Engraft: An API for Live, Rich, and Composable Programming

Joshua Horowitz
joshuah@alum.mit.edu
University of Washington
Seattle, United States

Jeffrey Heer
jheer@uw.edu
University of Washington
Seattle, United States

ABSTRACT

Live & rich tools can support a diversity of domain-specific programming tasks, from visualization authoring to data wrangling. Real-world programming, however, requires performing multiple tasks in concert, calling for the use of multiple tools alongside conventional code. Programmers lack environments capable of composing live & rich tools to support these situations. To enable this composition, we contribute Engraft, a component-based API that allows live & rich tools to be embedded within larger environments like computational notebooks. Through recursive embedding of components, Engraft enables several new forms of composition: not only embedding tools inside environments, but also embedding environments within each other and embedding tools and environments in the outside world, including conventional codebases. We demonstrate Engraft with examples from diverse domains, including web-application development, command-line scripting, and physics education. By providing composability, Engraft can help cultivate a cycle of use and innovation in live & rich programming.

CCS CONCEPTS

• **Human-centered computing** → **Graphical user interfaces**; • **Software and its engineering** → **Integrated and visual development environments**; **Application specific development environments**.

KEYWORDS

live programming, visual programming, end-user programming, composition, computational notebooks, GUIs

ACM Reference Format:
Joshua Horowitz and Jeffrey Heer. 2023. Engraft: An API for Live, Rich, and Composable Programming. In *The 36th Annual ACM Symposium on User Interface Software and Technology (UIST '23), October 29–November 01, 2023, San Francisco, CA, USA.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3586183.3606733

1 INTRODUCTION

Although the most familiar interface for editing programs is a plain-text editor, alternatives to static textual code have a long history. These alternatives range from spreadsheets [28] to block-based programming [42] to computational notebooks [20] and beyond. In recent decades, a distinctive new category of non-static-text programming tools has emerged. These tools bring tailored interfaces and interactions to tasks which previously would have required traditional code. For instance, Gneiss [4] lets users make web applications through direct manipulation centered around a spreadsheet. Lyra [43] lets authors create expressive data visualizations with drag-and-drop interactions. And Wrangler [17] lets users define complex scripts to clean up tabular data by directly manipulating a table. These are examples of *live* and *rich* tools: *live* in the sense that their editing interfaces incorporate feedback from a running program, and *rich* in the sense that they let programmers edit programs using domain-specific visual representations and interactions instead of code [14].

Live & rich tools make programming tasks direct, visible, and approachable. However, while particular live & rich tools have found niches of use, their role in programming at large remains marginal. Given the success of specialized direct-manipulation interfaces in everyday computing, this fact is striking. What fundamental limitations do live & rich programming tools have today which, if addressed, might bring them into broader use?

A major factor limiting use of live & rich programming tools is their present lack of *composability* [14]. Conventional programming derives its power in large part from composability, as programmers freely combine libraries, APIs, and language features to accomplish their goals. However, composability like this does not exist in the world of user interfaces, including live & rich tools.

As an example, suppose a digital artist wants to make a website that displays summaries of random Wikipedia articles.[1] This project involves multiple steps:

(1) Call the Wikipedia API to get raw data about articles.
(2) Process a JSON response to extract relevant information.
(3) Format this information into an attractive web page.

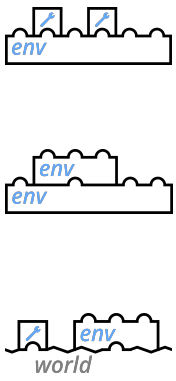
This is a heterogeneous workflow. We can imagine separate live & rich tools for each of these steps, like a tool which displays a JSON structure and lets a user pick the information they want (for step 2), and a tool which lets a user build a data-backed web page through direct manipulation (for step 3). Could the artist use such tools together to accomplish their larger goal? Right now, they cannot. Ad-hoc approaches are possible, including tools that generate code or hard-wired tool assemblies for specific workflows, but each gives up either the benefits of liveness or the flexibility of open-ended composition.

A practical programming system for live & rich programming must provide environments, such as computational notebooks, where domain-specific tools can be freely composed. This is composition of **tools in environments**. While this form of composition is significant and necessary, it is only the first step. Single environments reach a limit of expressivity when a situation calls for

[1] This example is inspired by a live-tweeted coding adventure described in Pipkin [36].

UIST '23, October 29–November 01, 2023, San Francisco, CA, USA

ACM ISBN 979-8-4007-0132-0/23/10.
https://doi.org/10.1145/3586183.3606733



Tools in environments. Live & rich tools can be joined together in a shared environment, where liveness between tools can be maintained.

Environments in environments. Environments can embed into other live environments, representing multiple levels in the programming process with nested scopes.

Tools and environments in the outside world. Environments containing live & rich tools can be embedded into codebases, applications, and other parts of the computing world.

Figure 1: Three forms of composition made possible with Engraft.

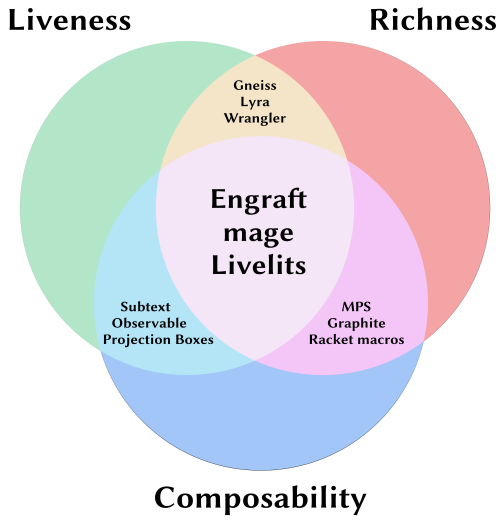


Figure 2: Pairwise and three-way intersections between liveness, richness, and composability.

nested structures like functions, loops, or conditionals. To use live & rich tools and environments inside these structures, we need the ability to nest environments into one another the same way traditional code is nested. This is composition of **environments in environments**. Finally, for the programs we make with these tools and environments to provide value in real-world contexts, we need to be able to connect them with existing programming activity, such as work in conventional codebases. This is composition of **tools and environments in the outside world**.

No programming system is available today which enables all three of these forms of composition. Without such a system, live & rich tools are limited in the real-world situations they can support.

In this paper, we present *Engraft*[2], a component-based API for interactive programming on the web platform which makes all three forms of composition possible. Live & rich tools implemented as Engraft components can be hosted together within live environments like notebooks. In this way, disparate task-specific tools can be joined together to solve problems that no single tool could solve alone, while maintaining liveness between tools. Furthermore, Engraft allows not just tools but *environments* to be implemented as components, making recursive embedding of tools and environments possible and enabling all three forms of composition.

In summary, this paper describes the following contributions:

- We articulate three forms of composition that are necessary for live & rich programming to match conventional programming in expressiveness and practical utility (§2.3).
- We present the design of Engraft, a functional API that represents both tools and environments with a uniform component interface (§3). Through recursive embedding, this API enables all three forms of composition.
- We present our open-source implementation of this API, together with numerous example components that showcase interaction styles that might be used in future tools & environments (§5). This implementation includes a new library for incremental computation which enables efficient updates to Engraft components while preserving Engraft's functional semantics (§5.2).
- We evaluate Engraft through a range of demonstrations (§4), as well as with a heuristic analysis based on Jakubovic et al.'s "Technical Dimensions of Programming Systems" [15] (§6)

2 BACKGROUND

Engraft brings new forms of composition to programming systems at the intersection of *liveness* and *richness*: tools like Gneiss [4], Lyra [43], and Wrangler [17]. In this section, we review the concepts of liveness and richness, the challenges of composing live & rich systems, and efforts made so far to overcome these challenges. (Our presentation here roughly follows that of Horowitz & Heer [14].)

2.1 Liveness & Richness

The term "live" refers to programming systems which "provide immediate feedback on the dynamic behavior of a program even while programming" [41].[3] Within this broad definition, liveness can take many forms. Inline displays of run-time values and behavior can be brought into traditional code editors [22, 40]. Inspired by spreadsheets, computational notebooks like Jupyter [20] and Observable [29] separate computations into cells which each show their own output. Farther from traditional code, we find structured editors which integrate live feedback into an interface that directly manipulates an underlying AST [8, 13, 24].

Gneiss, Lyra, and Wrangler, the tools we opened this paper with, are also live. Indeed, their liveness is a crucial part of their designs. Each is driven by concrete input data and shows the effects of

[2]Grafting is "a horticultural technique whereby tissues of plants are joined so as to continue their growth together" [53]. This provides an apt analogy for the way Engraft allows trees of live & rich tools and environments to work together.

[3]The use of "liveness" as a term of art dates back at least to Tanimoto [50].

programmers' actions on this data immediately. However, while the examples of programming systems listed in the last paragraph provide liveness in the context of textual code or an analogous symbolic interface, these three tools do not. Instead, each is based around graphical representations and direct-manipulation interactions specifically crafted for its domain. This suggests there is a second quality these tools share which they combine synergetically with liveness.

Horowitz & Heer [14] proposes "richness" as a name for this second quality, stating as a definition that "[a] programming system embodies richness insofar as it allows programmers to edit programs through visualizations and interactions that are tailor-made to the tasks, domains, and contexts they work in". Richness is not a subset or superset of liveness, but is rather an independent quality.[4] When richness and liveness are combined, new designs become possible. One particularly compelling possibility is *programming by demonstration* (PbD) [6]. This refers to programming systems that let programmers act directly on concrete data, with the programming system generalizing from this demonstration to new data, thereby creating a new program. Gneiss, Lyra, and Wrangler all qualify, at least in part, as PbD systems. Other live & rich tools, like InterState [35] and Object-Oriented Drawing [55], follow non-PbD paths, maintaining symbolic forms of expression but augmenting them with live feedback and domain-specific interactions.

2.2 Composability of Live & Rich Tools

The development of live & rich tools presents a unique opportunity to bring the ease-of-use and fluid productivity of application software into the world of programming. Our work on Engraft began with the realization that live & rich tools are generally not composable, undermining this opportunity.

In §1, we gave an example of a digital artist making a website driven by the Wikipedia API. Their work can be broken into multiple steps, each of which could be compellingly served by a live & rich tool. However, there is no environment available today in which such tools can be integrated. Live & rich tools (including Gneiss, Lyra, and Wrangler) are generally implemented as standalone application software, as this gives them full control over what graphical representations and interactions they use. But operating systems provide only manual forms of data flow between graphical applications, such as copy-paste and loading and saving files. Manual composition is slow and tedious. Furthermore, effects of changes in an "upstream" tool will only be visible in "downstream" tool after the user manually moves data between them. This means that even if programmers have liveness *within* their tools, they will lack liveness *between* their tools.

Some projects have tried to achieve liveness between rich tools by pre-packaging a set of tools for a specific domain into a single application. Gneiss [4] can be viewed as an example of this pattern, as it combines three tools, each of which might be useful by itself. Bundles like this can help users whose needs lie along the "happy path" of the application's design, but the moment their needs diverge, the bundle will no longer be usable. The limitations of pre-bundled applications are highlighted by a comparison to traditional programming, where libraries can be composed together at will.

To be clear, prior efforts are only limited by this lack of composability when they combine liveness *and* richness. Traditional code (neither live nor rich) is pervasively composable. Live-but-not-rich systems like Projection Boxes [22] extend traditional code with live feedback and rich-but-not-live systems like the visual macro system for Racket [1] extend code with graphical interactions, all without losing the composability of code. There are, however, precious few systems that combine liveness and richness without compromising composability.[5] The primary goal of Engraft is to overcome this limitation and make live, integrated composition of live & rich tools possible.

