trigger-Action Programming with Ply. In *The 38th Annual ACM Symposium on User Interface Software and Technology (UIST '25), September 28–October 01, 2025, Busan, Republic of Korea.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3746059.3747638

1 Introduction

Trigger-action programming offers an elegant interface to construct simple programs that result in customized behavior for software or devices. In popular automation software such as IFTTT (If This Then That) [28], trigger-action programs use a straightforward model of computation: a single conditional statement, in which the "this" and the "that" can be chosen from a list of various integrations (for example: *update your Android wallpaper every time you post a photo on Instagram*).

Such automations are simple to author, but they require an "impedance match" between a trigger and an action — *i.e.,* the data sent from trigger to action is in a shared format, at a common level of abstraction. When deeper customization is needed (for example: *update a spreadsheet of coastal locations visited every time you post a photo of a beach on Instagram*), the tools commonly require users to write "glue code" to either combine multiple rules or resort to other methods of programming (*e.g.,* in a Turing-complete programming language) through some escape hatch.

Code generation offered by large language models can serve to author this glue code for trigger-action programs, allowing for data from triggers to be mapped to input data for actions automatically even when their native data formats or intended functionality do not match exactly. However, such LLM-authored code, especially when implementing nontrivial logic, can be difficult to specify, understand or debug. Users need appropriate tools and handles to understand and make changes to the computation that is being performed in such code.

To tackle this challenge, we introduce *Ply*, a system that presents a core trigger-action programming paradigm but incorporates LLM code generation to enable a high degree of customization. Ply maintains the simplicity of a straightforward connection between a trigger and action but provides a structure within which users can enlist an LLM to specify the behavior of each trigger and action.

To assist with understandability and editability, Ply makes three concrete contributions:

- It encourages program decomposition into "layer" abstractions,
- It automatically creates visualizations of event payloads at layer boundaries to help users understand layer behavior without having to read the underlying generated code, and
- It constructs *ad hoc* parametrization interfaces that allow users to configure important dimensions of the behavior of each layer without having to re-author it.

Through the combination of these features, Ply allows users to develop, test, and tweak program components, exploring possibilities for how data can be transformed and composed to discover and achieve goals. This style of programming can support many use cases, even those not traditionally considered in the trigger-action programming model.

We demonstrate example programs that cover domains such as smart home alerts, multimodal budget tracking, and slideshow presentation assistance. Additionally, we report on a first-use study with seven participants, demonstrating how Ply's generative features can affect how users build programs and discussing how users handled errors and other unexpected behavior in Ply.

2 Related work

Ply builds on work in trigger-action programs, end-user programming, and LLM-supported synthesis of code and user interfaces.

2.1 Directly mapping triggers to actions

Interoperable standards can facilitate the communication needed to compose complex behavior between devices. "Tuplespace" platforms such as the Event Heap [29] provide for such communication through standardized I/O schemas, allowing for devices to subscribe to known event types. The Event Heap was the foundation of iStuff [8], a toolkit for authoring automation rules for ubiquitous computing environments such as smart rooms. The Xbox Adaptive Controller ecosystem [42] offers a pluggable interface that similarly allows for one-to-one correspondences between hardware devices and software actions.

Trigger-action programming systems, such as IFTTT (If This Then That) [28], include external integrations that allow the user to act as a "switchboard operator", choosing how high-level triggers from input integrations map to high-level actions. These systems serve as platforms for users to customize their environments [52], including smart homes [51], robot behavior [37], and DIY electronics [2].

HCI researchers have investigated how users understand [27] and debug [12, 19] trigger-action programs. Many techniques for improving users' understanding of their programs have been developed, including difference visualization [57], checks and simulations in EUDebug [17], and increasing expressivity, *e.g.,* by recommending rule compositions [18].

The straightforward one-to-one mappings suggested by high-level integration targets can be limited in customizability, however. For example, a button-press integration that is designed to be used in a one-to-one system may explicitly implement both a single-press and a double-press event so that the user can bind actions to either trigger. Alternatively, a system that allows custom code to mediate triggers and actions would not need an explicit double-press integration, since it could implement a double-press event using just a single-press event handler and timers; this would permit additional customization, *e.g.,* a triple-press trigger. In addition, this permits integrations to be simpler, offering a core set of composable features rather than a large set of specific features. Where systems like IFTTT support primarily commercial products [2], reducing the amount of labor required to create external integrations for a trigger-action system can improve the availability of integrations even for lower-resourced projects. Some trigger-action systems (including IFTTT) allow for small amounts of mediating code to be interposed between trigger and action, so that these programs resemble those developed in end-user programming contexts.

2.2 End-user programming

End-user programming has long been a means to enable users to specify precise behavior involving both hardware [7, 25] and software [11, 45]. Code in these systems takes many forms, such as block-based programming [14, 32, 41], node-graph representations [21, 31], and sometimes text-based programs [38]. These programming systems have seen widespread adoption, allowing users to program more specific correspondences between devices (as in Home Assistant [5], Node-RED [21]). However, designing complex behavior can be a difficult programming task, and program representations in end-user programming tools may not be well-suited for heavy programs.

Additionally, advances in promptable AI have encouraged deeper integration between these AI models and end-user programming systems. Intelligent sensor feeds have been in demand since before it was practical to automate them; Zensors [36] demonstrated a crowdsourcing-based approach that later inspired the AI-powered Gensors project [39]. Now, end-user programming techniques have begun to incorporate AI components, both in existing tools (reflected by integrations for Node-RED [44], Home Assistant [46], and Google Sheets [1, 3]) and new tools [50]. Through chains of AI components, more nuanced programs can be constructed [54].

2.3 LLM-supported code synthesis

Tools such as ChatGPT and Copilot have been studied extensively in how they assist users to author code [9, 20, 40]. Further work has explored interactions beyond simple back-and-forth chat for developing programs. For example, Zamfirescu-Pereira et al. [56] present Pail, a tool that allows users to explore complete program design through more abstract representations of the decision space.

Frequently, code-generation systems focus on building and then refining a full working application, using visibility of the full underlying code as a fallback when users need to build understanding of the generated program. Alternative approaches could encourage code to be built and solidified in stages, verifying individual components before moving onto the next part of an application.

Ply offers this LLM-supported program decomposition supported by visualization and parameterization UIs, permitting users to use interactions beyond chat to compose their programs incrementally. In Figure 2, we situate Ply among similar programming environments that support piecewise authorship and modification of programs. ComfyUI [47] offers user-controlled, node-based programming for AI data flows. CoLadder [55], like Ply, supports hierarchical decomposition of programs, but focuses on programs that use imperative code and linear data flows (compared to Ply's event publish-subscribe model), with data formats (*e.g.,* dataframes) that are straightforward to render intermediate outputs for. InstructPipe [58] provides users with fixed primitive nodes and uses LLMs to *compose* these nodes. Ply instead asks users to take charge of building their own node/layer graph, but uses LLMs to *create* layers that implement steps described by the user, offering supporting UIs for each new node. ChainForge [4], like Ply, mixes code- and AI-powered nodes in a programming interface. Code generation is available, but without visualized data transformations or real-time data flows. Generated programs are used to construct and evaluate prompts for prompt engineering, but the authored prompts (rather than the programs themselves) are the ultimate system output.

2.4 Dynamic and malleable user interfaces

Dynamic user interfaces can be key tools for accessibility and customization [22, 23]. LLM-based code generation has also inspired a recent refocusing on dynamic creation of user interfaces.

Dynamic widgets can assist in exploring and visualizing datasets, as highlighted by DynaVis [53]. Biscuit [15] further explores how

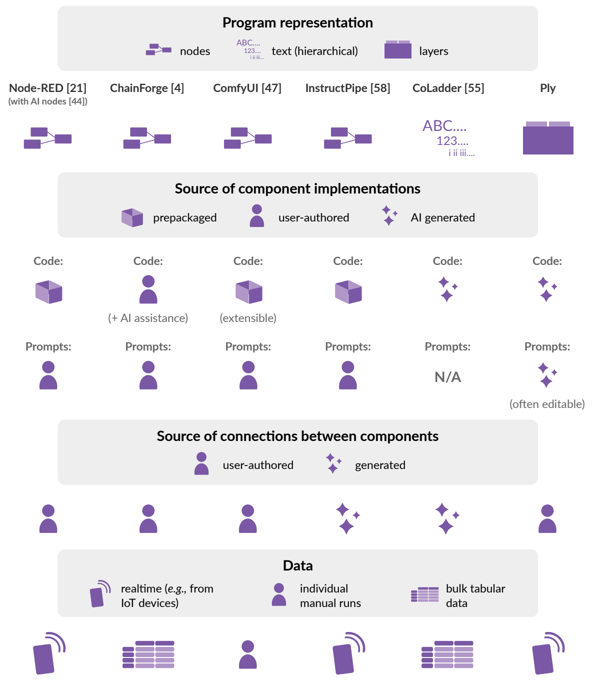


Figure 2: We situate Ply in typical use among other programming tools that support authoring and joining together program components. Most tools support both code-based and AI-based program components; when generative AI powers a component, prompts for these components may be supplied by the user or generated by the tool. Ply additionally generates components (layers) that incorporate both code execution and AI in the same component. Some tools generate links that connect components together (*e.g.,* by generating an entire node graph or program), where others depend on the user to compose programs out of individual components.

ephemeral UIs can allow users to explore varying choices for data transform they are interested in trying.

More recent work has looked into how user interfaces can be generated and customized to support specific tasks. Cao et al. [13] explore, through Jelly, how LLMs can generate data models that inform the construction of user interfaces relevant to particular tasks. Rather than relying entirely on user prompts to specify tasks, Jelly determines an abstraction over the task, defining a domain within which direct manipulation allows for fine-grained control over task execution.

Min et al. [43] explore the generation of both overview interfaces and detail interfaces through a system that customizes programs by extracting and structuring information relevant to a particular task. Users are able to choose and visualize exactly the information that is most helpful to see while working toward their goals.

Ply's approach embodies this genre of work through generation of parameterization datatypes and associated configuration UIs, which allow the user to configure the particulars of more abstract behavior, and through generation of glanceable interfaces that quickly communicate salient features of program data.

3 Ply User Experience

Ply's interface, shown in Figure 3, consists of a large infinite canvas on the left and a sidebar of components on the right. Users can bring in components from the sidebar, using the canvas to lay out and reorganize their work[1].

The sidebar includes a large set of built-in **sensors** and **actuators** that can be used in trigger-action programs.

Sensors in Ply are components that *provide information to Ply* through dispatched events with data payloads. The word "sensor" is used somewhat metaphorically; these components need not correspond to real-world physical sensors, but may refer to anything that dispatch events to provide information. Example sensors include: a sensor that informs Ply about button presses on a physical remote control, a sensor providing information about the presentation state of a slideshow, and a sensor that offers a live webcam feed.

