Ten Million Users and Ten Years Later: Python Tutor's Design Guidelines for Building Scalable and Sustainable Research Software in Academia

Philip J. Guo
UC San Diego
La Jolla, CA, USA

ABSTRACT

Research software is often built as prototypes that never get widespread usage and are left unmaintained after a few papers get published. To counteract this trend, we propose a method for building research software with scale and sustainability in mind so that it can organically grow a large userbase and enable longer-term research. To illustrate this method, we present the design and implementation of Python Tutor (pythontutor.com), a code visualization tool that is, to our knowledge, one of the most widely-used pieces of research software developed within a university lab. Over the past decade, it has been used by over ten million people in over 180 countries. It has also contributed to 55 publications from 35 research groups in 13 countries. We distilled lessons from working on Python Tutor into three sets of design guidelines: 1) user experience design for scale and sustainability, 2) software architecture design for long-term sustainability, and 3) designing a sustainable software development workflow within academia. These guidelines can enable a student to create long-lasting software that reaches many users and facilitates research from many independent groups.

CCS CONCEPTS

• **Human-centered computing → Human computer interaction (HCI)**.

KEYWORDS

research software, sustainability, Python Tutor, code visualization

ACM Reference Format:
Philip J. Guo. 2021. Ten Million Users and Ten Years Later: Python Tutor's Design Guidelines for Building Scalable and Sustainable Research Software in Academia. In *The 34th Annual ACM Symposium on User Interface Software and Technology (UIST '21), October 10–14, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3472749.3474819

1 INTRODUCTION

This paper has been nearly twelve years in the making. It tells the story of Python Tutor, a research software project that we started in 2009 and have been developing for over a decade entirely within academia. So far it has been used by millions of people in over 180 countries around the world (see Figure 1) and contributed to at least 55 publications from 35 research groups across 13 countries.

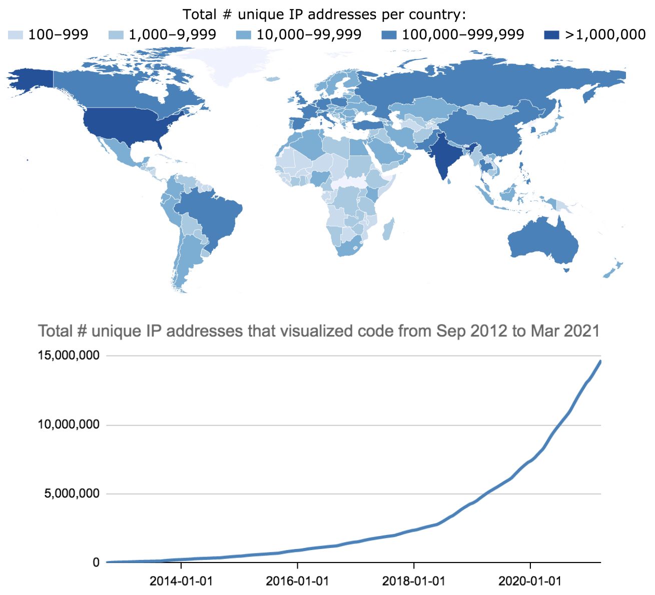


Figure 1: Number of Python Tutor users by location and date, estimated by unique IP addresses that have visualized code on the site. (There was minimal usage from 2009 to 2012.)

Why tell this story now? Because software is everywhere in modern-day research. Students, postdocs, and lab staff across many fields are now working day-to-day as software developers within academia. But instead of writing software to produce commercial products, they are writing software to discover new knowledge that they publish in peer-reviewed papers. For HCI and computing systems research, this knowledge comes in the form of new interaction techniques, system architectures, and algorithms [46, 150].

In academic research settings, the actual software artifact is usually a prototype built to validate an idea and is not meant to be maintained long-term. If researchers want to bring their ideas out of the lab and into the world (a process called *technology transfer* [19, 96]), they often start a company (some HCI examples include Tableau [134], Trifacta [63], and AnswerDash [27]) or a community-oriented nonprofit (e.g., the Scratch Foundation [2]).

But what if research software *itself* could be designed to gain widespread adoption and be sustained across many years of development entirely within academia? Is this even feasible? And is it a good use of time given that this maintenance work could instead be spent on developing new prototypes to explore new frontiers?