2.3 mage and Three Forms of Composition

The key idea of Engraft is that, by representing live & rich tools *and* environments through a shared "component" interface, we can enable not only composition of *tools in environments*, but also *environments in environments* and *tools and environments in the outside world*. "Tools" here refers to live & rich tools. "Environments" refers to live-programming systems that make it possible to compose separate bits of computation together in a live, integrated way. Our central examples of environments are computational notebooks like Jupyter [20] and Observable [29], but we are also inspired by divergent approaches like Natto [47], which spreads code cells out on a flexible two-dimensional canvas.

Environments like these provide a natural entry-point for live & rich tools. If cells of code can be composed together in environments, perhaps live & rich tools could take the place of code in these cells. mage [18] is a system based on this insight, aiming to provide "smooth transitions between GUI and code work" by embedding live & rich tools into Jupyter. It enables composition of tools in environments, at least the specific environment of Jupyter. Looking more closely, the example of mage shows how this one form of composition only goes so far, and how we need to support two additional forms of composition that mage does not provide.

Environments in environments. If live & rich tools can only be embedded into cells of a single notebook, they can only express a "flat" structure. Suppose a Jupyter user has an array of complex data elements and they want to process each element to obtain a new version of the array. In Jupyter, they can write a cell that does this with a "map" operator:

```
new_list = map(
  lambda element: step3(step2(step1(element))),
  old_list
)
```

Their per-item processing function consists of multiple function-call steps. A selling point of a computational notebook is that it can split separate steps into separate cells, providing visibility into intermediate values. But because this map operation needs to be contained within a single cell, the benefits of visibility that Jupyter

[4] Richness without liveness can take the form of "visual syntax" – directly editable domain-specific representations which replace certain instances of traditional code, as can be found in the visual macro system for Racket described in Andersen et al. [1], Graphite [34], and MPS [16].

[5] We direct the reader to Horowitz & Heer [14] for further analysis of the boundaries of liveness, richness, and composability. This includes discussion of the failure of most "visual programming languages" (such as Pure Data [39] and Lively Fabrik [23]) to be truly rich.

offers are not available here. Furthermore, if they are using mage, the user is unable to use live & rich tools inside this map operation, because live & rich tools can only exist on the top, cell level of the notebook. We need a way to get notebooks, and live & rich tools within them, *inside* this map operation, thereby nesting an environment within another environment.

Tools and environments in the outside world. The second limitation of the mage approach is even simpler. mage tools can only be used in Jupyter, and Jupyter cannot be integrated into larger computational systems. If live & rich tools can only be used inside of closed systems like Jupyter, they cannot be gradually incorporated into existing programming activities, severely limiting use. We call this the composition of *tools and environments in the outside world.*

2.4 Livelits

The second project most similar to Engraft in its motivations and design is livelits [32]. Livelits extends Hazel [33], "a live functional programming environment designed around hole-driven development" with "user-defined GUIs embedded persistently into code" [32]. Unlike mage tools, livelits are nestable within one another and inside Hazel programming constructs. This means that livelits achieves environment-in-environment composition, at least for the single environment of Hazel. However, new programming environments cannot be defined as livelits, because livelits' typing and binding disciplines are not expressive enough to represent them. For similar reasons, programming-by-demonstration tools like Gneiss, Lyra, and Wrangler cannot currently be implemented as livelits. In keeping with the name "live literals", livelits have so far been restricted to widgets like sliders, color pickers, and numeric arrays. As we will demonstrate in §4, Engraft supports a broader range of components, from nestable environments like **notebook** and **notebook-canvas** to programming-by-demonstration tools like **extractor** and **formatter**.

So far, livelits have not made contact with environment-in-the-outside-world composition, as they are part of Hazel, a self-contained research platform. In contrast, Engraft has prioritized direct integration with existing development environments, from traditional codebases to the UNIX shell (as we show in §4.3). In this way, Engraft can bring live & rich tools to places where programming activity is already happening and grow through contact with authentic use.

3 THE ENGRAFT ARCHITECTURE

This section presents a high-level overview of the architecture and design decisions that define Engraft. In §4, we show through demonstration how this architecture makes new forms of composition possible. In §5, we present technical details on the Engraft API.

The Engraft API describes a relationship between a *component* and its *host*. The prototypal model for this interface is the relationship between a live & rich tool (as component) and the live environment it is embedded within (as host). For a live environment like a notebook to host a live & rich tool, it must provide the tool with external context like the values of variables fed into the tool as input. In return, the tool must provide the environment with the output it computes, as well as with a user interface the environment can embed in a cell and present to a user. This exchange defines

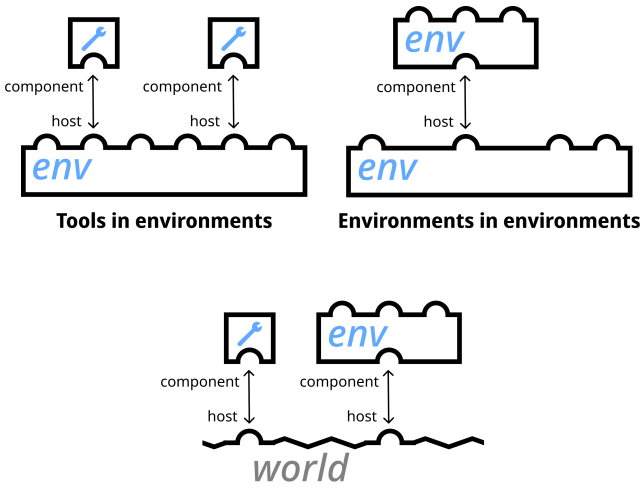


Figure 3: Through a uniform component/host interface, Engraft enables three forms of composition.

the responsibilities of an Engraft component. As we will describe in §5, an Engraft component fulfills these responsibilities with a `run` function. By calling a tool's `run` function, an environment can embed it. By calling multiple tools' `run` functions, passing the output of one tool to the input of another, an environment can wire these tools together. This is how Engraft enables live composition of *tools in environments* (Figure 3, top-left).

By itself, an environment hosting tools creates only a "flat" structure, and cannot bring live & rich tools into nested structures like loops and function definitions (see §2.3). Engraft solves this problem through *recursive embedding*. Environments can implement the component side of the Engraft API, thereby becoming components themselves which can be embedded inside of other components. With the aid of additional components, such as **map** (a component which applies a child component to every element of an array; see §4.2), this brings live environments and tools into all parts of the programming process. This is how Engraft enables live composition of *environments in environments* (Figure 3, top-right).

Finally, the component/host interface can be used to embed assemblies of tools and environments into pre-existing contexts, like conventional codebases. Any JavaScript code, including non-graphical software, can act as a host to Engraft components. This is how Engraft enables live composition of *tools and environments in the outside world* (Figure 3, bottom).

Although tools and environments play different roles in user workflows, Engraft makes no distinction between them. Both are components, and implement the same component interface. This uniformity makes it easy for Engraft to accommodate diverse forms of composition.

In the remainder of this section we outline four more important decisions we made while designing the Engraft API.

Engraft is built on the web platform. A large part of the computing world now exists on the web platform, and Engraft can be composed together with this existing activity in numerous ways.

Engraft's use of the web platform also supports the recursive embedding of components: Engraft components can host one another because they are both implemented on the web platform and hosted by the web platform. Systems like mage [18], which do not support open-ended embedding of tools on the web, lack the ability to be nested in this way.

The Engraft API is functional. A component's host provides it with variables bound to JavaScript values. The component in turn returns a new value. Extending this pattern through composition, we obtain a reactive dataflow, which has a legacy of ease-of-use and flexible composability in contexts ranging from spreadsheets to React [26]. A value-oriented paradigm also presents a natural foundation for live visibility: just show the values. (Performance challenges often accompany the use of functional programming in interactive contexts. §5.2 describes how we addressed those challenges in Engraft with a new library for incremental computation.)

Engraft components are not required to generate or edit underlying textual code. In Engraft, a component's underlying specification is not traditional code, but a serializable JavaScript object we call its *program*. The creator of a component has full control over the form of this program; to the API, the program is opaque data. This stands in contrast to a common approach (followed by, e.g., mage) where a live & rich tool generates source code defining its computational behavior, and might even be required to re-parse this code when it is manually edited by a user. The "grain" of systems where tools read and write textual code is for tools to follow pre-existing code patterns. With Engraft, we instead want to support live-and-rich-first tool design, untethered from the constraints and idioms of static text.

Engraft components can be run without being rendered. As part of running, a component offers its host a way to render the component's UI. In many situations, the host will *not* render this UI, such as when a codebase embeds a live tool in production or when a component like **map** only renders one example instance of its body. By letting components run invisibly, Engraft supports a broad range of practical computational uses, not only interactive use by a programmer.

4 LIVE COMPOSITION WITH ENGRAFT

In this section, we show and describe the end-user experience of building programs with Engraft. In three sub-sections, we demonstrate how Engraft makes three forms of composition possible. Later, in §5, we discuss the implementation of the underlying Engraft API.

4.1 Tools In Environments

Let us return to the example of a digital artist who wants to make a website that displays summaries of random Wikipedia articles, examining how this artist might accomplish their goal by composing live tools together in a live environment with Engraft.

They start by loading Graft Garden, a web application which provides a convenient starting point for using Engraft tools. Once the artist tells Graft Garden to make a new page, they are presented with a blank page (Figure 4A), containing a single small code-editor box, called a *slot*. This slot is the starting point for everything they will do.

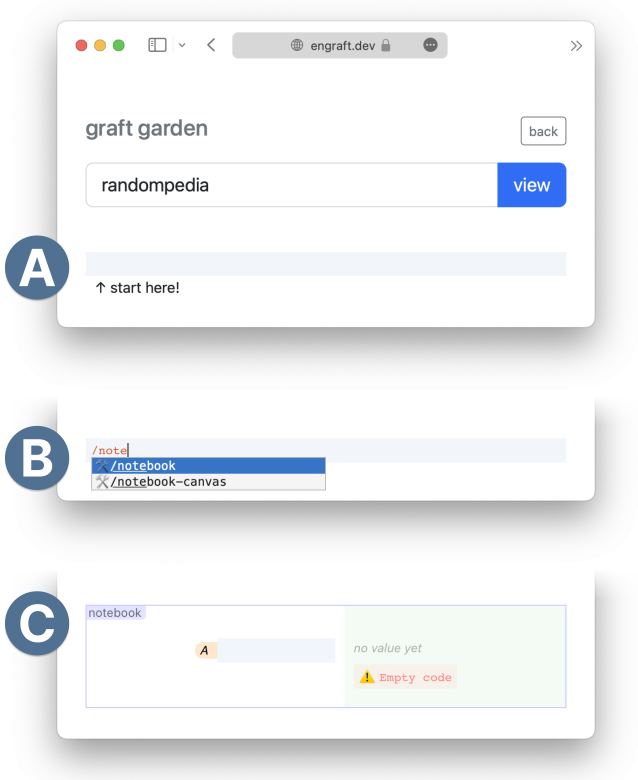


Figure 4: The artist starting out on Graft Garden. (A) A new Graft Garden page centered around an empty slot. (B) The artist begins to type `/notebook` into the slot, and an autocomplete menu appears. (C) The artist selects the autocomplete option, and a notebook is inserted into the slot in place of the code editor.