Actuators, likewise, *receive* information from Ply to take external action (which may or may not correspond to physical action). Example actuators include: smart lightbulb or power outlet control, moving to a different slide in a slideshow, or displaying information on a screen.

Sensors and actuators may integrate into Ply from many sources, positioning Ply as a hub for user-controlled interoperability between a broad range of devices and software programs.

3.1 Linking sensors to actuators

Ply attempts to capture the elegance of straightforward trigger-action pairs while allowing users to invoke LLM code synthesis to customize the specific behavior of triggers and actions. To create a trigger-action program, users can create a **linkage** between one sensor, which produces the trigger, and one actuator, which carries out an action.

When creating a linkage, users write a natural language description of how they want the triggering sensor to affect the actuator. Linkages are themselves entities within Ply, tracking not only the sensor and actuator being linked but also details about exactly how the linkage should be carried out.

For example, consider a user who wishes to use buttons on a physical remote to toggle a light on and off and control its brightness. The user creates a linkage between the remote sensor and the light, shown in Figure 3A.

LINKAGE Remote Buttons → Lightbulb (with color)
"let me toggle the light and control brightness"

Ply implements this correspondence using LLM code synthesis.

Large refinements to the behavior of the linkage can be made through a **chat interface**, which can also answer questions about how the linkage works. For example, a user could ask for brightness control to work continuously while holding down the buttons, not just when clicking them.

Multiple linkages can be created within a Ply workspace, so that the workspace represents a collection of trigger-action pairs. Sensors and actuators can be used in multiple places and can therefore be instantiated multiple times on the canvas. Every instance of a particular sensor or actuator on the canvas shares the same event data as other instances of that same sensor or actuator. Linkages are instantiated exactly once on the canvas.

3.2 Linkage parameterization

When building a linkage, Ply identifies **parameters** of the implementation that may be tweaked to customize the behavior of the linkage. In this example, the generated linkage has two parameters: "Brightness Step" (how much the brightness changes with each button press) and "Hold Repeat Interval" (time between repeated actions when a button is held down). For many modifications, these parameters can be used in place of full chat-based refinement.

A **customization interface** is generated automatically for these configuration options, shown in a detail view modal for that linkage.

3.3 Sensor abstraction

3.3.1 *Building sensor layers.*

Ply provides users with tools to build components incrementally, creating new *layers* on top of existing components that "wrap" the behavior of underlying layers.

For example, suppose a user wants to use other buttons of the remote to control the lightbulb's color. First, the user creates a "color chooser" sensor, a layer that dispatches events representing different colors which will serve as an event source later. By clicking the "Create layered sensor" button on the canvas item for the sensor representing the remote's buttons, the user can build up some additional behavior:

SENSOR Rainbow Color Cycler
(layered on *Remote Buttons*)
"left and right buttons cycle through colors of the rainbow"

This creates a *new* sensor that is "layered on top" of the base sensor. This new sensor, which has been automatically titled "Rainbow Color Cycler", has its code implementation generated by an LLM. It will listen for events from the underlying button sensor and cycle through different colors, emitting events that each contain data corresponding to a color. The old underlying sensor, an immediate dependency of the new sensor, now appears as a layer rendered *behind* the new sensor in the Ply interface and can be brought back onto the canvas for further use.

3.3.2 *Sensor parameterizations.*

Like linkages, sensor layers also include a **parameterization**, used within the layer's implementation code, and a customization interface for these parameters. Parameters may be used to alter many aspects of a sensor's behavior. In our testing, Ply has generated parameterizations that allow users to:

- choose how sensor input is handled, like with button mappings or thresholds;
- customize sensor outputs (*e.g.,* which colors are provided by the "Rainbow Color Cycler");
- change constants used by code (*e.g.,* a debounce or polling timer); and
- customize AI prompts used by 'smart' layers.

3.3.3 *Chat refinement.*

Sensors, like linkages, can additionally be refined through a chat interface when parameterization is insufficient. For example, the user can write "actually, shuffle through the colors instead" so that the sensor will choose random colors instead of cycling in order.

[1]Ply's primary interactions are inspired by Infinite Craft: https://neal.fun/infinite-craft/

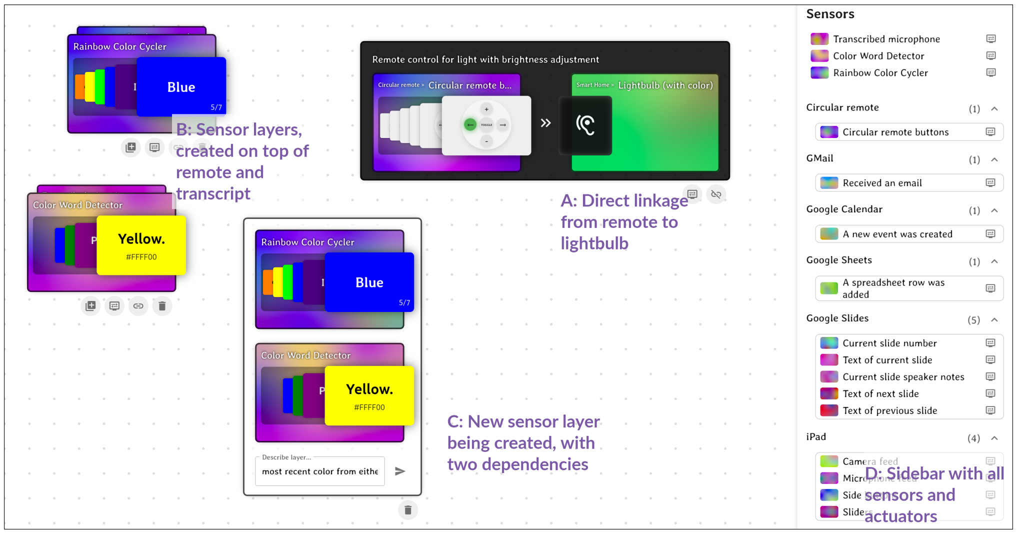

Rainbow Color Cycler
Blue
5/7
B: Sensor layers, created on top of remote and transcript
Color Word Detector
Yellow.
#FFFF00
Remote control for light with brightness adjustment
Circular remote › Circular remote b...
Smart Home › Lightbulb (with color)
A: Direct linkage from remote to lightbulb
Describe layer...
most recent color from eithe
C: New sensor layer being created, with two dependencies
Sensors
Transcribed microphone
Color Word Detector
Rainbow Color Cycler
Circular remote (1)
Circular remote buttons
GMail (1)
Received an email
Google Calendar (1)
A new event was created
Google Sheets (1)
A spreadsheet row was added
Google Slides (5)
Current slide number
Text of current slide
Current slide speaker notes
Text of next slide
Text of previous slide
iPad (4)
Camera feed
D: Sidebar with all sensors and actuators

Figure 3: A screenshot of the main Ply interface. On the left is the Ply canvas, where sensors, actuators, and linkages can be created, removed, and manipulated. A: A linkage is being created between a base sensor, Remote buttons, and a base actuator, Lightbulb (with color); B: Two sensors have been created already, and are on the canvas; C: The user is combining these two sensors into one larger sensor. On the right (D) is a list of all sensors and actuators that can be pulled onto the canvas, including both the external integrations and the new layers that have been introduced by the user.

Sensor layers, unlike linkages, choose their own output payload's datatype when created. Sensors will be updated in-place where possible, but updates that would change the structure of the output payload will instead fork and construct a new sensor, rather than risking an incompatibility with other sensors or linkages that use this sensor as a dependency.

3.3.4 Automatic decomposition. When building a new sensor layer, Ply will sometimes decompose the request into simpler sensors that are combined together automatically to produce the requested sensor. This feature tends to activate for prompts with more advanced processing, *e.g.*, "choose a random color, then also find its complementary color" — although there is some nondeterminism in when the system will break such a query down into multiple sensors. This allows Ply to leverage its existing layer features to offer intermediate visibility and control into complex computations, even when not explicitly requested by users. Intermediate dependencies are unobtrusively offered as additional components in the sidebar.

3.4 Sensor data visualization

Each sensor is accompanied by a glanceable **visualization** of the sensor's output payloads on the Ply canvas. This visualization is specific to the sensor and its output type, showing the most critical information for evaluating whether the sensor is behaving as expected. A detail view also shows an extended visualization, which is generated to show the full output payload of the sensor.

In our example, the user can now press the buttons that cycle through the sensor's colors, verifying this slice of behavior before using it to affect the light's color. This can be seen in Figure 3B.

3.5 Bundling sensors

When creating a sensor layer, the user can drag-and-drop multiple sensors together to create a layer that handles events from multiple dependencies. In Figure 3C, the user is creating a layered sensor that accepts a color either from spoken word (built on top of the "Transcribed Microphone" sensor) or from the remote's buttons:

> SENSOR Combined Color Sensor
> (layered on *Rainbow Color Cycler*, *Color Word Detector*)
> *"most recent color from either sensor"*

This sensor layer listens for events from any dependency, then dispatches event payloads of its own. Now, this sensor is a single, high-level trigger that represents a color from either source; it has its own visualization and parameterization. It can be tested independently of a trigger-action linkage, and each underlying layer can itself be isolated and tested by viewing its visualization.

3.6 Actuator abstraction

Layered actuators, mirroring layered sensors, transform input payloads so that appropriate events can be further dispatched to one or more downstream actuators. This layering can be used to present a simplified "façade" interface of a complex actuator; for example, a multicolor lamp may be layered so that it only lights up purple, presenting a simplified "on/off" input payload type rather than requiring a full color value every time. These layers can also be used to take multiple actions in response to one sensor event, by bundling two actuators together before inserting them into a linkage.

We have generally found during our own use of this feature that this layering is less intuitive to use than sensor layering, so we did not flesh out this feature or emphasize it in our first-use study. Equivalent behavior can often be achieved just by layering sensors

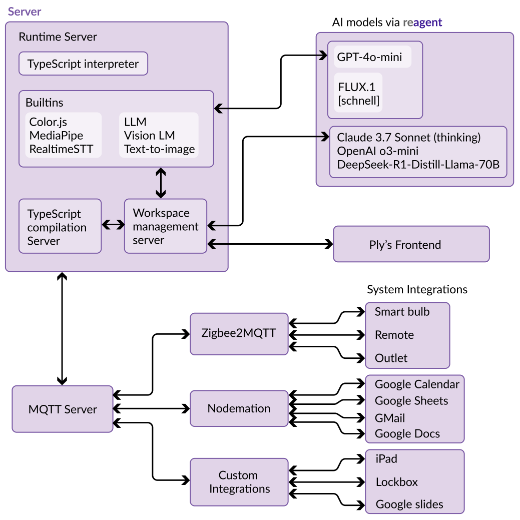


Figure 4: Ply's system architecture. The runtime server executes the implementation code for sensors, actuators, and linkages, coordinating with a central MQTT server that exchanges messages between the server and external integrations. AI models are used both in layer creation and in the builtin code that can be used by a layer.

and (if necessary) using more than one linkage on the Ply canvas, leaving the linkage to perform the final work of transforming the sensor's output payload into an actuator input payload.