Design Guideline	Examples from Python Tutor project
User Experience	
Walk-up-and-use	no installation; no accounts or logins; just write code and press 'Visualize' to see results
Should 'Just Work'	robust enough to have visualized 200 million pieces of code in Python, Java, C, C++, JavaScript, & Ruby
Sharing, Not Hosting	do not store any user-generated content; share URLs and embed visualizations in other sites; this helps Python Tutor expand its reach and outsources the hosting and moderation of user-generated content
Minimize User Options	almost no user options or opaque heuristics; simplifies both user experience and software maintenance
Software Architecture	
Be Stateless	web application maintains no state; more robust, secure, and testable; scales with low-cost servers
Use Old Technologies	use older and slower-moving technologies; easier to maintain long-term; can host on low-cost servers
Minimize Dependencies	minimize dependencies on external code; bundle dependencies directly into codebase; stay independent by not depending on any external institutions (e.g., schools, companies, MOOCs)
Development Workflow	
Single Developer	single developer to maintain continuity; rarely accept outside code contributions
Start Specific	do not design as generalizable research from the beginning; start with only Python then slowly expand
Ignore Most Users	most user suggestions lead to feature creep; take precautions to protect against noise and harassment

Table 1: Our ten design guidelines for building scalable and sustainable research software within academia (Sections 6–9)

We believe this is feasible and has notable benefits: Such software directly benefits users and would likely not exist if it were not made in academia, because many kinds of useful software are not profitable for companies to fund. It also benefits researchers since it opens up opportunities for more impactful longer-term research. Since researchers can keep full control of their software, their incentives are aligned with doing follow-up research rather than making marketable products or fostering a nonprofit community.

In this paper we use our decade-plus of experience working on Python Tutor to demonstrate one way to build scalable and sustainable software in academia and to show some benefits of doing so. Python Tutor (despite its outdated name) is a web-based educational tool that visualizes the run-time state of code in Python, Java, C, C++, JavaScript, and Ruby in order to help people learn programming. Aside from widespread usage (see Figure 1), it has also spawned a long lineage of research: Dozens of labs worldwide have used its open-source codebase to build new software systems, used its large userbase to run empirical studies, and evaluated its efficacy as a pedagogical tool in both classroom and online settings.

We first present the design and implementation of Python Tutor. Then we reflect on our software development experiences to distill a set of *design guidelines for building scalable and sustainable research software in academia*. We focus on academia due to the especially challenging constraints of this environment: Unlike in industry, within academic labs there is often no full-time software development staff, no stable long-term funding, and no marketing or PR resources to publicize one's projects to broader audiences.

Table 1 summarizes our three sets of design guidelines: 1) user experience design for scale and sustainability, 2) software architecture design for long-running research software, and 3) designing a sustainable software development workflow within academia. Many of these guidelines go against the best practices for industry and open-source software development [24, 37, 68, 89, 117] due to the unique resource constraints of academia (e.g., use old technologies, minimize dependencies, single developer, ignore most users).

Our story shows one extreme point along the spectrum of methods for building research software – one that prioritizes scale and long-term sustainability over rapidly producing new ideas. The main limitation of our approach is that it is a very inefficient way to do research – it took us almost twelve years to write a single paper! But even for the majority of researchers who choose faster prototyping methods, Table 1 is still a useful way to think about design tradeoffs throughout the research programming process.

More broadly, we want to spark discussions about how much time should be devoted to maintaining software long-term in an academic setting. Through both our own experiences building research software and advising many students on doing so, a common question that comes up is: *"How much time should I spend on refining my software and getting users versus moving onto new projects?"* We empathize with those students who want to see their thousands of hours of programming work have more direct impact on users. But we also recognize that the way most HCI and computing systems research makes impact is via novel ideas validated by prototypes, not via the software itself; over time, some of these ideas diffuse into widely-used products as industry picks up on the most marketable ones (see tire-tracks diagrams [66, 125] for a visual history). In the end, the right time balance depends on each individual's goals.

In sum, this paper's contributions are:

- The design and implementation of Python Tutor, an educational program visualization tool that is, to our knowledge, one of the most widely-used pieces of research software.
- Ten design guidelines to help researchers in academia build software that can potentially grow a large userbase and be sustained across many years of development.

2 RELATED WORK

Program visualization systems: Python Tutor [54] continues the long lineage of research in program visualization systems for education. Sorva et al. analyzed the design of 46 of these systems from the 1980s to the 2010s [130]. Although the idea of visualizing runtime state of programs is now decades-old, Python Tutor's design is novel in three ways: 1) it is the only one to work across multiple popular languages (Python, Java, C, C++, JavaScript, Ruby) and to formalize a language-independent schema for execution traces (Figure 3), 2) it is the only one that can visualize memory-unsafe C/C++ code that commonly occurs in-the-wild, 3) it is designed for widespread deployment and long-term sustainability, which has made it into the most widely-used program visualization system [129, 130].