The artist knows that their project will involve combining together multiple steps. So they would like to work in a *live environment* that allows multiple computational steps to be combined together in fluid and flexible ways. They pick the environment most familiar to them: a computational notebook. They do this by beginning to type `/notebook` in the slot, and selecting 'notebook' from the autocomplete menu that pops up (Figure 4B). Once they do this, the slot is replaced with an Engraft tool called **notebook** (Figure 4C). This is a reactive notebook in which a user can type code snippets into cells. The notebook evaluates these snippets and shows their resulting values alongside the cells (Figure 5). Cells receive default names (like in a spreadsheet), but can be renamed. Cells can refer to one another, and the notebook ensures that a cell is re-evaluated when one of its references changes.

The first thing the artist needs to do in this notebook is get hold of data from the Wikipedia API. Rather than write networking code directly, they type `/request` into the first cell, and select 'request' from the autocomplete menu that pops up. This inserts a **request** tool into the cell. The notebook's cells are slots, just like the root slot the artist used to invoke the notebook, so they can also be used to insert Engraft components.

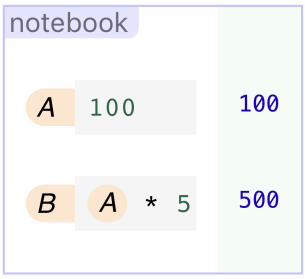


Figure 5: Example usage of a notebook. Output is shown on the right-hand side with a green background. The second cell (B) refers to the first (A).

request has two slots: one to provide a URL and one to provide an object of query parameters (Figure 6A). As the artist skims the Wikipedia API documentation, they experiment with different query parameters. The tool re-sends the query with every change, so the artist can see the effects of their changes live. Just like a snippet of code typed into a computational notebook, **request** produces *output*, which is returned to the notebook that hosts it. The notebook can then display this output live, and make it available to other cells by reference. Examining the output, and playing with query parameters, the artist eventually finds a configuration they like. It delivers five random articles, complete with titles and HTML-summary "extracts".

While the artist can see the data they want in the API response, the task of extracting it out of the complicated JSON structure is a bit intimidating. Fortunately, they know a second tool, called **extractor**[6] (Figure 6B). This tool accepts JSON data as input and presents an interface that allows the user to select data values by clicking on those values directly. The tool generalizes from these clicks, providing an output structure with the values the user wants. The result is equivalent to writing code that loops over arrays and objects with chains of property accesses. This tool is more interesting, in its liveness, than **request**. While **request** offered a convenient form interface for specifying a request, that interface was still structurally similar to working with code. In contrast, **extractor**'s interface relies on live input data to support programming-by-demonstration.

Now that the artist has extracted the data they care about, they want to reshape it into an attractive interface. For this they use a **formatter** tool[7] (Figure 6C). Given a JavaScript data structure, **formatter** automatically suggests a way to format the data into HTML output. It also provides direct-manipulation handles the user can use to choose different options for this formatting. Like **extractor**, **formatter** uses live data to power a programming-by-demonstration interface. The artist hands **formatter** the output of **extractor** as input. They click on bits of the data structure and use Formatter's inspector palette to assign styles, turning the raw data into legible HTML.

Last, the artist wants to add a header to the page. They know a bit of HTML, so they add a new cell to the notebook that they keep as a code editor, rather than invoking a rich tool. They type JSX markup in to this cell to add the header (Figure 6D).

The artist has now built a live computational notebook, consisting of four cells: (A) a **request** tool to get data from the Wikipedia API, (B) an **extractor** tool to pluck out the data they need in a clean format, (C) a **formatter** tool to shape the data into an HTML document with the appearance they desire, and (D) a **slot** tool to include a bit of traditional code.

The last step is to "deploy" this living program to a website, where a visitor will receive their own random set of articles. Conveniently, Graft Garden, the web application that has hosted their work so far, does this automatically. The output of the last cell in the notebook (the **slot** with HTML, in this case) is returned back to Graft Garden. Graft Garden provides a shareable link that displays this final return value without tool UI (Figure 7). Note that the Wikipedia articles shown here are different than above, as the artist's program runs anew every time this page is loaded.

In this example, the artist does most of their work with live & rich tools and only uses small bits of code. Other situations will instead call for mostly code, with just one or two live & rich tools invoked. Engraft supports use-cases all along this spectrum. Our goal is not to completely replace conventional programming with visual programming, but to open up a new possibility: replacing particular steps with interactive, direct-manipulation tools.

In the story above, the artist chose to compose their tools inside of a **notebook**. This is not the only live environment an Engraft user may choose to use. For different tasks, different environments may be preferred. Figure 8 shows how the tools the artist used in the **notebook** could instead be embedded in a new environment (loosely modeled on Natto [46][8]), called a **notebook-canvas**. Cells on a **notebook-canvas** work the same way as on a **notebook**, but they can be freely dragged around a canvas and resized however the user likes. A user might choose to use **notebook-canvas** over **notebook** for a number of reasons: higher visual density, space to organize nonlinear data-flows, and a looseness that avoids some "negative effects of prematurely or unnecessarily imposing a structure" [49]. The user has this choice because Engraft decouples tools from environments, making it possible to choose the right environment for the job without losing access to the right tools.

4.2 Environments In Environments

Live environments are powerful building blocks for live programming. The above example uses a single **notebook** to link together four tools. However, as we discussed in §2.3, one environment by itself is often not enough. A single environment is *flat*, and programming contains nested structures where a single environment cannot reach. Engraft allows environments to be built as components, meaning that they can be threaded through the hierarchical structures of programming along with the live & rich tools they host.

For instance, we have built a tool called **map**, designed to support a user mapping through an array. **map** takes an input array and

[6] **extractor** is similar to an interaction found in Gneiss [4].
[7] **formatter** is similar to a (second) interaction found in Gneiss [4], and is also inspired by Yoshiki Schmitz's work [45].

[8] A more faithful adaptation of Natto, which could also be implemented with Engraft, would connect cells with wires rather than references.

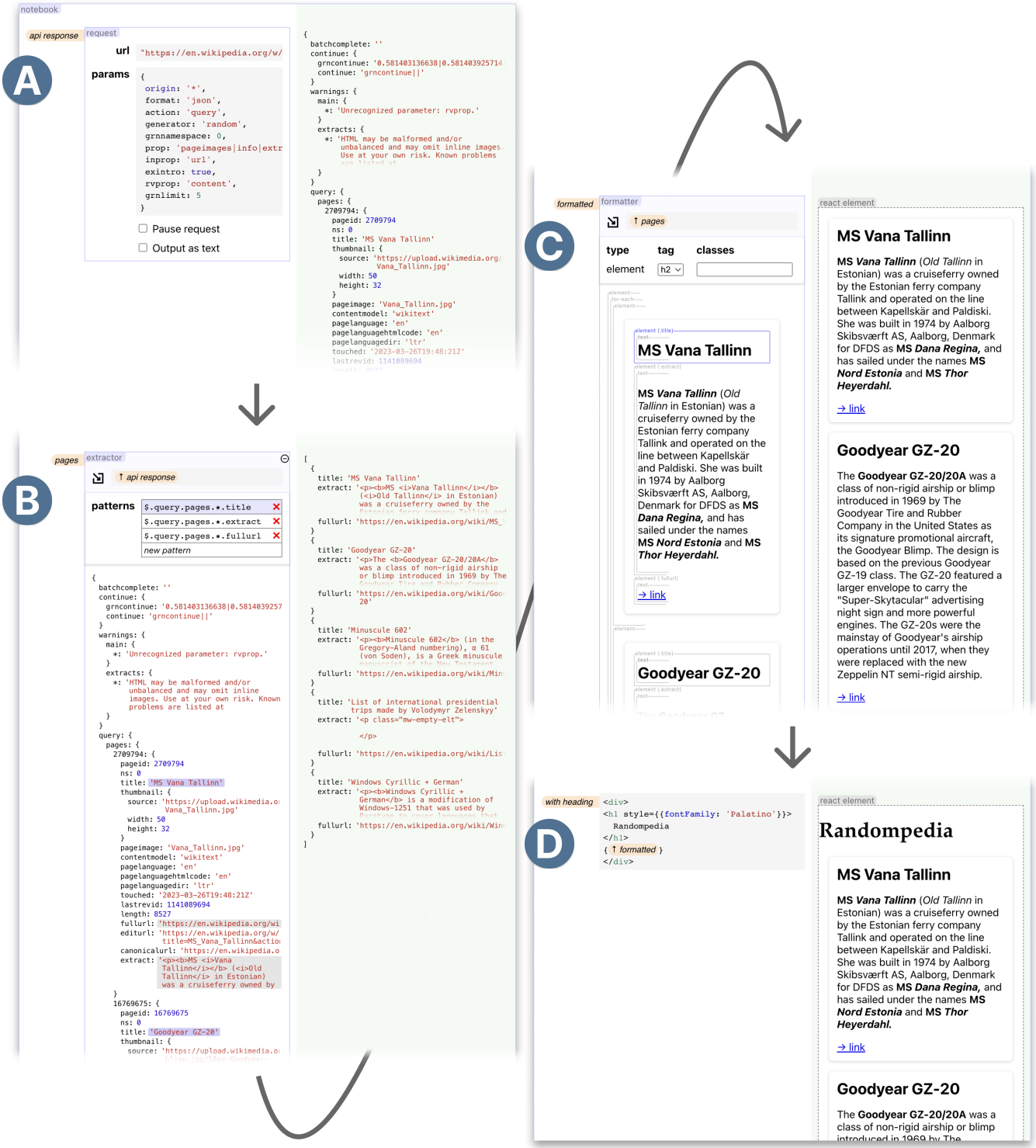


Figure 6: Graft Garden editing the completed "Randompedia" project. Four tools are composed together in a notebook to achieve the artist's goal: (A) request, (B) extractor, (C) formatter, and (D) slot (a traditional code editor).

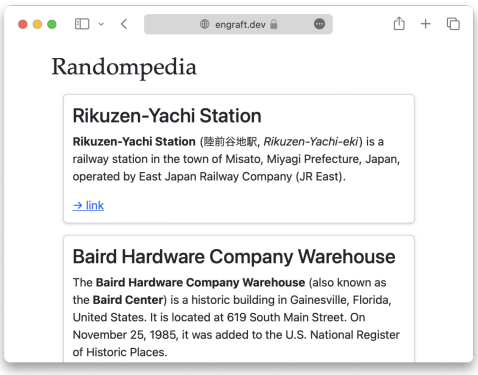


Figure 7: The artist's final creation: a dynamic website invisibly powered by their Engraft program.