3.7 Layer-based abstraction

Each layer in Ply tracks its dependencies; sensors receive data *from* their dependencies, actuators push data *to* their dependencies, and linkages each refer to exactly one sensor and one actuator dependency. Collections of layers and linkages in Ply are isomorphic to *node graphs* in node-based programming languages.

However, Ply presents layers differently than typical node-based tools, which show computation as a flow that transmits data from node to node along edges. When a layer is on Ply's canvas, its dependencies are not directly shown; users may reintroduce these dependencies to the canvas manually, but layers are otherwise drawn in a way meant to signal that they are self-contained. This design draws inspiration from abstraction layers in system design, *e.g.,* the OSI model for network design; once lower-level layers have been comfortably tested and finalized, users can use these layers as black boxes without thinking about the details of their implementations. The final act of linking a sensor to an actuator is much like drawing an edge in a node-based tool, except that the user builds up and organizes each linkage so that *one* representative trigger is linked to *one* representative action.

4 Technical implementation

Ply consists of a user interface for creating and modifying trigger-action programs and a server that coordinates communication among system components. The full system architecture is diagrammed in Figure 4.

4.1 Code generation

Ply uses a server program written in TypeScript to make code generation requests to a large language model and to execute the resulting code, which passes messages to and from sensors and actuators. Layers in Ply include a large amount of structured information (see Figure 5 for the metadata stored in a sensor), some in natural language and some in code. By ensuring that this rich metadata is populated for each new linkage or layer created, Ply enables further generation of working glue code, visualizations, and parameter editing interfaces.

4.1.1 Provided context. When Ply creates new layers or linkages, prompts include the full documentation of dependency sensors and/or actuators. The LLM is prompted to generate a TypeScript function implementing the requested behavior.

Linkages can subscribe to their sensor dependency and publish to their actuator. Sensors can subscribe to any of their dependency sensors and publish to their output, and actuators correspondingly subscribe to their input and publish to any of their dependency actuators. Linkages, sensors, and actuators may also store persistent data between code invocations.

In addition, linkages and sensor/actuator layers are permitted to use "builtins" that implement commonly-needed functionality. These builtins are documented in the prompt. Ply includes some "core" builtins in every request:

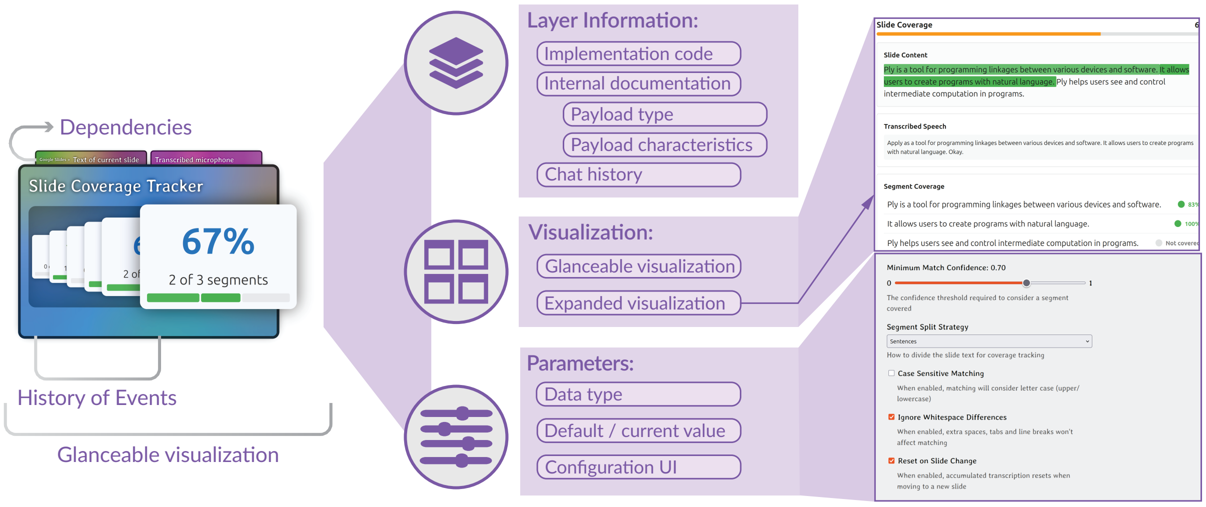
- An LLM function, which takes a text prompt and returns a string response (useful for text processing and basic world knowledge); this was implemented using `gpt-4o-mini` for its fast response time.
- A vision-language model function, also using `gpt-4o-mini`, which takes in a text prompt and a single image, and outputs text.
- The `underscore.js` library, with a particular focus in documentation on `debounce` and `throttle` functions, commonly used to avoid processing events in quick succession.

Depending on the specific user request, Ply additionally selects (using a quick `gpt-4o-mini` invocation) other libraries from our expanded set of builtins that may be useful; chosen libraries have documentation included in the main code generation prompt. In our prototype, these additional builtins are:

- Text-to-image generation using the *FLUX.1 [schnell]* model [35]
- Color manipulation using *Color.js* [16]
- Real-time audio transcription using *Whisper*, through *RealtimeSTT* [10]
- Real-time hand gesture detection using *MediaPipe* [24]

In a safe, sandboxed environment, Ply could even be permitted to import libraries directly from a package manager repository, enabling a much wider range of capabilities. Because Ply is grounded in code synthesis, its capabilities grow with its underlying code generation system.

These builtins enable Ply to invoke AI models not only at the time of linkage or layer creation, but also at runtime as the system processes events. For example, a sensor that uses a camera as a dependency can use a vision-language model to identify objects in the camera's field of view by generating code that prompts the model at runtime with the input image, an appropriate text prompt,



Slide Coverage

Slide Content

Ply is a tool for programming linkages between various devices and software. It allows users to create programs with natural language. Ply helps users see and control intermediate computation in programs.

Transcribed Speech

Apply as a tool for programming linkages between various devices and software. It allows users to create programs with natural language. Okay.

Segment Coverage

Ply is a tool for programming linkages between various devices and software. 83%

It allows users to create programs with natural language. 100%

Ply helps users see and control intermediate computation in programs. Not covered

Minimum Match Confidence: 0.70

0 1

The confidence threshold required to consider a segment covered

Segment Split Strategy

Sentences

How to divide the slide text for coverage tracking

☐ Case Sensitive Matching

When enabled, matching will consider letter case (upper/lowercase)

☑ Ignore Whitespace Differences

When enabled, extra spaces, tabs and line breaks won't affect matching

☑ Reset on Slide Change

When enabled, accumulated transcription resets when moving to a new slide

Figure 5: The anatomy of a sensor. Sensor layers in Ply contain rich metadata, used both to assist further code generation and to provide user visibility and control.

and a description of the desired output format. Often, prompts to language models are included by Ply as parameters for the user to customize, which offers direct control over these flexible, language-based components of programs.

Ply also includes recent event payloads alongside sensor documentation, allowing users to ground requests in actual data that they have seen a dependency sensor include in an event payload.

Model responses for new layers include additional natural language documentation describing the generated layer so that linkages or layers created later have sufficient context to use the new layer. The full set of metadata stored in a sensor layer can be seen in Figure 5.

Each sensor layer is accompanied by an output datatype specified in TypeScript, chosen by the code generation step. Generated natural language documentation can also help to elucidate semantic information about sensor output formats, beyond what simple datatypes communicate (*e.g.*, specifying "a color name" rather than just `string`). Further layers will be generated *without* access to the direct code implementation of dependencies, which promotes interoperability; this architecture would support a broader ecosystem of layers that are implemented even in different languages or even by non-programmatic means (*e.g.*, crowdsourcing, as in [36], or user intervention).

4.1.2 *Generating visualizations.*

Ply generates glanceable visualizations for sensor payloads, authoring React code based on the sensor documentation. React components are populated in the Ply canvas with real-time data from sensors, showing event payloads with custom visuals rather than simply showing the raw data associated with each event. Some information about the payload structure is thus abstracted away from the user, who can subsequently prompt for new programs in terms of the high-level information presented by the sensor, rather than referencing specific components of output JSON objects.

Since these glanceable components do not always show all of the information from the payload in the available space, Ply also generates "extended" visualizations that appear in the sensor's detail view.

4.1.3 *Choosing and configuring parameters.*

When creating linkages and sensors, Ply instructs the code generation LLM to choose and integrate a structured set of configurable parameters that alter the code's behavior. Parameters may come explicitly from the prompt (*e.g.*, "translate the input text to English" may have a *targetLanguage* parameter) or be implicit (*e.g.*, a choice of a *literal* or *interpretive* translation). These may be primitive types (like numeric, string, and Boolean values), or they may be more complex structured values (like color values, lookup tables, and lists). The LLM also provides a set of default values for this configuration. Then, Ply creates a React component that allows the user to configure these parameters interactively.

Parameter UIs are generated using an LLM prompt shared between linkages and sensors. This prompt requests a standalone React component that can be used to update the linkage or sensor's parameters object. These configuration UIs may use complex structure (*e.g.*, for a lookup table) and rich built-in browser components (*e.g.*, a color picker) so that even similarly complex parameter shapes can be customized.

4.1.4 *Refinement.*

When users send a chat message in an existing linkage or sensor's detail view, the original prompt is included, alongside a subsequent prompt indicating that the system should either respond with a follow-up message (*e.g.*, to ask for more detail or to warn the user that a requested change may not be possible to implement), or with a patch to the previously-created linkage or sensor. The prompt may include new builtin documentation, if necessary.

When a chat message results in a change, the LLM is asked to gauge whether the update would cause an incompatibility with the existing linkage or sensor, both for any internally-stored data and (in the case of sensors) for the output payload type. Internal incompatibilities are resolved by clearing the internal data state of the linkage or sensor. If a sensor would need to change its output payload type, the sensor is instead cloned for the new behavior.

4.2 Payload characteristics

In addition to the expected datatypes of output and input payloads, we identified three payload *characteristics* that are domain-agnostic and often relevant for constructed layers to include in their documentation. LLM-generated layers declare each of these characteristics explicitly, and their generated documentation may provide additional context in natural language when choices are not clear-cut (*e.g.,* when the *retention* of a message can differ depending on the message). Here we discuss these characteristics primarily with respect to generated sensors and their output payloads, but all three also apply in reverse form to actuators' input payloads.