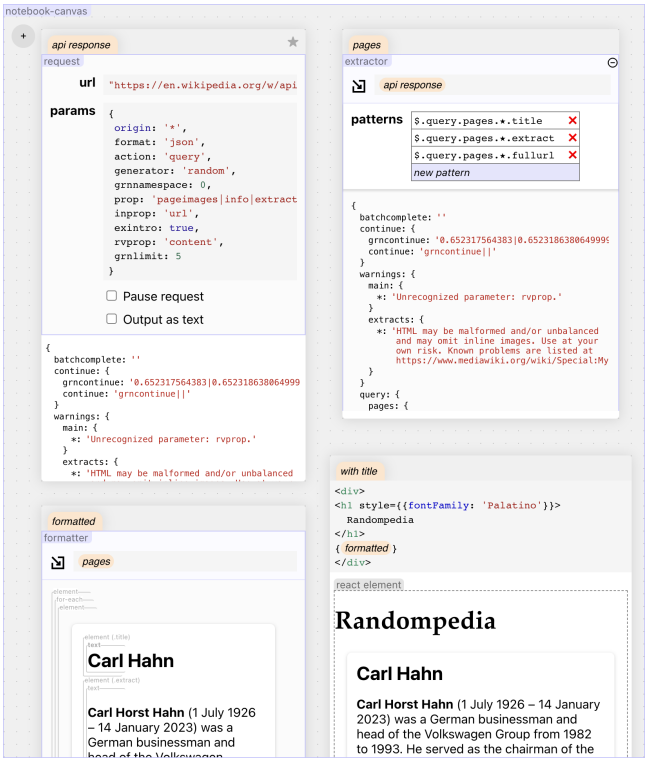


Figure 8: The tools from Figure 6, composed together in a notebook-canvas rather than a notebook.

provides a slot for a "per-item tool". **map** shows the per-item tool being run on a single element of the array, so that the user has concrete data to inform their experience of live-programming the per-item tool. The user can use **map**'s interface to select which element of the array they would like to use as their example, so they can test that their per-item tool performs well across variations. To compute its final output, **map** also runs a copy of the per-item tool on each element of the input array, though most of these executions are invisible. (This is a good example of why it is important that

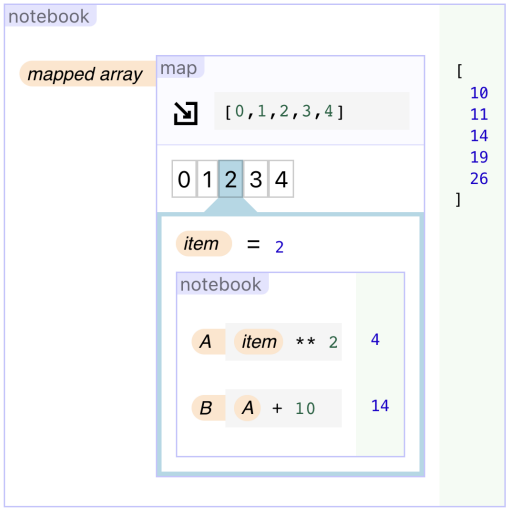


Figure 9: A toy example, showing `notebook` inside of `map` inside of `notebook`. `map` has been given the array `[0,1,2,3,4]` as input. Its inner notebook squares an item of an array and then adds 10, in two separate cells. The user has selected index 2 in the `map` tool as the example they would like to display in the inner notebook.

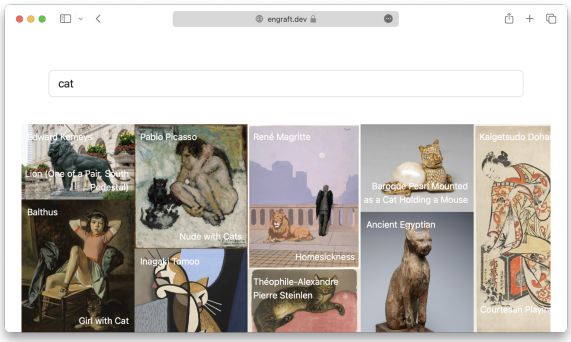


Figure 10: The image quilt application, displayed in Graft Garden's "view" mode. The images shown are a result of the user typing "cat" into the search box.

tools can run without their interfaces being rendered.) Figure 9 shows a notebook embedded inside of **map**, so that multiple stages in the per-item tool can be examined live as they are programmed.

To see **map** used in a real-world context, let's look at an "image quilt" generator made with Engraft (Figure 10). Starting with a user-supplied query, like "abstract" or "cat", this web application displays a dense array of annotated artworks.

The Engraft program behind this application starts by querying the Art Institute of Chicago API for matching works of art. This returns an array of objects representing works of art. To make the quilt, we need to turn each element of this array into a composite of image and text. We can do this with **map** (Figure 11).

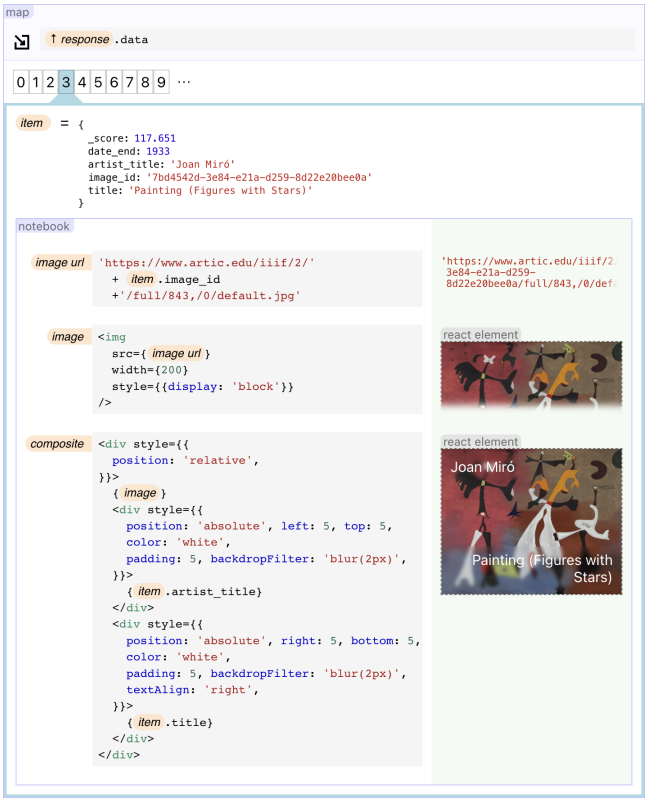


Figure 11: An excerpt of the program used to generate the image quilt. A map tool takes in an array of data returned by the Art Institute of Chicago API. A notebook embedded inside the map processes each element of this array. In three cells, it 1. constructs an image URL, 2. loads this URL into ean image element, to check that it is constructed correctly, and 3. builds a composite, layering text on top of the image element with appropriate styling.

By splitting the per-item process into multiple steps in the **notebook**, we receive immediate feedback about each step. Is the URL of the image being generated correctly? How does the composite look with its current styling? (One can certainly imagine this composition step being replaced someday with a direct-manipulation tool!) Once the array of HTML elements is returned by **map**, it can finally be composed into the quilt.

Programming is full of nested abstractions, so mapping an array is only one example of where it can be valuable to nest environments. As a very different example, consider programming a physics simulation. We are inspired here by the Bootstrap curriculum, which uses programming to teach algebra, physics, and computation to students in grades 5-12 [3]. We adopt a functional structure for our simulations similar to Bootstrap's "reactor" [37], where a simulation is defined by a *state* initialized to a certain value, an *on-tick* function which updates the state on each time step, and a *to-draw* function which visualizes the state.

A tool called **simulation** lets a user define each of these pieces in slots. It can be useful to insert a live environment into one of these

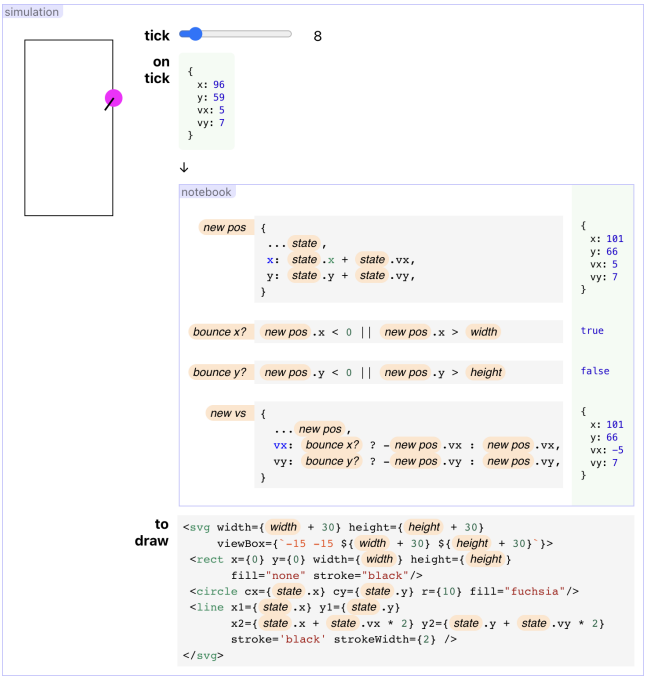


Figure 12: A simulation tool, loaded with code that describes how a ball bounces around a rectangular region. In this view, the tool's user has provided 1. *on-tick*: a slot (notebook, here) describing how the state should be updated on each tick of time, and 2. *to-draw*: a slot describing how the state should be rendered. The user has scrubbed the step slider to step 8, at which point the ball is bouncing off the rectangle's right-hand side.

slots – say, to break down *on-tick* into steps. Here, a **notebook** in the *on-tick* slot of a **simulation** describes a bouncing ball behavior (Figure 12).

By dragging the "tick" slider, the user can see the *on-tick* and *to-draw* behaviors in the context of that particular tick. Here, for instance, we see that the "bounce x?" cell in the notebook has evaluated to `true`, so the ball's x velocity will be inverted in the next step, as it bounces off the right-hand side of the box. Scrubbing through the ticks with the slider, the user can check that the pieces of their computation do what they expect, even as conditions change.

simulation benefits from embedding smaller components within it, like **notebook**. It also benefits from being embedded within larger components. In the example above, **simulation** is in fact embedded in a larger **notebook** (not shown). This larger notebook provides the shared variables "width" and "height" that the **simulation**'s different slots refer to. The **simulation** tool also provides output of its own back to the **notebook**: a trace of all the states the simulation passes through. This trace can be used, live, in other cells to analyze the output of the simulation. Here, we feed it into **voyager**, a tool that embeds the Voyager 2 [54] visual data exploration system. Using Voyager 2's interface, the user plots the x position over time (Figure 13). This visualization gives us an instantly-responsive higher-level vantage point on the behavior of the simulation. We

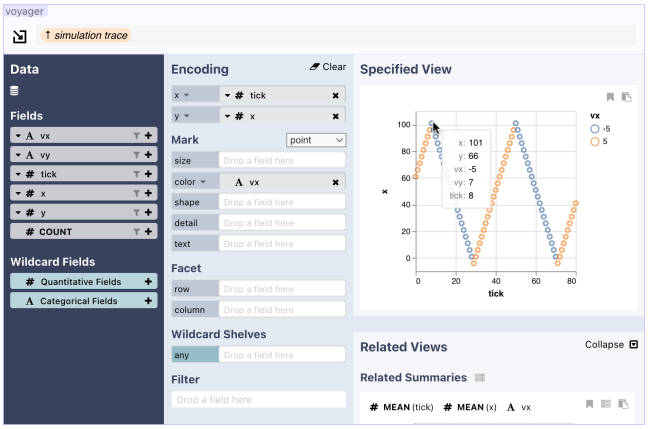


Figure 13: A voyager tool in the same notebook as the simulation. It has been provided with the simulation's output as its input, and the user has dragged fields onto the encoding shelves to plot "x" against "tick", with "vx" shown as a color.

do not expect that Voyager 2's creators anticipated this mode of use. Composability allows unanticipated uses to flourish.

There is nothing special about the example situations described above. Nested structures are pervasive in programming. To provide the benefits of liveness & richness across these nested structures, live environments must be similarly composable.

4.3 Tools and Environments In the Outside World

Given an Engraft **slot**, a programmer has access to the entire ecosystem of interoperable Engraft components. The question remains of how they get to that **slot** in the first place, and how the program in the **slot** gets things done in the larger world. Here, we discuss how the Engraft architecture makes it possible to use live & rich tools in contexts across the computational landscape, from codebases to the UNIX shell to applications.