4.2.1 Event timing. Sensors can provide either continuous data (*e.g.,* from a camera or other readout) or discrete data (*e.g.,* from button presses or received email messages). Further, discrete events may signal one-off occurrences with a low duration of relevance (*e.g.,* when a button press is used to take an action, like advance the slide in a slideshow), or they may be accompanied by data that is relevant even after some time has passed, and therefore should be *retained* for future use (*e.g.,* when a radio button is used to choose a mode that affects future computations). Ply asks generated layers to choose which of these three options ('stream', 'event', or 'event with retention') is most suited to describe how the layer's payloads should be used. Our choice to delineate explicitly between discrete and continuous sensor data reflects a core takeaway from prior work in sensor-based programming [26, 48].

This payload characteristic is exposed indirectly to the user in sensor visualizations. On the main Ply canvas, sensors that use the 'event' or 'event retained' characteristic have their output data visualized using discrete cards, each representing a separately-published event. LLM-generated UI components that render these events are contained to the area of the card. Sensors using the 'stream' characteristic are visualized using a more freeform panel, rendering an LLM-generated UI component that can choose exactly how to show the latest stream information, which may update too quickly for discrete cards to be practical.

4.2.2 Update mode. Sensors should identify whether their emitted data represents a *replacement* of previously-emitted data or an *update* of prior data. 'Replace' events tend to be appropriate for absolute-value sensors, whether continuous or discrete. 'Update' events reflect deltas from previous events, *e.g.,* representing just the new text appended to a transcript.

Distinguishing between these two modes is important when processing dependencies. For example, consider two sensors that use some text as a dependency and emit an output payload containing the summary of the input text. The first sensor uses a dependency that emits the full contents of the current slide in a slideshow presentation, updating when the slide is changed. The second sensor instead uses a dependency representing a running transcript of a connected microphone, which may send a few words at a time, splitting sentences across multiple event dispatches. The technique used to summarize the dependency's text will need to be different between the first sensor (which handles 'replace'-style events) and the second sensor (which handles 'update'-style events).

4.2.3 Staleness. Some sensors will take time to process input events, like when waiting for a response from a vision-language model to interpret an input image. Such sensors will need to decide how to handle inputs that come in while a previous event is processing.

A common pitfall is to allow asynchronous processing to resolve out of order. Consider a sensor that receives "current slide text" events from the user's slideshow software and uses an LLM to translate the current slide to a different language. If this code is not authored carefully, a long translation request (combined with a quick slide change) may result in a later slide being translated before an earlier slide. When the earlier slide's translation is ready, the "translated slide" sensor would dispatch a new event, overriding the later slide, which is likely to be undesirable.

Ply prompts the LLM performing layer generation to choose an explicit strategy for handling stale requests. A few approaches are viable, such as canceling outdated events, using a queue to enforce exhaustive in-order execution, or simply allowing out-of-order execution. By explicitly describing this issue and prompting the LLM to articulate (in natural language) a strategy for handling staleness in each layer's processing code, Ply allows downstream layer generation to take this behavior into account. In addition, we noticed that including this instruction in the prompt generally improved the system's reliability in generating layers that handle staleness in a way that is logical for the sensor.

4.3 External integrations

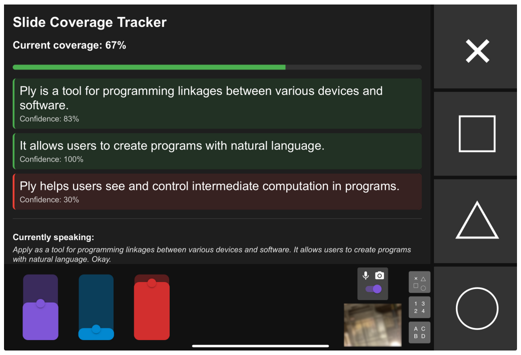
Ply coordinates communication with external services through JSON-encoded MQTT messages published on topics unique to each sensor and actuator. In our prototype, external integrations are all configured using metadata hardcoded into Ply; there is no discovery mechanism for new integrations to be added at runtime. These integrations exchange messages with the MQTT server, and we handwrote natural language documentation (stored in each sensor and actuator's metadata) describing what the integrations do and the structure of event payloads that they dispatch or receive.

4.3.1 Existing ecosystem integrations. Because MQTT is a common protocol for communication between devices, we were able to include a number of integrations without authoring new code.

We integrated an LED strip using the WLED firmware[2], which directly supports MQTT/JSON control, with only a natural language description in the system of how API messages should be formatted. Many integrations are available in the Zigbee2MQTT project, a community-supported ecosystem of MQTT bindings for smart home devices[30]; we integrated a Zigbee remote control, toggleable smart outlet, and a Philips Hue color lightbulb.

We introduced integrations with software components using a self-hosted instance of Nodemation [45], a low-code/no-code software workflow automation platform; in particular, we used Nodemation to create input and output bindings for Google Sheets, Google Calendar, GMail, and Google Docs. Nodemation is a programming system in its own right, but we created very simple

[2]https://kno.wled.ge/interfaces/mqtt/


Slide Coverage Tracker

Current coverage: 67%

Ply is a tool for programming linkages between various devices and software.
Confidence: 83%

It allows users to create programs with natural language.
Confidence: 100%

Ply helps users see and control intermediate computation in programs.
Confidence: 30%

Currently speaking:
Apply as a tool for programming linkages between various devices and software. It allows users to create programs with natural language. Okay.

Figure 6: A screenshot of the tablet interface we created for integration with Ply. The interface includes a large output panel, in this case showing a “slide coverage tracker”, and software controls for sensor inputs.

two- or three-node graphs for use with Ply, binding MQTT publications/subscriptions directly to these underlying APIs by passing data straight through the node graphs.

4.3.2 Additional integrations. We developed a real-time Google Slides integration, implemented as a browser extension, that provides information to Ply about a presentation currently being presented. The integration also includes a “next slide” and a “previous slide” actuator. While Nodemation is able to provide some control over software, these API-based integrations tend to work at the document management level (*e.g.,* “Create a new slideshow”) rather than in a way that could enable users to customize the interface they use to *control* the software. Thus, our custom Slides integration allowed us to explore the creation of more interactive software customizations through Ply.

To add some baseline software-based input controls and output capability, we also developed an iPad integration, including four sensors and one actuator (see Figure 6): software buttons; analog sliders; microphone and camera feeds; and an actuator called “Display panel”, which is a large display area that accepts React code in event payloads and renders it in-place.

Interfaces rendered in the display panel cannot directly dispatch events into Ply, although this UI may be locally interactive through *e.g.,* scrolling or collapsible sections. Interactive seemingly-standalone “apps” can be created using the iPad integration’s sensors and the display panel, routing actions through Ply’s backend.

4.3.3 Integrating flexibly. As Ply generates code, each prompt to the language model includes the documentation of dependency sensors and/or actuators. Although Ply depends on the presence of integration code for each device or software component that synchronizes state through JSON-encoded MQTT events, there are no further restrictions on the precise format or timing of event payloads. This is why we were able to use existing MQTT integrations within Ply. For example, the Philips Hue light color is controlled via `hue` and `saturation` values in the JSON object, while WLED requires RGB values to be included in a nested sub-object; rather than authoring shim code that presents the same standard interface for both lights, we simply document each device’s preferred format and allow the system to author the “last-mile” code that transforms the data to the format consumed by the integration.

4.3.4 External integrations as a building block. Through flexibly-defined integrations with other systems, Ply allows users to combine composable workflow components with one another.

On the trigger side, for example, Ply can react to new messages in an email inbox or rows added to a Google Sheets spreadsheet. Thus, any data that can be received through an email notification or a submission to a custom form can trigger behavior in Ply. Because Ply can invoke AI models to process incoming language data, these integrations need not be configured with a high degree of precision; users can choose any reasonable form field structure or receive emails in many different formats, and data can still be extracted accurately.

On the action side, Ply has also been equipped with the ability to write rows *into* a spreadsheet, closing an interoperability loop with this end-user-programmable software. We also created a physical integration with a small lockable box that we built using a microcontroller and stepper motor, so that trigger-action programs can lock and unlock the box. This illustrates that the target behavior of a Ply program may itself be a means to some further goal, like providing access to a physical key.

Therefore, Ply does not need to seize end-to-end control of a workflow to offer value. Emails, spreadsheets, and lockboxes are representative components that are *already* used in everyday life to compose more complex workflows — whether to ingest information, process data, or provide access to physical objects – and Ply can help users “glue” these pieces together, even when those pieces do not fit together natively.

4.4 Implementation choices

Our current version of Ply uses Anthropic’s Claude 3.7 Sonnet language model, with 1200-3600 tokens of “thinking” budget for the various task prompts, when various features are invoked (primarily layer generation, visualization generation, and parameterization component generation). Ply can also be configured to use OpenAI’s o3-mini model, at medium and high reasoning levels, and has also been tested with a distillation of the DeepSeek-R1 model (based on the 70B Llama model).

Our initial prototype, built with o3-mini, takes around 30 seconds to create a typical linkage or layer. The distilled R1 backend offered much faster generation time (3-5 seconds), but responses often resulted in parsing errors. We ultimately used Claude 3.7 Sonnet for our prototype, despite slower generation time (~45 seconds), because of its high-quality visualizations and rich parameterizations. It may be possible to bring model performance into closer parity through more precisely-tailored model-specific system prompts. We generally tried to keep LLM prompts simple, preferring to avoid prescribing behavior too narrowly in ways that could harm either generalizability across domains or generalizability between underlying models (to ensure that Ply remains well-positioned to take advantage of future developments in AI code generation capability). LLM and text-to-image model calls from Ply are sent through REAGENT, a server platform (introduced in [6]) that centralizes API calls for different backing models.

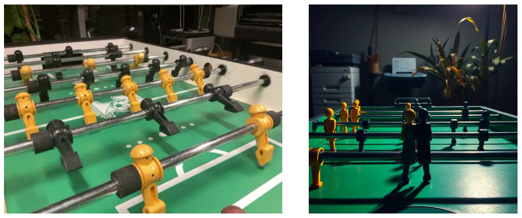


Figure 7: An input and output image for the "moody" version of the AI digital camera created in Section 5.1. The program created within Ply describes, dramatizes, then re-draws the image.