4.3.1 Codebases. So far, we have presented Engraft in the context of Graft Garden, a simple web application that hosts Engraft tools and lets users create custom web applications. While Graft Garden is easy to access and use, it naturally has a limited range of usefulness. We do not expect developers of complex web applications to abandon their preferred frameworks, throw out their codebases, and switch to Graft Garden (or any other imagined Engraft host, for that matter). However, we still believe that programmers working in codebases could benefit from the judicious use of live tools and environments, if this didn't require switching entirely into a new, all-encompassing platform. Fortunately, we have found that the structure of the Engraft ecosystem offers opportunities for integration with present programming practices. With these integrations, programmers can take advantage of what Engraft has to offer in an unobtrusive and gradual fashion.

As an example of this, we have built an integration called `useEngraft` which allows a live tool to be embedded into a React codebase. At development time, `useEngraft` presents the Engraft user interface running alongside a live version of the web application being developed (Figure 14A). Data is fed, live, from the web application being developed into the Engraft user interface. The results are fed, live, back to the web application. When the developer is done working with the tool, they can save its program back to a JSON file in the codebase and disable the tool from being displayed (Figure 14B). In production, the "computational behavior" of the tool is used without any visual presentation – a user of the web application would not know Engraft was used to make it.

This is only one example of how Engraft could be embedded into existing codebases. Different situations will call for embeddings that work in different ways. For instance, someone writing server software may want to define a request handler with Engraft. Because the server runs imperatively, performing side effects and returning a single response to the client, it can not use `useEngraft`'s fully-reactive approach where the program re-runs as the user edits the function in Engraft. However, a "programming with examples" approach [27] could be employed, where the user gathers a number of input values for their function before iterating on their function's implementation, testing it on examples as they go.

While using an embedding like `useEngraft` is straightforward, building new embeddings of Engraft into new development contexts is not a trivial task. Embeddings must bridge gaps between a variety of programming paradigms and Engraft's own reactive model. The experience of using an embedding must also be carefully designed, as it is competing against refined and entrenched text-only workflows. However, we have found that the simple, functional structure of the Engraft API has made it adaptable to diverse embeddings.

4.3.2 UNIX Shell. The UNIX shell environment is a powerful and ecologically important programming system. As Jakubovic et al. [15] observe, the shell's power comes from composability: commands are often built up by piping data from process to process. While the composability of Unix tools is celebrated, the interfaces of command-line tools themselves receive less enthusiasm. Data-transformation commands like `sort`, `sed`, and `tr` are controlled by idiosyncratic languages of arguments which the user must navigate without guidance or immediate feedback.

Engraft can fit naturally into the UNIX shell's architecture, bringing liveness, data-visibility, and rich tools into everyday shell use. To explore this, we built a command-line tool which embeds Engraft into UNIX's network of processes, receiving and sending data over pipes.

Suppose a researcher wants to find large documents in their Zotero collection so they can move them to an external drive. In their shell, they navigate to the location of their collection. They then run:

```
find . -printf "'%P',%s\n" | engraft script.json --edit
```

The first part of this pipeline uses the UNIX `find` command to recursively search the current directory, printing the path and size of each file in a CSV-compatible format. The second part pipes the output of `find` into the Engraft command-line utility.

When this command runs, it guides the user to a URL served at `localhost` and waits for the researcher to continue their scripting with Engraft. Visiting the URL in their browser, the researcher finds an Engraft **notebook** loaded with the data produced by `find` and

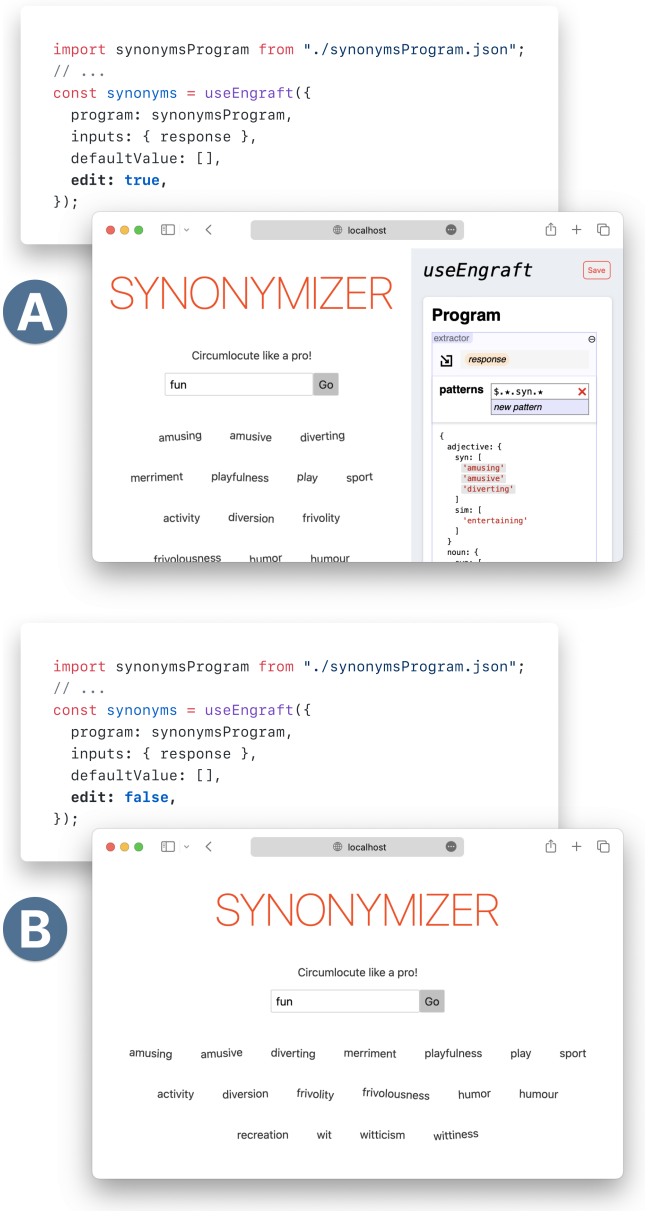


Figure 14: A conventional web application, Synonymizer, developed with `useEngraft`. (A) When `useEngraft` is called with `edit: true`, the live tool is displayed in the browser, to the right of the running application. The Engraft tool extractor is used in this side pane to transform an API response passed from the conventional code into the tool into a set of words that can be displayed on screen. The running app shows the live output from the tool. (B) Once `edit` is changed to `false`, the browser no longer shows the Engraft interface. The Engraft program runs invisibly, and the application is ready to be deployed publicly.

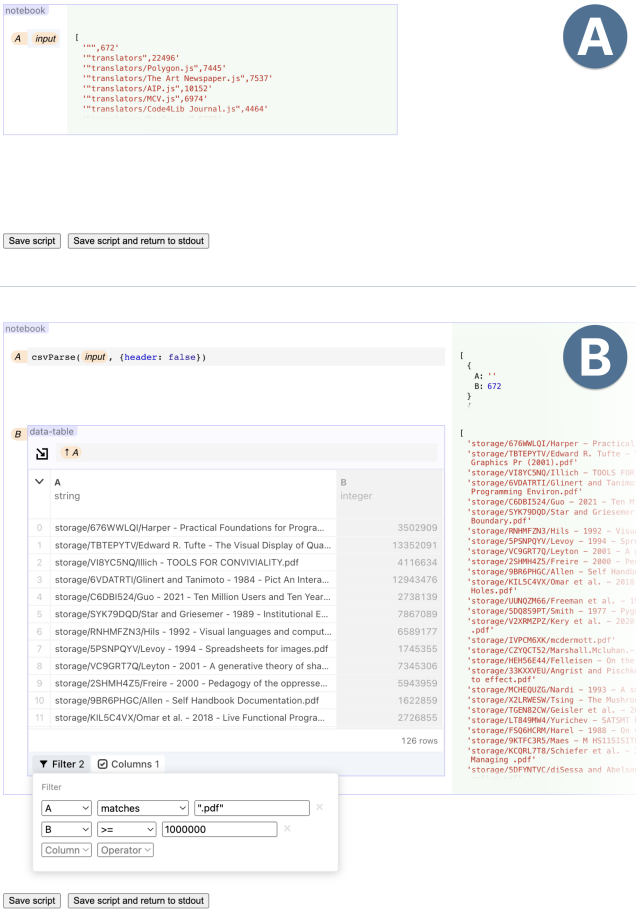


Figure 15: The editing interface launched by the Engraft command-line tool. (A) is the view first seen by the user – raw string input displayed in a notebook. (B) is the view after the user edits the data in the data-table tool.

piped into `engraft`, as shown in Figure 15A. They parse the CSV in one cell. In another, they load the parsed data into **data-table**, a general-purpose tool for viewing and editing tabular data inspired by Observable's data table cell [30]. In this tool, the researcher adds filters to select the files they are looking for and disables the column of sizes so that only the file paths will be output back to the shell. Their final Engraft script is shown in Figure 15B.

When they press the "Save script and return to stdout" button, their script is saved to `script.json`, which they specified in their original command. At their shell, they will see a list of paths returned back from Engraft. It looks good, so they run:

```
find . -printf '"%P",%s\n' | engraft script.json
  | xargs -d\\n -I % mv "%" /external-drive
```

Here, the `-edit` flag is removed from the call to `engraft`. This means that, rather than launching a web interface, `engraft` runs the script it is passed in automatically ("headlessly"). The filtered list of files it outputs is piped onwards to another UNIX tool that moves the files to the external drive.

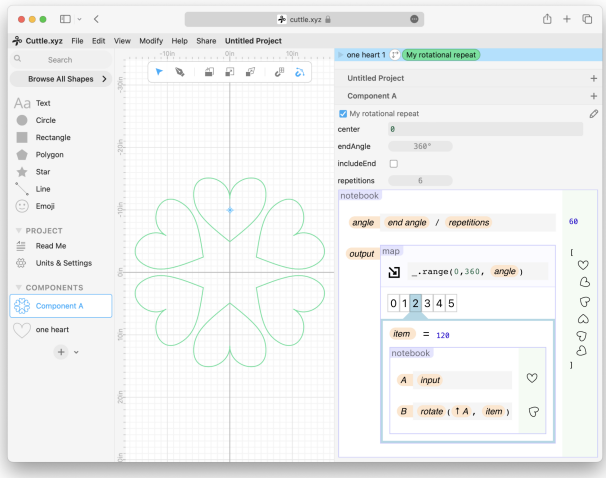


Figure 16: A mockup of an imagined embedding of Engraft into Cuttle. The Engraft program defines a modifier in Cuttle which transforms an input heart shape into an array of rotated hearts.

It is unusual to see the same interactive tools used across contexts as varied as computational notebooks, web application development, and command-line scripts – though traditional programming languages often cross these boundaries. Engraft aims to level the playing field between interactive tools and static text, bringing the benefits of broad applicability to live & rich tools.

4.3.3 Applications. We are also interested in ways interactive end-user applications can host Engraft tools. We have already seen one example: Graft Garden is a simple web-application which hosts Engraft tools and makes their output available as web pages. But Graft Garden is just one instance of an application host. By providing an Engraft **slot**, any application built on the web platform can tap into the Engraft ecosystem.

For instance, Cuttle [5], a vector editor for digital fabrication, currently has the ability to implement components and modifiers with bits of JavaScript code. If Cuttle's code box were replaced with an Engraft slot, the world of Engraft programming tools would be available, in-place, to Cuttle users. (Figure 16)

One can imagine applications farther from the world of programming using Engraft to provide open-ended extensions of their own interfaces. An animation application's easing-function editor and color picker could be implemented as Engraft slots prepared with default tools. Experienced users could then choose to remove these defaults and replace them with their own tools. In this way, Engraft could enable end-user customization, blurring the lines between application user and developer.