5 Example programs

We offer example programs that we have successfully authored in Ply, alongside details of how we built up the programs. In describing these programs here in the text, we have occasionally changed the names of integrated and generated sensors/actuators for clarity. The described prompts are exact, although we did occasionally need to retry actions due to errors in responses (such as with the *iPad display* actuator, which sometimes results in linkage code with syntax errors in our prototype).

5.1 AI digital camera

Here, we create an "AI digital camera", using a chain of sensor layers to build toward a program that dramatizes and redraws photographs, as in Figure 7.

Since the iPad streams a live feed of its camera, we first create a sensor that allows us to take a photo by pressing a button:

Sensor Take a Photo
(layered on *Camera Feed*, *iPad Buttons*)
"when I press the X button, take a photo"

Then, we add a layer that describes what's in the photo:

Sensor Photo Description
(layered on *Take a Photo*)
"describe what's in the photo"

Now, we can dramatize the description and make a painting based on the dramatization:

Sensor Photo Description Enhancer
(layered on *Photo Description*)
"make the description sound more dramatic, heroic"

Sensor Photo Description Painter
(layered on *Description Enhancer*)
"make a painting of this"

Linkage Photo Description Painter ⟶ iPad display
(no prompt)

When the user presses the X button, a photo is taken, described, dramatized, and re-painted onto the iPad display.

At each layer, the user can see the data in its transformed state. For sensors whose payloads contain a lot of text, the text is often cut off in the glanceable visualizations; the extended visualization in the sensor's detail view shows the complete description. After creating the final linkage, this "camera" can be used outside of Ply.

The user can still make changes. Suppose we want the description to become not heroic, but instead moody and brooding. In this case, Ply generated parameters that allow us to configure the LLM prompt used to improve the description, so there is no need to use the chat interface to refine the behavior.

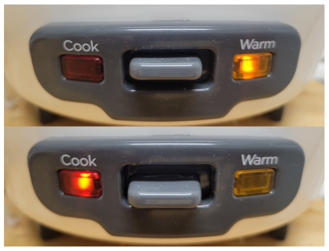


Figure 8: A rice cooker whose current state is identifiable using a vision language model.

5.2 Rice cooker timer

Here, we use Ply to create a sensor layer representing the status of a rice cooker (cooking, warming, or off – see Figure 8). Since this is not a "smart" rice cooker, we affix a camera pointing at the front of the rice cooker, then use AI image processing to interpret the light-up interface.

Sensor Rice Cooker Status
(layered on *Camera Feed*)
"every 30 seconds, tell me if the rice cooker is: cooking, warming, or off. the left red light means cooking, right yellow light means warm, if neither then it's off."

Once we have confirmed that the sensor correctly reads the rice cooker's status, we can layer on additional functionality:

Sensor Rice Cooker Timer
(layered on *Rice Cooker Status*)
"start a timer once the rice cooker starts cooking. reset the timer and let me know when it changes to warming"

This sensor dispatches a new "time-elapsed" event every second (configurable with parameters) while the timer is on, then a special "finished" event when the timer switches to "warm". We can keep track of the status from our office:

Linkage Rice Cooker Timer ⟶ Office light
"when it's cooking, make the light yellow and get brighter over the course of 25 minutes. once it finishes, change to green"

Through Ply, we improvise a sensor with equivalent functionality to a smart appliance, building a trigger-action program that deals with higher-order information about the rice cooker's status even though the trigger is ultimately based on the camera feed.

5.3 Transaction tracking

In this example (also demonstrated in Figure 1), we create a program that helps us track our spending. This workspace uses just one linkage, using a sensor that represents a compilation of multiple underlying transaction sensors and an actuator that takes multiple actions in response to a transaction. We assume the presence of a "Take a Photo" sensor like the one created in Section 5.1.

SENSOR Photo Receipt Reader
(layered on *Take a Photo*)
"if there's a receipt, get the vendor and $$ amount"

SENSOR Email Receipt Extractor
(layered on *On Email Received*)
"if i get a digital receipt, get the vendor and $$ amount"

SENSOR Transaction Detector
(layered on *Photo Receipt Reader*, *Email Receipt Extractor*)
"detect transactions"

ACTUATOR Add Budget Row to Spreadsheet
(layered on *Insert Row into Spreadsheet*)
"add budget row to spreadsheet with Vendor and Amount columns"

ACTUATOR LED Strip Progress Bar
(layered on *WLED Strip*)
"create an LED strip progress bar"

ACTUATOR Transaction Tracker
(layered on *Add Budget Row to Spreadsheet*, *LED Strip Progress Bar*)
"Track my transactions"

LINKAGE Transaction Detector –> Transaction Tracker
(no prompt)

Although this program consists of only one trigger-action pair, both sides of the linkage are rich with behavior, and the program cleanly maps a complex trigger to a complex action using one core linkage.

5.4 Slideshow progress

This program uses a microphone transcription during a slideshow presentation to track how much of the current slide has been covered so far, providing both a visual guide to the presenter on the iPad display (shown in Figure 6) and also an automatic slide advancement feature once the slide has been fully covered.

SENSOR Slideshow Coverage
(layered on *Transcribed Microphone*, *Text of Current Slide*)
"track which parts of the slide I've said already"

LINKAGE Slideshow Coverage ⟶ iPad display
"show me what I have and haven't covered"

LINKAGE Slideshow Coverage ⟶ Go to Next Slide
"go to the next slide when I finish this one"

The generated sensor (expanded in Figure 5) keeps a buffer of transcribed speech that it resets when the slide changes. As generated, it splits slide text into segments and matches against the transcript using basic string processing operations, with a configurable match threshold, although the sensor can also be modified to use AI for more approximate semantic matching. In the sensor's glanceable visualization on the Ply canvas, users can see how much of their slide has been covered. The full event payload is larger, though, as revealed by the expanded visualization in the detail view.

Here, we use the created sensor in two distinct linkages, each serving a different purpose. By reusing the sensor, we can ensure that the slide change is using the same (customizable) coverage tracking logic that is shown on the display.

5.5 Stateful lockbox

In this example, we illustrate a trigger-action program, authored in Ply, that allows a box to be locked and unlocked. This program has two modes: a "standard" mode which unlocks a box when the camera sees an open hand gesture and locks the box when the camera sees a closed fist, and a "locked-down" mode, activated through a thumbs-down gesture, which transitions back into the "standard" mode only when the user successfully enters a four-digit passcode. The current mode is communicated through a colored light. This program is stateful, including both the PIN entry state and the current mode. Algorithm 1 offers pseudocode for this program.

Algorithm 1 Gesture and PIN-based Lock System

```
mode ← standard
entered_pin ← ‘ ’
function onGestureDetected(gesture)
    if mode = standard then
        if gesture = open-hand then
            unlock box
        else if gesture = closed-fist then
            lock box
        else if gesture = thumbs-down then
            mode ← locked-down
            set light color to red
        end if
    end if
end function
function onNumericButtonPressed(button)
    entered_pin ← entered_pin + button
end function
function onCircleButtonPressed
    if entered_pin = ‘1234’ then
        mode ← standard
        set light color to green
    else
        entered_pin ← ‘ ’
    end if
end function
```

By decomposing this task into layers, the user can visualize the program's intermediate state within Ply, with each layer managing its own slice of state. Here is how we constructed this program in Ply:

SENSOR Pin Detector
(layered on *Buttons*)
"When I type 1234 and hit O, unlock, otherwise O should clear pin"

Sensor Gesture Detector
(layered on *Camera feed*)
"Look for open hand, closed fist, or thumbs down"

Sensor Box Mode
(layered on *Pin Detector, Gesture Detector*)
"When I give a thumbs down, go into lockdown mode until I type the right pin"

Sensor Box Unlock
(layered on *Box Mode, Gesture Detector*)
"Lock and unlock with a closed fist or open hand, but only if box is not in lockdown mode"

Linkage Box Unlock → Lockbox
(no prompt)

Linkage Box Mode → Lightbulb (with color)
"red for lockdown mode, green otherwise"

Even this highly stateful program can be built easily in Ply, with each linkage presenting a different portion of state to the user. In this example, Ply provides a number of parameters, including the ability to change the unlock PIN and also an option to flash the red light during lockdown mode.

When we first generated this example in Ply, the program did initially contain a bug: after correctly entering the PIN, the PIN variable would not clear, and the subsequent entry would fail (thus resetting correctly for the next entry). Because of the layer decomposition and data visualizations, it was easy to see which layer contained the bug ("Pin Detector", which did not correctly unlock the second time). When the buggy behavior is invoked, the glanceable visualization also provided insight: by showing a number of bullet points equal to the number of keys entered, the sensor showed that its value had not reset. We fixed this bug by refining this sensor with the chat message "*it unlocks the first time but then the next one fails before it works again*", which triggered a reasoning response from the model (*"Looking at my code: ... I see the problem! When the PIN is correct and we unlock the system, we never clear the current PIN!"*) that resulted in the correct repair.

5.6 Reusing layer components

In addition to supporting deep chains of logic (as in Section 5.1), Ply's reusable layers offer potential for "wide" computation graphs. One Ply canvas can support an ecosystem of interoperating devices using shared layer abstractions, for example in a single Ply-powered smart home. We built these examples in Ply, demonstrating that shared layers can be powerful ways to derive higher-level behavior to be used in multiple places.

5.6.1 Hotword detector. On top of a "transcribed microphone" sensor for a centrally-placed microphone, a *hotword detection sensor* detects the words "Hey computer", followed by a command; then, many different devices (*e.g.,* lights, outlets, electric window blinds) can be controlled using this detector, and the hotword trigger behavior can be modified later for all devices at once.

5.6.2 Porch camera. A camera pointed outside the home's front door has two sensor layers built on top of it: one interprets local weather conditions, and another periodically checks to see whether a package is on the doorstep. These sensors offer different information, higher-level than the raw camera feed each is based on, to be used in further automations. For example, the weather sensor adjusts the brightness of interior lights to compensate for dull weather, while the package detector sends a notification to the homeowner when a new package arrives. Later, the sensors are recombined, notifying the next-door neighbors to ask for help when a package is at risk of being stuck out in the rain.

5.6.3 Ambient alerts. An actuator layer coordinates multiple data sources through a lightbulb in the user's study, providing passive status information about the home. The actuator pools incoming alert data; when no relevant information is available, the bulb defaults to a warm, soft light. As the actuator collects data requiring the user's attention (*e.g.,* incoming emails or a package on the doorstep), the bulb increases in intensity, slowly turning to a cooler color and eventually blinking until the user notices and clears these alerts. More critical data sources (*e.g.,* an email deemed to be urgent) can publish events to the actuator that quickly elevate the actuator's alert status, and new data sources can be added by linking them to the existing alert system.

These three examples coexist in one Ply project representing the smart home, and interconnections between these abstractions can be built just as if they were low-level primitive components.