5 IMPLEMENTATION

We now present implementation details of the Engraft prototype. We begin with a slightly idealized description of the Engraft API, before explaining an important deviation from this ideal, then go on to discuss the Engraft component ecosystem.

5.1 The Engraft API

The core of an Engraft component is a function called `run`. As we discuss further in §5.2, performant interactivity requires augmenting this function with a system for incremental evaluation, but for the remainder of this section we will put this complication aside and focus on the pure semantics of this function.

A component's `run` function takes in two "props" as input:

- `program`: The component's program, a serializable JavaScript object that defines its behavior.
- `varBindings`: Bindings of variable ids & names to promised values.

In return, the component synchronously returns two "results":

- `outputP`: A promise of a value.
- `view`: A representation of how to render the component.

This short description captures the basic structure of the Engraft API. We discuss important details in the following subsections. For the definitive details, please refer to *@engraft/core* in the Engraft source repository [52], which provides TypeScript types for the Engraft API.

5.1.1 Views. The view returned by `run` gives the component's host everything it needs to display an interactive interface for the component. The most important part of the view is a function called `render` which, when given a few important parameters, returns a React node for the component's view.[9] The most important parameter `render` receives is `updateProgram`, a function which can be called to transform the component's program according to an updater function. As a user interacts with a rendered view, their interactions will produce calls to `updateProgram`, which will change an underlying program, triggering a re-run of the component's `run` function and producing a new view. This is the model-view loop of interactive program editing in Engraft.

5.1.2 Asynchronicity. Engraft components can produce output values synchronously or asynchronously. For instance, components which make network calls or perform expensive computations will only be able to produce output asynchronously. We use promises to represent values that may not be synchronously available. However, built-in JavaScript promises are *necessarily* asynchronous, meaning that even if they are resolved synchronously, this value will not be available to the promise's users until a future tick. To support both synchronous and asynchronous output, we adapt an alternative promise library called `synchronous-promise` [25] for promises in the Engraft ecosystem.[10] Because a component's output can be asynchronous, and may feed into another component, we also allow the `varBindings` coming into a component to include promised values. This supports fine-grained handling of asynchronicity by components: a component can start a computation on one input even while another is pending.

[9] Using React to implement views is an expedient compromise of Engraft's general design approach of remaining unopinionated about component implementation. However, as adapters are available between React and many other frameworks, this does not reduce flexibility of implementation as much as it may seem.

[10] This is almost equivalent to an alternative standard in which a component produces either a value or a promise. "Synchronous promises" have two advantages over this alternative: 1. they can represent both returned values and thrown errors, and 2. they present a more uniform API to consumers.

While output values can be deferred, a component's resulting view is always returned immediately. Even if a component takes time to run, it must provide an interface to show the user while it is running. This view can listen to promises involved in the component's computation to update its display while computation progresses.

5.1.3 Error Handling. While output values are wrapped in promises primarily to support asynchronous components, promises also conveniently provide support for error handling. If a component throws an error during computation of its output, this error will be held by its output promise. It can then be displayed in an Engraft component's UI, providing helpful feedback to the programmer rather than crashing the system.

5.1.4 Additional Parts of a Component. While the majority of a component's definition resides in its `run` function, a few other bits of information are required from a component. The first is a function called `makeProgram`, which is used to initialize a program for a new instance of the component. To support the common pattern of a component having a "main input" which an environment may want to pre-populate, `makeProgram` optionally accepts a string `defaultInputCode`. This allows ergonomic interactions, such as new cells in a **notebook** automatically taking in the previous cell as input. The second is a function called `computeReferences`, which takes in one of the component's programs and returns the set of "free" variables referred to by the program (that is, the variables the component wants its host to pass in as `varBindings`). Components which manage data-flow between their children, such as **notebook**, use `computeReferences` to construct a dependency graph.

These pieces, together with `run`, are bundled by a component's developer into a single JavaScript object (TypeScript type `Component`) which is exported by a module for use.

5.2 Incremental Computation in Engraft

The architecture described in the last section provides a simple functional API for Engraft: a component's output and view are pure functions of its program and inputs. However, implemented directly, this scheme would entail unacceptable computational costs. As an illustration: Every time a cell is added to a notebook, or a key is pressed in a code editor inside a notebook, the notebook's underlying program changes. This means we must re-run the notebook's `run` function on the new program. If each such run started from scratch, each would have to recompute all the cells in the notebook, even those manifestly unaffected by the change that prompted it.

This is clearly not a scalable approach. What we need is a way for each run of the notebook (or any other component) to build intelligently on previous runs, re-using old work when possible rather than running everything from scratch. This feature is known as incremental computation [38].[11] Engraft's situation, in which a tree of components must be incrementally maintained as changes occur, is reminiscent of React [26], a library for building user interfaces. Inspired by the way React performantly maintains its functionally-defined tree of components, we have built a novel library for incremental computation named Refunc.

[11] Incremental computation is closely related to reactive and dataflow programming [2].

Refunc is a general library for functional incremental computation, used by Engraft but not otherwise tied to it. The external API of Refunc is minimalist, consisting solely of the concept of a refunction: a function that is pure except that it can read and write to a "memory" provided to it as an extra argument. Using this memory, the function can store information on intermediate results between runs. What is stored in the memory, and how it is accessed by the refunction, is up to the refunction's implementation.

In Engraft, a component's `run` function is implemented as a refunction rather than a bare JavaScript function. Components can use their memories to memoize their entire computation, to memoize costly sub-computations (like compiling textual code to a JavaScript function), and, recursively, to memoize the memories of sub-components they embed in slots.

What actually happens when a **notebook** is edited in Engraft? When a cell is added, the **notebook**'s program changes. But the **notebook** has stored the memories of all its cells' computations in its own memory. It can immediately detect that these cells haven't changed between the new and old programs[12] and re-use their old values and views. By using Refunc, Engraft can expose a simple functional model while still maintaining the performance necessary for live program editing.

5.2.1 Hooks. Refunc is exposed as part of the Engraft API, so, in keeping with our general design principles, we have kept the Refunc API minimal and unopinionated. However, in practice component-makers will not want to directly manage memories in an ad-hoc way, and will instead want to use principled memoization primitives. Inspired by React's "hooks" system, we have implemented an analogous, optional, system for Refunc. Our hooks system provides a convenient way to write functions that use memoization primitives without having to explicitly manage a refunction's memory.

To support the use-cases we ran into developing Engraft components, we extended Refunc's hooks system beyond React's in several ways, including supporting keyed groups of hooks and allowing hooks to run asynchronously. With these extensions, we have been able to express sophisticated incremental computation patterns. For more on Refunc, we refer you to the README in the *@engraft/refunc* package in the Engraft source repository [52].

5.3 Slots and the Component Tree

Throughout this paper, we have shown Engraft programmers using "slots" to compose nested programs. The **slot** component is the glue that holds together Engraft programs. It is a built-in component that appears at first as a code editor. Arbitrary JavaScript can be entered into this code editor, where it is compiled, evaluated, and returned as output. References to Engraft variables can be inserted into this editor using an auto-complete window. If a component's name is selected via auto-complete, the slot will be replaced with the component, entering "host mode" (Figure 17).

When **slot** renders a hosted component, it provides it with a "component frame" to identify it. On hover, the title bar of this frame also reveals a few buttons (Figure 17): "cp", which copies the component's program to the clipboard so it can be pasted into a

[12] When Engraft programs are edited, "structural sharing" is employed to re-use unchanged portions of the program. This means that detecting that a cell hasn't changed can be done with a constant-time reference-equality check.

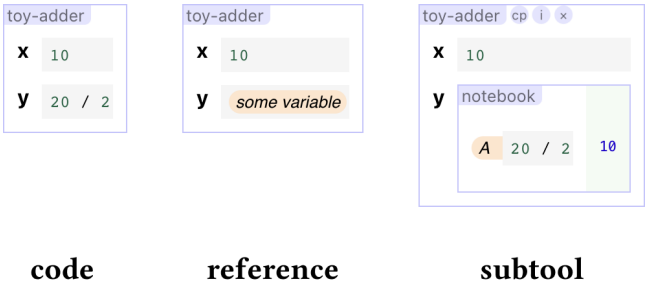


Figure 17: A simple toy-adder component. Its "y" slot is provided with (a) a code snippet, (b) a reference to a variable provided by the toy-adder's host, (c) a nested notebook. In (c), the toy-adder's title bar is hovered, revealing three circular buttons.

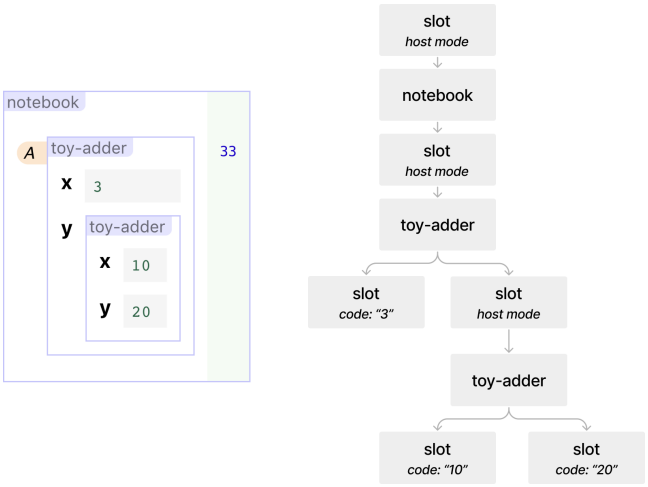


Figure 18: An arrangement of an toy-adder nested in an toy-adder nested in a notebook, together with a diagram of the tree of components resulting from this arrangement. Note the slot components, which provide the code-editors shown as leaves of the tree, as well as invisible intermediates between components and their sub-components.

different location, "i", which displays a pop-up debugger window helpful for component development, and "x", which removes the hosted component and brings the slot back into code mode. We anticipate this frame will support additional general-purpose interactions in the future, such as "maximizing" a component for focused work or "pinning" it to a sidebar.

Through the use of slots, a "tree of components" is formed during the use of Engraft. Figure 18 shows an example arrangement of components together with a diagram of the resulting tree. Note, however, that there is no explicit reference to a tree of components anywhere in the Engraft API. Rather, this tree emerges from the fact that a component, like any other software system, is free to act as a host that embeds other components.

slot plays an essential role in the Engraft user experience. While the Engraft API provides the computational structures needed for open-ended composition, **slot** provides users with an interface that makes that composition accessible in practice.

5.4 Component Creation and Distribution

An Engraft component is implemented as a JavaScript object adhering to a certain interface, as described in §5.1. This technical description raises some important questions.

5.4.1 How are components made? The Engraft API is low-level. Building a component directly with this API gives a developer full control of all aspects of component-host communication, including details such as asynchronicity and incremental computation. However, attending to these details can be complex, even for simple components which do not use these features in distinctive ways.

We believe the best way to deal with this problem is to build higher-level layers on top of the Engraft API while maintaining the underlying API for those who need its full power. To experiment with this approach, we made `defineSimpleComponent`, a function which lets programmers implement components at a higher level than the Engraft API, provided their components satisfy certain simplifying assumptions. A simple **toy-adder** component that adds the contents of its two slots, which takes 60 lines of code to implement directly, can be made with `defineSimpleComponent` in 25 lines.

So far, Engraft components have been made with conventional textual code, whether low- or high-level. The question naturally arises as to how Engraft components could be made in a live, rich environment, perhaps within Engraft itself. For instance, a user could build a composite of Engraft components (e.g., a Wikipedia-specific API tool built in a computational notebook) and then select parts of this composite to expose in a custom tool interface. Or a user could take an existing web-based tool, and then describe how it could be embedded into an `iframe` and driven to make it participate in the Engraft API. Meta-tools like these could make tool creation part of the process of tool use, empowering end-users and end-user communities [28].

5.4.2 How are components accessed by Engraft users? In our work on Engraft so far, we largely side-step this question. All components are either bundled together with the Engraft system, or injected by a specific host for use within that host. This is a stop-gap solution. Ultimately, it will be important for third-party component creators to be able to post components publicly and for users to be able to access them easily from any host. One convenient approach would be to leverage existing package managers like NPM.

5.5 Components Built So Far

To date, we have implemented twenty-five components with Engraft. The examples in §4 highlight some of these: **notebook**, **request**, **extractor**, **formatter**, **notebook-canvas**, **map**, **simulation**, **voyager**, and **data-table**. Appendix A includes pictures and brief descriptions of 10 more components. All of these components are "sketches" intended to test the Engraft API's design and demonstrate its combinatorial possibilities. We expect that as Engraft finds use with more communities, tool-makers will contribute both more polished and more radically divergent components back into the Engraft ecosystem.

6 HEURISTIC ANALYSIS

While the examples in §4 demonstrate how Engraft enables promising new workflows, more systematic evaluation is necessary to characterize the limits of Engraft's approach. Towards that end, this section analyzes Engraft according to the "Technical Dimensions of Programming Systems" (TDPS) taxonomy introduced by Jakubovic et al. [15]. We focus on dimensions which reveal strengths and weaknesses of Engraft's design, as well as those which suggest comparisons to existing systems (complementing our discussion in §2). Dimensions from TDPS are printed in bold.

Feedback Loops (TDPS §4.1.1) asks how a system minimizes the gulfs of evaluation and execution. These gulfs correspond to two of Engraft's motivating qualities: *liveness* narrows the gulf of evaluation with immediate feedback (TDPS §4.1.2) while *richness* narrows the gulf of execution with direct manipulation (TDPS §4.1.3). **Modes of Interaction** (TDPS §4.1.4) further asks how the system stages access to different forms of interaction at different times. Like the example of Jupyter notebooks provided by TDPS, interactions with Engraft itself are un-staged – there are not separate editing and debugging modes, but rather a single live mode. Embedding Engraft into another system may necessitate moving between different modes, such as moving between editing a shell script that calls an Engraft program and editing the Engraft program itself. The usability of an embedding depends critically on how it supports movements between modes like this.

Notational Structure (TDPS §4.2.1): Like Boxer [7], Engraft uses "complementing notations" (TDPS §4.2.3). While Boxer uses two forms of notation (code and boxes), Engraft has an extensible notational system whereby new components can introduce arbitrary new notations. This extensibility enables highly customized programming experiences not possible within Boxer (e.g. **formatter** and **voyager**), but also makes Engraft's notation less uniform (see **Uniformity of Notations**, TDPS §4.2.10). This heterogeneity comes with risks: it might be challenging for users to learn new components, or uncomfortable for users to work in a space that ties together discordant interfaces. Furthermore, because every Engraft component defines its own program specification, Engraft does not easily support "overlapping notations": parallel representations for the same program (TDPS §4.2.2). This contrasts with Sketch-n-Sketch's "bidirectional" approach [12], in which direct-manipulation interfaces edit general-purpose code so multiple interfaces can edit the same underlying program.

Composability (TDPS §4.3.4) asks how primitives can "be combined to achieve novel behaviors." This question, applied to live & rich tools, is the impetus behind Engraft. Engraft composes live & rich tools within live environments, and goes further than systems supporting only "flat" composition (e.g., mage [18]) by supporting "nested" composition (see our §4.2). However, the composition provided by Engraft is limited in some ways. For instance, it is natural to want to "unbundle" the three panes of Gneiss [4] as three separate Engraft components, as we alluded to in our §2.2. However, cyclic data flows exist between these panes which cannot be precisely recreated in Engraft's functional-programming model. Furthermore, Gneiss uses bespoke drag-and-drop interactions between panes, while Engraft components' UIs cannot interact with one another.

Self-Sustainability (TDPS §4.4.3) is the ability to modify a programming system from within itself, in contrast to the typical separation of "user level" and "implementation level". Engraft does not prioritize self-sustainability. At present, Engraft components are created and modified outside of Engraft with a traditional JavaScript software-engineering process. This stands in contrast to systems like Smalltalk [11] and Webstrates [19] which host their own development environments. These approaches may be complementary; embedding Engraft into Webstrates is a possible future direction. Alternatively, meta-tools could bring component-building into Engraft itself, as discussed in our §5.4.1.

Learnability of Engraft (TDPS §4.7.1) is as-yet untested. Certain aspects of Engraft might contribute to learnability: Live feedback lets users try things out and learn from immediate responses. Rich components let users use familiar direct-manipulation interactions to build programs, rather than abstract syntax. The uniform behavior of slots offers a conceptually simple model for composition. Other aspects might detract: It may be burdensome to learn a large variety of tools, or it may be hard to discover the right component to use when a task is first encountered. Engraft's approach to **Sociability** (TDPS §4.7.2) is centered around its integration with existing technology stacks, allowing "incremental adoption". This occurs in two ways: embedding existing systems into Engraft (as seen with **voyager** in our §4.2) and embedding Engraft into existing systems (as seen in several ways in our §4.3). However, Engraft's departures from conventional programming practices like textual code limits its integration with some sociotechnical systems. For instance, while an Engraft program can be written to a text file and checked into a version control system like git, programmers would need to work directly with unfamiliar JSON representations in order to resolve merge conflicts. Recent work has sought to extend version control to non-textual coding interfaces [9], but this is still an area of active research.

We have found this analysis helpful for validating Engraft's strengths, identifying its possible weaknesses, and mapping out its position in the broader space of programming systems. We are especially excited by future directions the analysis suggests, such as extending Engraft's UI model to support richer interactions between components and embedding Engraft into self-sustaining platforms where tool development can be closer to tool use.

7 CONCLUSION

Our starting point in this paper is a simple observation: Despite efforts from industry and academia spanning decades, live & rich programming tools are not a mainstream part of programming practice. To our knowledge, this fact has not found much discussion in the literature.[13] We believe that, given the potential benefits live & rich programming might offer, this gap should be confronted head-on.

In this work, we take on the necessary challenge of *composing live & rich tools*. We further articulate three distinct forms of composition that must be supported in order for live & rich programming

[13]Tanimoto's 2013 reflections on live systems [51] includes a section on "Criticisms of Liveness", though he quickly dismisses them in favor of a generally optimistic view. Lau's short piece [21] on why programming-by-demonstration systems sometimes fail is insightful, but it focuses on AI-specific aspects and does not apply to live & rich tools in their full breadth.

to rival entrenched textual alternatives: *tools in environments*, *environments in environments*, and *tools and environments in the outside world*. We achieve these forms of composition with Engraft, an API for the web platform that enables recursive embedding of components within hosts. Engraft demonstrates that composition along these three forms is possible while serving as a foundation for further explorations.

Engraft is ready for use by other researchers in our community. In particular, we hope that Engraft can serve as an enabling platform for researchers developing experimental live & rich tools. Novel tools built as Engraft components can more easily provide value and be validated, as they can be used with all the tools, environments, and outside-world integrations that already exist in the Engraft ecosystem The Engraft codebase, with accompanying documentation, is hosted at https://github.com/engraftdev/engraft [52]. It contains the libraries necessary for implementing new tools and hosts, as well as implementations of the tools and hosts discussed in this paper. We invite researchers interested in building on Engraft to contact us to discuss possible collaborations.

ACKNOWLEDGMENTS

We thank Ken Gu, Eunice Kim, Geoffrey Litt, Edward Misback, Josh Pollock, the UW Interactive Data Lab, and the anonymous reviewers for their valuable feedback. This work was supported by a Moore Foundation software grant.

REFERENCES

[1] Leif Andersen, Michael Ballantyne, and Matthias Felleisen. 2020. Adding interactive visual syntax to textual code. *Proceedings of the ACM on Programming Languages* 4 (nov 13 2020), 1–28.

[2] Engineer Bainomugisha, Andoni Lombide Carreton, Tom van Cutsem, Stijn Mostinckx, and Wolfgang de Meuter. 2013. A survey on reactive programming. *Comput. Surveys* 45, 4 (8 2013), 1–34.

[3] Bootstrap. 2022. Bootstrap. https://bootstrapworld.org/index.shtml.

[4] Kerry Shih-Ping Chang. 2016. A Spreadsheet Model for Using Web Services and Creating Data-Driven Applications. *Carnegie Mellon University* (2016).

[5] Cuttle Labs Inc. 2022. Cuttle - Design tool for digital cutting machines. https://cuttle.xyz/.

[6] Allen Cypher (Ed.). 1994. *Watch what I do: Programming by demonstration.* The MIT Press.

[7] A. A diSessa and H. Abelson. 1986. Boxer: a reconstructible computational medium. *Commun. ACM* 29, 9 (9 1986), 859–868.

[8] Jonathan Edwards. 2005. Subtext. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications.* ACM.

[9] Jonathan Edwards and Tomas Petricek. 2021. Typed Image-based Programming with Structure Editing. Presented at Human Aspects of Types and Reasoning Assistants (HATRA'21).

[10] Kasra Ferdowsifard, Allen Ordookhanians, Hila Peleg, Sorin Lerner, and Nadia Polikarpova. 2020. Small-Step Live Programming by Example. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology.* ACM.

[11] Adele Goldberg and David Robson. 1983. *Smalltalk-80: the language and its implementation.* Addison-Wesley Longman Publishing Co., Inc.

[12] Brian Hempel, Justin Lubin, and Ravi Chugh. 2019. Sketch-n-Sketch. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology.* ACM.

[13] Joshua Horowitz. 2018. PANE: Programming with Visible Data. (2018). Presented at the Workshop on Live Programming (LIVE) 2018.

[14] Joshua Horowitz and Jeffrey Heer. 2023. Live, Rich, and Composable: Qualities for Programming Beyond Static Text. (2023). Presented at the 13th annual workshop on the intersection of HCI and PL (PLATEAU 2023).

[15] Joel Jakubovic, Jonathan Edwards, and Tomas Petricek. 2023. Technical Dimensions of Programming Systems. *The Art, Science, and Engineering of Programming* 7, 3 (feb 15 2023).

[16] JetBrains s.r.o. 2022. MPS: The Domain-Specific Language Creator by JetBrains. https://www.jetbrains.com/mps/.

[17] Sean Kandel, Andreas Paepcke, Joseph Hellerstein, and Jeffrey Heer. 2011. Wrangler. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems.* ACM.

[18] Mary Beth Kery, Donghao Ren, Fred Hohman, Dominik Moritz, Kanit Wongsuphasawat, and Kayur Patel. 2020. mage: Fluid Moves Between Code and Graphical Work in Computational Notebooks. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology.* ACM.

[19] Clemens N. Klokmose, James R. Eagan, Siemen Baader, Wendy Mackay, and Michel Beaudouin-Lafon. 2015. *Webstrates.* In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology.* ACM.

[20] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian E. Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica B. Hamrick, Jason Grout, Sylvain Corlay, Paul Ivanov, Damián Avila, Safia Abdalla, Carol Willing, and Jupyter Development Team. 2016. Jupyter Notebooks - a publishing format for reproducible computational workflows. In *International Conference on Electronic Publishing.*

[21] Tessa Lau. 2009. Why Programming-By-Demonstration Systems Fail: Lessons Learned for Usable AI. https://doi.org/10.1609/aimag.v30i4.2262. *AI Mag.* 30, 4 (2009), 65–67.