6 User study

We conducted a first-use study with seven participants to observe how they construct trigger-action programs with Ply.

6.1 Study design

We recruited participants for an in-person two-hour lab study from university mailing lists for various disciplines. All participants were 18-22 years old (four male and three female, self-reported in a freeform field), and all reported some programming experience. Participants were compensated $40 for their participation.

6.1.1 Training session. We began each visit with a training session, consuming around one hour. We directed participants to walk through a set of basic tasks involving constructing sensor layers and connecting them to actuators through linkages, using various external integrations. Participants made sensor layers with multiple dependencies. They also modified sensors and linkages, both through parameters and through chat refinement. The "AI digital camera" described in Section 5.1 reflects one chain of sensors that participants created during the training session. As described earlier, we did not emphasize the actuator abstraction feature during the study, and no participant used this feature. To help participants understand the programming model, the training session included a usage example that involves Ply choosing a random color when a button is pressed, to be used in multiple actions. This *requires* a layered sensor to be created; we demonstrated that simply creating a direct sensor-actuator linkage without an intermediate layer would result in the linkage containing hidden state that could not be used by another linkage's execution.

6.1.2 *Study tasks.*
Participants were then presented with a fresh workspace and asked to complete four tasks. We allowed participants to take their time with the tasks in order, but in some cases, we asked participants to move onto the next task without completing a prior task, to ensure timely completion.

6.1.3 *Task: Pokémon card database.*
In the first task, participants used the iPad camera to create a digital record of a deck of Pokémon cards, *e.g.,* by processing photographs and inserting records into a spreadsheet. This is similar to the example described in Section 5.3, although it is smaller in scope.

We also revealed a second portion of the task, in which the total accumulated HP (hitpoints) values of the photographed Pokémon cards would control the brightness of a lamp. This subtask is best completed through reusing a sensor layer, but can also be completed by re-creating the data extraction logic.

6.1.4 *Task: Slang translator.*
Then, participants were asked to use the *Transcribed Microphone* sensor (which is a sensor we pre-loaded into the workspace, built on top of the raw microphone feed using the transcription builtin) to detect slang in speech, and to use the iPad display to provide assistance to a hypothetical person in the room who might have trouble understanding the slang (*e.g.,* by showing definitions or an altered transcript).

As a second portion of the task, we again asked participants to control the brightness of a lamp, this time based on the total amount of slang used in the conversation.

6.1.5 *Task: Slide clicker.*
Participants were then asked to build a workspace that allows users to use both buttons and voice control to move forward and backward in a slide presentation, and also to use a document or spreadsheet to keep track of when the current slide changes.

6.1.6 *Task: Lockbox.*
Finally, participants were asked to use at least two distinct sensors to build some kind of puzzle or challenge that could be used as the condition to unlock a physical lockbox.

6.2 Results

Participants were generally successful in completing the provided tasks, and found the tasks enjoyable; on a five-point Likert scale, five participants agreed and two strongly agreed with the statement "I enjoyed using [Ply]".

Not all participants made ready use of sensor layering during their one-hour task completion period. One of seven participants chose to complete tasks only using linkages between base-level sensors and actuators, which often resulted in duplicated logic between linkages (*e.g.,* to determine the HP value of a Pokémon card in the first task). When users are not comfortable decomposing their programs into smaller component layers, Ply can still provide value through natural language specification and refinement of trigger-action programs and through the parameterization interface (both of which were used by the participant who did not create new layers).

All but one of the participants who did make sensor layers *reused* these layers at least once, breaking up their computations into individual components that could be tweaked and verified individually.

Participants were not wholly effective at decomposing their code into reusable layers. In the slide clicker task, for example, only P6 attempted to combine voice-based and button-based actions into one slide-change sensor layer, and they ultimately still made separate voice- and button-based linkages when their first attempt didn't work. P3 explicitly called out feeling that their program structure wasn't ideal in the slide clicker task: *"I feel like there's definitely a way I could be doing this. I feel like I'm being super redundant by making, like, separate ones."* When programs were decomposed well, however, the sensor visualizations and parameterizations proved helpful.

6.2.1 *Using sensor visualizations.*
Sensor visualizations served multiple purposes during task completions. A common use was to verify a sensor's behavior before linking it; for example, all five participants who created a specific "Pokémon card detector" layer abstraction took a photo and checked the layer's output before building on top of it with another layer or linkage.

Even after linking a sensor to an actuator, however, sensor visualizations proved to be a useful way to see the intermediate state of a program and to see what information was available to the linkage. For example, P1 discovered an issue with their Pokémon card detector (which would not detect a card if multiple were present) just through a distinct "No card detected" sensor visualization, before needing to check explicitly whether rows were being added to the target spreadsheet.

Visualizations were even useful in verifying end-to-end behavior. While building their slide clicker, P3 used the built-in "Current slide number" *sensor* and its associated visualization to verify that the slide was being advanced correctly in response to their actions, rather than checking the slideshow manually. By providing real-time feedback about the status of integrated elements, Ply supports end-to-end visibility of executions in the same medium as visibility of internal state, combating "information" and "understanding" learning barriers (as in [34]) together.

6.2.2 *Dealing with errors.*
Errors in Ply can emerge either when generated code does not run correctly within the system (generally causing an explicit "Error" response in the UI) or when code does not behave as a user requested or intended. For example: in the "AI camera" task of the training session, which all participants completed identically, there were 3 instances of non-parsing code and 5 instances of layers behaving differently than users intended (and needing refinement or recreation), of 48 total sensor/linkage creations.

When code behaves in unexpected ways, users can detect these problems either through unexpected visual outputs from layers or through unexpected end-to-end behavior. Because Ply encourages users to deconstruct programs into layers, users can find and fix errors in one layer while locking existing working functionality.

In self-guided tasks, discovering and correcting errors was more freeform (and some errors went uncorrected), because users were free to choose how they ultimately completed the task. In the Pokémon task, for example, six errors were detected through layer visualizations, and of the five that were corrected by the user, three were corrected by recreating sensors from scratch and two through chat refinement.

6.2.3 Parameterization as documentation. Although parameterizations were largely designed to provide *control* over synthesized code, we sometimes observed that these configuration UIs helped users understand how some generated code worked or what it did, even before tweaking the parameters. For example, seeing a *polling interval* parameter on a sensor that detects an object in a camera feed served as an indication that the sensor was periodically checking the feed, rather than reacting to changes in the feed. Giving operable *controls* may help to overcome the information overload presented in text-only chat responses.

7 Discussion

Ply's core features offer visibility over and control into the structure of trigger-action programs. We compile insights from our worked examples, our user study, and personal usage experience.

7.1 Trade-offs

Ply's programming model is not as expressive as that of general, unstructured code. Here, we discuss some conceptual limitations and how they trade off against desirable characteristics of Ply.

7.1.1 State-heavy programs. Programs that cannot be framed as trigger-action pairs may not be well-suited to being built in Ply. For example, consider a "snake" game which accepts directional button inputs and uses a visual display to show the current status of the game. It is unclear how a user of Ply should compose sensors in a way that meaningfully breaks down this complex program. Ply *can* implement such a game in a one-shot linkage prompt, directly mapping a non-layered "buttons" sensor to a non-layered "output display" sensor. The result is a game with state stored internally in the runtime data of the linkage, which cannot be inspected using Ply's multi-layer interface; this interaction essentially reduces to a simpler chatbot-driven code synthesis tool.

Long-term state, computed and then stored for later executions, may also be awkward to include in a trigger-action program, since it is not possible to "loop" back to the start of the execution graph using information computed later in the program's execution. One solution is to route persistent information *through* the real world, *e.g.,* by using an actuator to add information to a spreadsheet, then using a sensor to re-ingest that information from the same sheet. This is reminiscent of software engineering architecture techniques that insist on unidirectional dataflow until new executions are triggered from sources external to the data processing code (*e.g.,* [49]).

This requirement for looping programs to exit and re-enter Ply may even be desirable, since it encourages users to author programs that store their state in the real world. This can improve *visibility* into the state of long-term workflows even when Ply is not in active use, which can support coordination external to the software [33]. Using the external environment to store and retrieve state can also improve applications' compatibility with other workflows. For example, consider two programs to toggle the state of a smart outlet that receives "on" or "off" event payloads. The first program links a button directly to the outlet; this program stores a state value for the outlet within the linkage, and a button press flips this state and sends an "on" or "off" event to the outlet. The second program uses a button *and* a sensor that pulls in the *current* state of the outlet on the trigger side; the button press dispatches a payload that flips the outlet to the opposite state. Because the second program uses the up-to-date outlet state instead of an internal state variable, it is resilient to the state being changed from outside of Ply, for example by a physical button on the outlet.

7.1.2 Coupling data and behavior. Linkages and sensor/actuator layers in Ply are not polymorphic; their code is written *ad hoc* to use exactly the available dependencies. By synthesizing code on the spot to join together concrete implementations, Ply frees users from needing to think about abstract interfaces between components of their programs. As long as some data is visible in an event payload, it can be used in a subsequent higher-level layer.

This makes authoring data transformations very straightforward. When building up programs based on existing sensors, the user prompts Ply in terms of the data *already available* from the underlying sensor implementation, relying on the LLM backend to interpret the prompt in context with dependencies' payload types and sensors' example payloads.

This approach also frees integrated tools to dispatch or receive payloads in any format they choose, as long as the format is documented in natural language. Approaches that require tools to comply with standard interchange formats can require heavy work on the part of the developer to meet these standards, and they also discourage tools from introducing unique affordances that would not be supported by compatibility layers. Ply, on the other hand, can support any documented behavior of an integration. Because it generates interoperability code on the fly, it does not discriminate between standard and nonstandard devices.

The drawback to this approach is that dependencies cannot easily be replaced or modified after they have been integrated into a program; replacing a dependency would require nontrivial logic to handle potential conflicts or incompatibilities, which could cascade through a program's code. Ply's visualizations help to allay this downside by empowering users to test sensor layer components before committing to building upon them, and generated parameterizations can also help layer code remain flexible.

7.1.3 Locality. Programs authored in Ply are split across abstraction layers, and downstream uses of a layer are reliant on that layer's documentation (not its code implementation) to ensure correct use. AI features of Ply are not able to cross these boundaries and modify multiple layers' code at once; this choice permits users to create, test, and then "freeze" components in place. Sensor layers, then, serve as *checkpoints* for program behavior that can be reused later. This is in contrast to tools that directly generate and update large blocks of text-based code, which may make unexpected changes to aspects of full programs that the user did not want to modify.

However, this locality can be limiting, where broadly-scoped context would help the AI provide non-obvious recommendations to the user. For example, the *staleness* behavior discussed in Section 4.2.3 can be handled in different ways, depending on the context. Take the example of a sensor that translates the text on the current slide of a presentation. If the presenter skips through many slides too quickly for translations to be created, should the sensor discard stale translations, or should they be queued so that all translations are eventually provided by the sensor? This will depend on the application's downstream use of translations, but Ply will make a choice about this layer's staleness properties before the final trigger-action program is written; users can request a change to this behavior, but only if they understand that the AI has made a decision on their behalf.

7.2 Composing programs with layers

Ply allows boundaries between software components to be specified in approximate terms, using natural language instead of specific data types to describe the data provided by a sensor layer. This can simplify the process of decomposing a program into components. However, participants in our user study used programs with many redundant linkages instead of building minimal decompositions.

Ply flexibly supports both workflows that engage in much decomposition and those engaging in little decomposition, and many participants used different approaches in different tasks. Still, Ply could be improved in the future to provide more support in creating and informing the user about program decompositions. Because the visualization and parameterization tools offered by Ply are most effective when layer abstractions are created, Ply's interactions should encourage and support decomposition-heavy workflows.

7.3 Code generation for a messy environment

Through using Ply, we sometimes noticed characteristics of some generated code that, although we did not prompt for them, we believe to be valuable.

7.3.1 Defensive programming. Generated code was often *defensive*, checking input payloads carefully (*e.g.,* to prevent null dereferencing) even when the datatype was specified by the documentation from dependencies. This can help reduce the impact of a miscommunication or a mistake from an underlying dependency, providing some reasonable default even when dependencies do not adhere to their self-stated contracts. There is no runtime type checking in Ply, so it is important for components to be able to fail gracefully if unexpected inputs are received. Even layers' internal state data can be made obsolete by user updates to parameters (*e.g.,* causing a list index to go out of range), but we did not find this to cause frequent breakages in defensively-written code.

7.3.2 Information passthrough. When we switched Ply's primary LLM from o3-mini to Claude 3.7 Sonnet, Claude was more likely to include not just the requested output data, but also a copy of the input data that was used to process the output. For example, in the "slang detector" task from our user study, a sensor that detects and translates slang may also include the entire original text from which slang is being extracted.

This additional context can be helpful in a few ways. The sensor's output visualization can include this information, in this case highlighting *where* slang appeared in the original text. In addition, a downstream sensor may use this context to, *e.g.,* improve the quality of a translation. If a sensor needs to be changed, having additional data from dependency sensors improves the chances that the right information is available to implement the updated behavior without having to first update the dependency explicitly. Because sensors are built incrementally, it is not always clear in advance which information may be needed in the future. Including more information can help deal with this uncertainty.

This behavior does begin to violate the abstraction suggested by Ply's core layering technique, which may cause unexpected behavior. However, each new layer is a chance for Ply to determine which prior information remains relevant and to recontextualize the data with respect to the new layer (for example, calling an output value "pre-translation text" instead of "text from microphone").

7.4 Conclusion and future work

We present Ply, a system that supports trigger-action programming through code generation. By offering users tools to **decompose**, **visualize**, and **parameterize** their programs, we offer finer understanding and control of the programs built with LLM assistance. We discuss five example programs and report on a user study to describe how users engage with this new programming technique.

Because Ply can author code on the fly to meet users' demands, it can integrate much more flexibly with external tools than traditional trigger-action programming systems can. We believe that Ply and future systems of this nature are valuable for more than just automating or speeding up workflows. These systems can improve accessibility through interface customizability, including through physical, real-time control of software. Users can explore novel creative uses of their software tools through new, custom-built interactions. These custom tools could allow users to set their own stage for computer-supported expressive *performances*, beyond integrations just with slideshow presentation software.

Future work could expand Ply to new domains, focusing, for example, on programs that continuously map inputs to outputs or on programs that require more explicit long-term state management. Other improvements could give users more visibility into how programs work. For example, Ply could show the state *inside* generated layers, rather than just at the boundaries of system components. Future work could also lean into the communicative power of *parameterizations* as a means of outlining a component's functionality beyond long chat-response descriptions, for example by requiring users to choose some parameters (instead of providing defaults) as a way to induce reflection in users about how generated components work.

Although layer decomposition was central to Ply, not all participants made full use of this decomposition strategy to author more complex programs. Future work should investigate the question: could improved usability or a longer-term deployment of Ply elicit more of this type of use, or does decomposition remain a difficult task in building programs regardless? Our Ply-only evaluation stops short of comparing the usability of Ply's decomposition approach to that of other tools, and a comparative evaluation could offer valuable insight into the design of future programming systems.

To improve Ply's generalizability, later work could tackle the complex interactions that would enable users to swap dependencies after building programs without sacrificing the benefits provided by concrete, grounded code synthesis. We even see potential for Ply's techniques in impacting practice beyond trigger-action programming, through building similar visualization and parameterization tools for other paradigms (like node-based or even text-based programming).

Acknowledgments

We thank James Smith, Shm Garanganao Almeda, J.D. Zamfirescu-Pereira, Peitong Duan, and Matthew T. Beaudouin-Lafon for conversations that inspired this work and for feedback, both early in its conception and as it neared completion. We also thank Shm for lending us Pokémon cards for our user study.

References

[1] Shm Garanganao Almeda, J.D. Zamfirescu-Pereira, Kyu Won Kim, Pradeep Mani Rathnam, and Bjoern Hartmann. 2024. Prompting for Discovery: Flexible Sense-Making for AI Art-Making with Dreamsheets. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems (CHI '24)*. Association for Computing Machinery, New York, NY, USA, 1–17. doi:10.1145/3613904.3642858

[2] Fraser Anderson, Tovi Grossman, and George Fitzmaurice. 2017. Trigger-Action-Circuits: Leveraging Generative Design to Enable Novices to Design and Build Circuitry. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*. ACM, Québec City QC Canada, 331–342. doi:10.1145/3126594.3126637

[3] Anthropic. 2025. *Google Sheets add-on*. https://docs.anthropic.com/en/docs/agents-and-tools/claude-for-sheets

[4] Ian Arawjo, Chelse Swoopes, Priyan Vaithilingam, Martin Wattenberg, and Elena L. Glassman. 2024. ChainForge: A Visual Toolkit for Prompt Engineering and LLM Hypothesis Testing. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems (CHI '24)*. Association for Computing Machinery, New York, NY, USA, 1–18. doi:10.1145/3613904.3642016

[5] Home Assistant. 2024. Home Assistant. https://www.home-assistant.io/

[6] Timothy J. Aveni, James Smith, Armando Fox, and Björn Hartmann. 2025. Supporting Students in Prototyping AI-backed Software with Hosted Prompt Template APIs. In *Proceedings of the 30th ACM Conference on Innovation and Technology in Computer Science Education V. 1* (Nijmegen, Netherlands) *(ITiCSE 2025)*. Association for Computing Machinery, New York, NY, USA, 65–71. doi:10.1145/3724363.3729109

[7] Rafael Ballagas, Faraz Memon, Rene Reiners, and Jan Borchers. 2007. iStuff mobile: rapidly prototyping new mobile phone interfaces for ubiquitous computing. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '07)*. Association for Computing Machinery, New York, NY, USA, 1107–1116. doi:10.1145/1240624.1240793

[8] Rafael Ballagas, Meredith Ringel, Maureen Stone, and Jan Borchers. 2003. iStuff: a physical user interface toolkit for ubiquitous computing environments. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '03)*. Association for Computing Machinery, New York, NY, USA, 537–544. doi:10.1145/642611.642705

[9] Shraddha Barke, Michael B. James, and Nadia Polikarpova. 2023. Grounded Copilot: How Programmers Interact with Code-Generating Models. *Proc. ACM Program. Lang.* 7, OOPSLA1, Article 78 (April 2023), 27 pages. doi:10.1145/3586030

[10] Kolja Beigel. 2025. *KoljaB/RealtimeSTT*. https://github.com/KoljaB/RealtimeSTT original-date: 2023-08-29T17:58:28Z.

[11] Michael Bolin, Matthew Webber, Philip Rha, Tom Wilson, and Robert C. Miller. 2005. Automation and customization of rendered web pages. In *Proceedings of the 18th annual ACM symposium on User interface software and technology (UIST '05)*. Association for Computing Machinery, New York, NY, USA, 163–172. doi:10.1145/1095034.1095062

[12] Will Brackenbury, Abhimanyu Deora, Jillian Ritchey, Jason Vallee, Weijia He, Guan Wang, Michael L. Littman, and Blase Ur. 2019. How Users Interpret Bugs in Trigger-Action Programming. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems* (Glasgow, Scotland Uk) *(CHI '19)*. Association for Computing Machinery, New York, NY, USA, 1–12. doi:10.1145/3290605.3300782

[13] Yining Cao, Peiling Jiang, and Haijun Xia. 2025. Generative and Malleable User Interfaces with Generative and Evolving Task-Driven Data Model. doi:10.48550/arXiv.2503.04084 arXiv:2503.04084 [cs].

[14] Sarah E. Chasins, Maria Mueller, and Rastislav Bodik. 2018. Rousillon: Scraping Distributed Hierarchical Web Data. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology (UIST '18)*. Association for Computing Machinery, New York, NY, USA, 963–975. doi:10.1145/3242587.3242661

[15] Ruijia Cheng, Titus Barik, Alan Leung, Fred Hohman, and Jeffrey Nichols. 2024. BISCUIT: Scaffolding LLM-Generated Code with Ephemeral UIs in Computational Notebooks. doi:10.48550/arXiv.2404.07387 arXiv:2404.07387 [cs].

[16] Color.js. 2025. *Color.js: Let's get serious about color • Color.js*. https://colorjs.io/

[17] Fulvio Corno, Luigi De Russis, and Alberto Monge Roffarello. 2019. Empowering End Users in Debugging Trigger-Action Rules. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems* (Glasgow, Scotland Uk) *(CHI '19)*. Association for Computing Machinery, New York, NY, USA, 1–13. doi:10.1145/3290605.3300618

[18] Fulvio Corno, Luigi De Russis, and Alberto Monge Roffarello. 2020. TAPrec: supporting the composition of trigger-action rules through dynamic recommendations. In *Proceedings of the 25th International Conference on Intelligent User Interfaces* (Cagliari, Italy) *(IUI '20)*. Association for Computing Machinery, New York, NY, USA, 579–588. doi:10.1145/3377325.3377499

[19] Luigi De Russis and Alberto Monge Roffarello. 2018. A Debugging Approach for Trigger-Action Programming. In *Extended Abstracts of the 2018 CHI Conference on Human Factors in Computing Systems* (Montreal QC, Canada) *(CHI EA '18)*. Association for Computing Machinery, New York, NY, USA, 1–6. doi:10.1145/3170427.3188641

[20] Paul Denny, Viraj Kumar, and Nasser Giacaman. 2023. Conversing with Copilot: Exploring Prompt Engineering for Solving CS1 Problems Using Natural Language. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1 (SIGCSE 2023)*. Association for Computing Machinery, New York, NY, USA, 1136–1142. doi:10.1145/3545945.3569823

[21] OpenJS Foundation and Contributors. 2025. *Node-RED*. https://nodered.org original-date: 2013-09-05T13:30:47Z.