[22] Sorin Lerner. 2020. Projection Boxes: On-the-fly Reconfigurable Visualization for Live Programming. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems.* ACM.

[23] Jens Lincke, Robert Krahn, Dan Ingalls, and Robert Hirschfeld. 2009. Lively Fabrik A Web-based End-user Programming Environment. In *2009 Seventh International Conference on Creating, Connecting and Collaborating through Computing.* IEEE.

[24] Eyal Lotem and Yair Chuchem. 2022. Lamdu. https://www.lamdu.org/.

[25] Davyd McColl. 2023. synchronous-promise. https://github.com/fluffynuts/synchronous-promise.

[26] Meta Platforms, Inc. 2022. React – A JavaScript library for building user interfaces. https://reactjs.org/.

[27] B. A. Myers. 1986. Visual programming, programming by example, and program visualization: a taxonomy. In *Proceedings of the SIGCHI conference on Human factors in computing systems - CHI '86.* ACM Press.

[28] Bonnie A Nardi. 1993. *A small matter of programming.* MIT Press, London, England.

[29] Observable Inc. 2022. Observable - Explore, analyze, and explain data. As a team. https://observablehq.com/.

[30] Observable Inc. 2022. Quickly Explore and Analyze Your Data With Data Table Cell / Observable / Observable. https://observablehq.com/@observablehq/introducing-data-table-cell.

[31] Observable Inc. 2023. Sample datasets / Observable / Observable. https://observablehq.com/@observablehq/sample-datasets.

[32] Cyrus Omar, David Moon, Andrew Blinn, Ian Voysey, Nick Collins, and Ravi Chugh. 2021. Filling typed holes with live GUIs. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation.* ACM.

[33] Cyrus Omar, Ian Voysey, Ravi Chugh, and Matthew A. Hammer. 2019. Live functional programming with typed holes. *Proceedings of the ACM on Programming Languages* 3 (jan 2 2019), 1–32.

[34] Cyrus Omar, Young Seok Yoon, Thomas D. LaToza, and Brad A. Myers. 2012. Active code completion. In *2012 34th International Conference on Software Engineering (ICSE).* IEEE.

[35] Stephen Oney, Brad Myers, and Joel Brandt. 2014. InterState. In *Proceedings of the 27th annual ACM symposium on User interface software and technology.* ACM.

[36] Everest Pipkin. 2021. been having some motivation troubles recently (god who hasn't) so i'm gonna pick a tiny personal project off my ideas list and see if i can get it working by morning. tonight - a lil bash script that emails me the summaries of 5 random wikipedia articles each morning. https://twitter.com/everestpipkin/status/1349274983651012609.

[37] Joe Politz, Benjamin Lerner, Sorawee Porncharoenwase, and Shriram Krishnamurthi. 2019. Event Loops as First-Class Values: A Case Study in Pedagogic Language Design. *The Art, Science, and Engineering of Programming* 3, 3 (feb 1 2019).

[38] W. Pugh and T. Teitelbaum. 1989. Incremental computation via function caching. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '89.* ACM Press.

[39] Pure Data. 2023. PD community site. https://puredata.info/.

[40] David Rauch, Patrick Rein, Stefan Ramson, Jens Lincke, and Robert Hirschfeld. 2019. Babylonian-style Programming: Design and Implementation of an Integration of Live Examples into General-purpose Source Code. *The Art, Science, and Engineering of Programming* 3, 3 (feb 1 2019).

[41] Patrick Rein, Stefan Ramson, Jens Lincke, Robert Hirschfeld, and Tobias Pape. 2018. Exploratory and Live, Programming and Coding. *The Art, Science, and Engineering of Programming* 3, 1 (jul 23 2018).

[42] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and Yasmin Kafai. 2009. Scratch. *Commun. ACM* 52, 11 (11 2009), 60–67.

[43] Arvind Satyanarayan and Jeffrey Heer. 2014. Lyra: An Interactive Visualization Design Environment. *Computer Graphics Forum* 33, 3 (6 2014), 351–360.
[44] Arvind Satyanarayan, Dominik Moritz, Kanit Wongsuphasawat, and Jeffrey Heer. 2017. Vega-Lite: A Grammar of Interactive Graphics. *IEEE Transactions on Visualization and Computer Graphics* 23, 1 (1 2017), 341–350.
[45] Yoshiki Schmitz. 2019. I've been jamming on this concept for making data-driven designs... https://twitter.com/yoshikischmitz/status/1176642448077967362.
[46] Paul Shen. 2021. Show HN: Natto – a canvas for writing and manipulating JavaScript. https://news.ycombinator.com/item?id=26478548.
[47] Paul Shen. 2022. welcome! – natto. https://natto.dev/.
[48] Vlad Shilov. 2023. react-colorful. https://omgovich.github.io/react-colorful/.
[49] Frank M. Shipman and Catherine C. Marshall. 1999. Formality Considered Harmful: Experiences, Emerging Themes, and Directions on the Use of Formal Representations in Interactive Systems. *Computer Supported Cooperative Work (CSCW)* 8, 4 (12 1999), 333–352.
[50] Steven L. Tanimoto. 1990. VIVA: A visual language for image processing. *Journal of Visual Languages and Computing* 1, 2 (6 1990), 127–139.
[51] Steven L. Tanimoto. 2013. A Perspective on the Evolution of Live Programming. In *Proceedings of the 1st International Workshop on Live Programming (LIVE '13)*. IEEE Press, San Francisco, California, 31–34.
[52] Engraft team. 2023. Engraft. https://github.com/engraftdev/engraft.
[53] Wikipedia contributors. 2022. Grafting — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Grafting. [Online; accessed 01-September-2022].
[54] Kanit Wongsuphasawat, Zening Qu, Dominik Moritz, Riley Chang, Felix Ouk, Anushka Anand, Jock Mackinlay, Bill Howe, and Jeffrey Heer. 2017. Voyager 2. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*. ACM.
[55] Haijun Xia, Bruno Araujo, Tovi Grossman, and Daniel Wigdor. 2016. Object-Oriented Drawing. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*. ACM.

A COMPONENT MENAGERIE

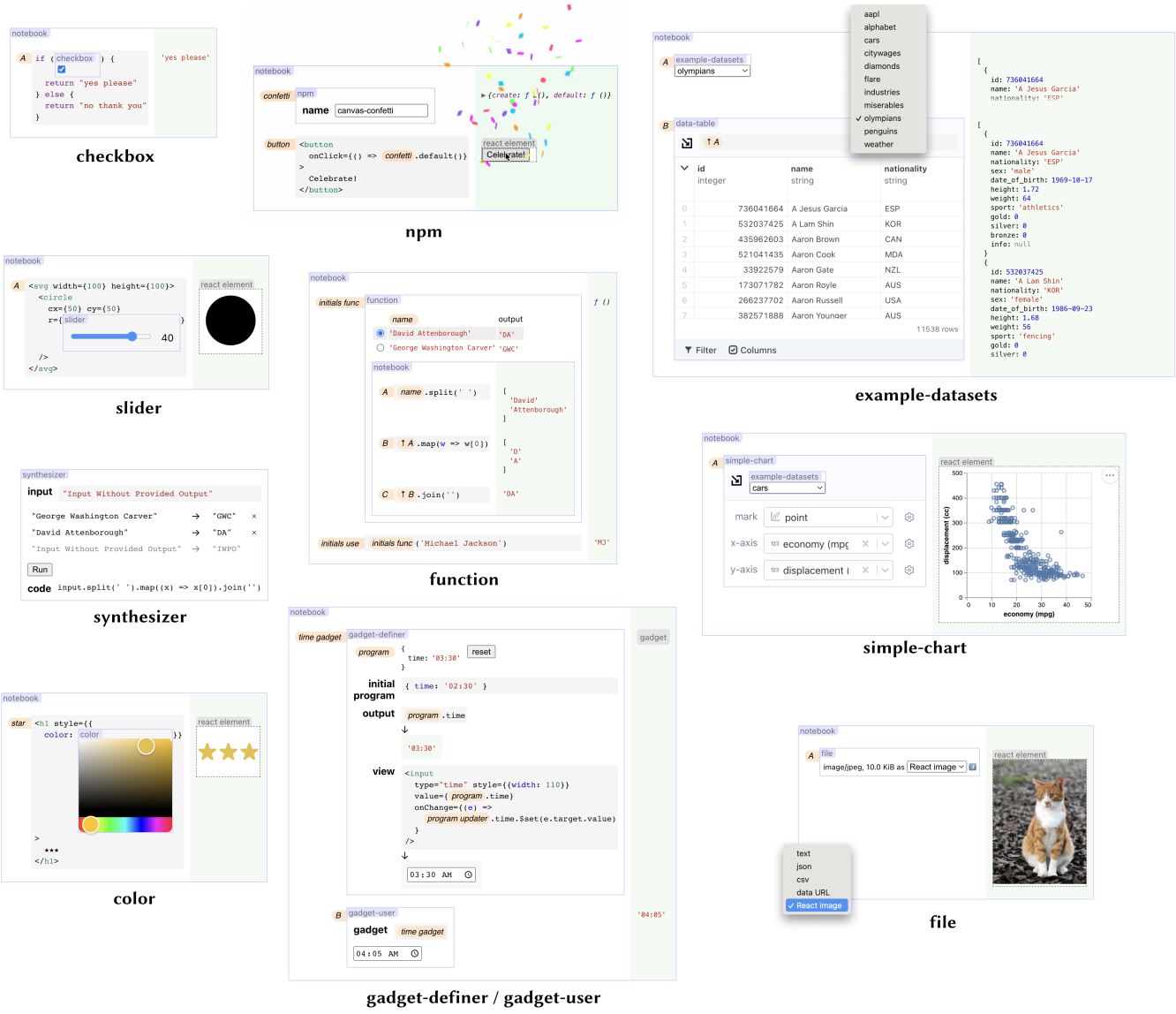


Figure 19: Example uses of ten additional components built for the Engraft platform. In rows: • checkbox: A simple checkbox control. • npm: Imports JavaScript packages from the npm registry. Here, a confetti package is used to create a celebratory button. • example-datasets: Pre-bundled copies of eleven datasets adapted from the Observable standard library [31]. Here, the "olympians" dataset is examined in a data-table. The menu of dataset options is shown inset. • slider: A simple slider control. Bounds and step size can be modified with a right-click menu (not shown). • function: A tool for defining a function in the presence of example inputs. Here, function is used to make a name-to-initials function. In a second cell of the notebook, the user uses the output of function as an ordinary JavaScript function. • simple-chart: A simple chart builder which gives the user encoding options based on columns in the data. Charts are rendered with Vega-Lite [44]. Here, simple-chart takes in the "cars" dataset provided by from example-datasets. • synthesizer: A simple example-based program synthesizer, following the example of Ferdowsifard et al. [10]. • color: A color picker wrapping `react-colorful` [48]. Here, it is embedded into HTML to style stars. • gadget-definer / gadget-user: A sketch of a meta-tool allowing new tools to be defined from within Engraft. Here, gadget-definer is used to define a new tool wrapping the HTML time-input control and gadget-user uses this tool. • file: A tool for uploading a file and embedding it persistently inside an Engraft program. Here, it is used to embed an image of a cat. A menu of alternative ways file can interpret uploaded files is shown inset.