[22] 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. doi:10.1145/964442.964461

[23] Krzysztof Z. Gajos, Jacob O. Wobbrock, and Daniel S. Weld. 2007. Automatically generating user interfaces adapted to users' motor and vision capabilities. In *Proceedings of the 20th annual ACM symposium on User interface software and technology (UIST '07)*. Association for Computing Machinery, New York, NY, USA, 231–240. doi:10.1145/1294211.1294253

[24] Google. 2025. *google-ai-edge/mediapipe*. https://github.com/google-ai-edge/mediapipe original-date: 2019-06-13T19:16:41Z.

[25] Saul Greenberg and Chester Fitchett. 2001. Phidgets: easy development of physical interfaces through physical widgets. In *Proceedings of the 14th annual ACM symposium on User interface software and technology (UIST '01)*. Association for Computing Machinery, New York, NY, USA, 209–218. doi:10.1145/502348.502388

[26] Björn Hartmann, Leslie Wu, Kevin Collins, and Scott R. Klemmer. 2007. Programming by a sample: rapidly creating web applications with d.mix. In *Proceedings of the 20th annual ACM symposium on User interface software and technology*. ACM, Newport Rhode Island USA, 241–250. doi:10.1145/1294211.1294254

[27] Justin Huang and Maya Cakmak. 2015. Supporting mental model accuracy in trigger-action programming. In *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing* (Osaka, Japan) *(UbiComp '15)*. Association for Computing Machinery, New York, NY, USA, 215–225. doi:10.1145/2750858.2805830

[28] IFTTT. 2025. *IFTTT - Automate business & home*. https://ifttt.com/

[29] Brad Johanson and Armando Fox. 2002. The Event Heap: a coordination infrastructure for interactive workspaces. In *Proceedings Fourth IEEE Workshop on Mobile Computing Systems and Applications*. IEEE Comput. Soc, Callicoon, NY, USA, 83–93. doi:10.1109/MCSA.2002.1017488

[30] Koen Kanters. 2025. *Koenkk/zigbee2mqtt*. https://github.com/Koenkk/zigbee2mqtt original-date: 2018-04-08T12:01:34Z.

[31] Yoshiharu Kato. 2010. Splish: A Visual Programming Environment for Arduino to Accelerate Physical Computing Experiences. In *2010 Eighth International Conference on Creating, Connecting and Collaborating through Computing*. 3–10. doi:10.1109/C5.2010.20 ISSN: 1556-0090.

[32] Caitlin Kelleher, Randy Pausch, and Sara Kiesler. 2007. Storytelling alice motivates middle school girls to learn computer programming. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, San Jose California USA, 1455–1464. doi:10.1145/1240624.1240844

[33] Scott R. Klemmer, Björn Hartmann, and Leila Takayama. 2006. How bodies matter: five themes for interaction design. In *Proceedings of the 6th conference on Designing Interactive systems (DIS '06)*. Association for Computing Machinery, New York, NY, USA, 140–149. doi:10.1145/1142405.1142429

[34] Amy J. Ko, Brad A. Myers, and Htet Htet Aung. 2004. Six Learning Barriers in End-User Programming Systems. In *2004 IEEE Symposium on Visual Languages - Human Centric Computing*. IEEE, Rome, Italy, 199–206. doi:10.1109/VLHCC.2004.47

[35] Black Forest Labs. 2025. *black-forest-labs/flux*. https://github.com/black-forest-labs/flux original-date: 2024-08-01T09:04:19Z.

[36] Gierad Laput, Walter S. Lasecki, Jason Wiese, Robert Xiao, Jeffrey P. Bigham, and Chris Harrison. 2015. Zensors: Adaptive, Rapidly Deployable, Human-Intelligent Sensor Feeds. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems (CHI '15)*. Association for Computing Machinery, New York, NY, USA, 1935–1944. doi:10.1145/2702123.2702416

[37] Nicola Leonardi, Marco Manca, Fabio Paternò, and Carmen Santoro. 2019. Trigger-Action Programming for Personalising Humanoid Robot Behaviour. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems* (Glasgow, Scotland Uk) *(CHI '19)*. Association for Computing Machinery, New York, NY, USA, 1–13. doi:10.1145/3290605.3300675

[38] Gilly Leshed, Eben M. Haber, Tara Matthews, and Tessa Lau. 2008. CoScripter: automating & sharing how-to knowledge in the enterprise. In *Proceedings of the*

SIGCHI Conference on Human Factors in Computing Systems. ACM, Florence Italy, 1719–1728. doi:10.1145/1357054.1357323

[39] Michael Xieyang Liu, Savvas Petridis, Vivian Tsai, Alexander J. Fiannaca, Alex Olwal, Michael Terry, and Carrie J. Cai. 2025. Gensors: Authoring Personalized Visual Sensors with Multimodal Foundation Models and Reasoning. In *Proceedings of the 30th International Conference on Intelligent User Interfaces (IUI '25)*. Association for Computing Machinery, New York, NY, USA, 755–770. doi:10.1145/3708359.3712085

[40] Yue Liu, Thanh Le-Cong, Ratnadira Widyasari, Chakkrit Tantithamthavorn, Li Li, Xuan-Bach D. Le, and David Lo. 2024. Refining ChatGPT-Generated Code: Characterizing and Mitigating Code Quality Issues. *ACM Trans. Softw. Eng. Methodol.* 33, 5 (June 2024), 116:1–116:26. doi:10.1145/3643674

[41] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. 2010. The Scratch Programming Language and Environment. *ACM Trans. Comput. Educ.* 10, 4 (Nov. 2010), 16:1–16:15. doi:10.1145/1868358.1868363

[42] Microsoft. 2024. Xbox Adaptive Controller | Xbox. https://www.xbox.com/en-US/accessories/controllers/xbox-adaptive-controller

[43] Bryan Min, Allen Chen, Yining Cao, and Haijun Xia. 2025. Malleable Overview-Detail Interfaces. doi:10.1145/3706598.3714164 arXiv:2503.07782 [cs].

[44] Michael Montaque. 2025. *@technithusiast/node-red-contrib-ai-intent.* http://flows.nodered.org/node/@technithusiast/node-red-contrib-ai-intent

[45] n8n. 2025. *n8n-io/n8n.* https://github.com/n8n-io/n8n original-date: 2019-06-22T09:24:21Z.

[46] Alex O'Connell. 2025. *acon96/home-llm.* https://github.com/acon96/home-llm original-date: 2023-12-23T04:50:06Z.

[47] Comfy Organization. 2024. *ComfyUI.* https://www.comfy.org/

[48] Yvonne Rogers and Henk Muller. 2006. A framework for designing sensor-based interactions to promote exploration and reflection in play. *Int. J. Hum.-Comput. Stud.* 64, 1 (Jan. 2006), 1–14. doi:10.1016/j.ijhcs.2005.05.004

[49] Meta Open Source. 2023. *In-Depth Overview | Flux.* https://facebookarchive.github.io/flux/docs/in-depth-overview/

[50] Tray.ai. 2025. AI-ready integration & automation platform. https://tray.ai/

[51] Blase Ur, Elyse McManus, Melwyn Pak Yong Ho, and Michael L. Littman. 2014. Practical trigger-action programming in the smart home. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Toronto, Ontario, Canada) *(CHI '14)*. Association for Computing Machinery, New York, NY, USA, 803–812. doi:10.1145/2556288.2557420

[52] Blase Ur, Melwyn Pak Yong Ho, Stephen Brawner, Jiyun Lee, Sarah Mennicken, Noah Picard, Diane Schulze, and Michael L. Littman. 2016. Trigger-Action Programming in the Wild: An Analysis of 200,000 IFTTT Recipes. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems* (San Jose, California, USA) *(CHI '16)*. Association for Computing Machinery, New York, NY, USA, 3227–3231. doi:10.1145/2858036.2858556

[53] Priyan Vaithilingam, Elena L. Glassman, Jeevana Priya Inala, and Chenglong Wang. 2024. DynaVis: Dynamically Synthesized UI Widgets for Visualization Editing. doi:10.48550/arXiv.2401.10880 arXiv:2401.10880.

[54] Tongshuang Wu, Michael Terry, and Carrie Jun Cai. 2022. AI Chains: Transparent and Controllable Human-AI Interaction by Chaining Large Language Model Prompts. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems (CHI '22)*. Association for Computing Machinery, New York, NY, USA, 1–22. doi:10.1145/3491102.3517582

[55] Ryan Yen, Jiawen Stefanie Zhu, Sangho Suh, Haijun Xia, and Jian Zhao. 2024. CoLadder: Manipulating Code Generation via Multi-Level Blocks. In *Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology (UIST '24)*. Association for Computing Machinery, New York, NY, USA, 1–20. doi:10.1145/3654777.3676357

[56] J. D. Zamfirescu-Pereira, Eunice Jun, Michael Terry, Qian Yang, and Björn Hartmann. 2025. Beyond Code Generation: LLM-supported Exploration of the Program Design Space. doi:10.1145/3706598.3714154 arXiv:2503.06911 [cs].

[57] Valerie Zhao, Lefan Zhang, Bo Wang, Shan Lu, and Blase Ur. 2020. Visualizing Differences to Improve End-User Understanding of Trigger-Action Programs. In *Extended Abstracts of the 2020 CHI Conference on Human Factors in Computing Systems* (Honolulu, HI, USA) *(CHI EA '20)*. Association for Computing Machinery, New York, NY, USA, 1–10. doi:10.1145/3334480.3382940

[58] Zhongyi Zhou, Jing Jin, Vrushank Phadnis, Xiuxiu Yuan, Jun Jiang, Xun Qian, Kristen Wright, Mark Sherwood, Jason Mayes, Jingtao Zhou, Yiyi Huang, Zheng Xu, Yinda Zhang, Johnny Lee, Alex Olwal, David Kim, Ram Iyengar, Na Li, and Ruofei Du. 2025. InstructPipe: Generating Visual Blocks Pipelines with Human Instructions and LLMs. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems (CHI '25)*. Association for Computing Machinery, New York, NY, USA, 1–22. doi:10.1145/3706598.3713905