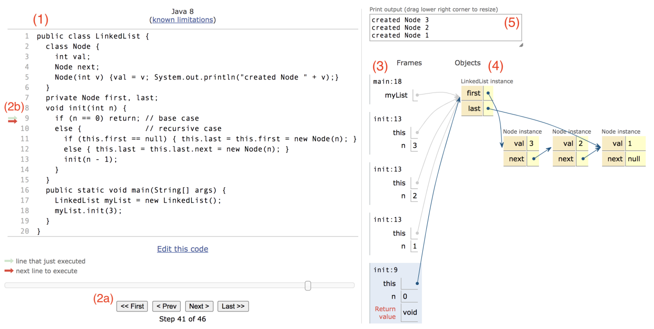
Building scalable and sustainable software in academia: This paper uses Python Tutor's decade-long development process as a case study to distill a set of design guidelines summarized in Table 1. To our knowledge, we are the first to introduce design guidelines for building scalable and sustainable software within the setting of academia. In contrast, existing design principles for software engineering all target either industry settings where there are teams of full-time developers [24, 68, 89] or geographically-distributed open-source software development efforts [37, 117].

Aside from Python Tutor, we know of only a few active research systems in HCI-related fields that have both: 1) independently grown a large userbase of at least tens of thousands of users without industry/product collaborations (i.e., *scalable*) and 2) lasted for a decade or more while being developed inside academia without being turned into commercial products (i.e., *sustainable* within academia). These include Vega/Vega-Lite [123], LabintheWild [118], MovieLens/GroupLens [60], Scratch [87], Racket [44], BlueJ [75], OpenDSA [47], Runestone [42], and ASSISTments [62]. In addition, software such as D3 [23], Jupyter [74], and LLVM [82] started in academia but later grew into open-source projects with a community of developers, many of whom are now employed by companies.

Several of these researchers wrote about their field deployment experiences [60, 62, 118], but those focus on the specifics of each project. Our paper contributes to this discussion by generalizing our decade-long experiences into high-level design guidelines that can apply across a broad range of research software.

Challenges of building software in academia: Career incentives can make it hard to sustain long-term software projects within academia since the main expected output is peer-reviewed publications. Thus, software is often made just 'good enough' to run experiments and get the needed results for publications [133]. The reproducible research movement [108, 139] has encouraged academics to make their code public and reusable so that others can verify one's experiments, but there is little career incentive for doing so [36]. Relatedly, it is hard to hire full-time software developers in academia since they are able to get higher-paying and more stable jobs in industry rather than relying on grant funding [102].

Within the HCI community, prominent researchers have highlighted the challenges of doing systems research. Landay's widely-read 2009 blog post [81] pointed out that evaluation criteria for HCI paper submissions can put systems work at a disadvantage due to the high amount of implementation effort required to build and validate end-to-end systems. In 2017 Marquardt et al. organized



```
1  public class LinkedList {
2    class Node {
3      int val;
4      Node next;
5      Node(int v) {val = v; System.out.println("created Node " + v);}
6    }
7    private Node first, last;
8    void init(int n) {
9      if (n == 0) return; // base case
10     else {              // recursive case
11       if (this.first == null) { this.last = this.first = new Node(n); }
12       else { this.last = this.last.next = new Node(n); }
13       init(n - 1);
14     }
15   }
16   public static void main(String[] args) {
17     LinkedList myList = new LinkedList();
18     myList.init(3);
19   }
20 }
```

```
created Node 3
created Node 2
created Node 1
```

```
main:18
  myList
init:13
  this
  n 3
init:13
  this
  n 2
init:13
  this
  n 1
init:9
  this
  n 0
  Return value void
```

first	
last	

val	3
next	

val	2
next	

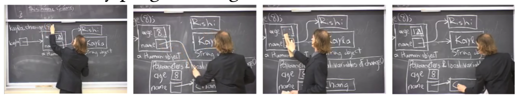
val	1
next	null

Figure 2: Python Tutor lets users (1) write code in six languages (Python, Java, C, C++, JavaScript, Ruby), (2) step through its execution forwards and backwards, see the runtime memory contents of (3) global variables, stack frames, and (4) heap objects, and (5) see what the program prints.

a CHI workshop on HCI toolkits [88], where Fogarty articulated ways that code can be a significant contribution [46] while Rädle and Klokmose described the challenges of maintaining toolkit code longer-term after initial papers get published [116]. Myers preserves his group's interactive systems research long-term by making detailed video recordings [95]. Our paper contributes to this discussion by presenting design guidelines for building scalable and sustainable HCI software within these constraints of academia.

3 OVERVIEW OF PYTHON TUTOR

Before describing its technical details, we first provide an overview of Python Tutor. This paragraph at the top of its user FAQ page [115] summarizes what it is designed to do:

> Python Tutor is designed to imitate what an instructor in an introductory programming class draws on the blackboard:
>
> 
>
> It's meant to illustrate small pieces of self-contained code that runs for not too many steps. [...] If your code can't fit on a blackboard or a single presentation slide, it's probably too long to visualize effectively in Python Tutor.

Instructors use it as a teaching tool, and students use it to visually understand code examples and interactively debug their programming assignments. Figure 2 shows how a typical user (either an instructor or a student) would interact with it:

(1) Go to pythontutor.com and select a language. Here the user chose Java and wrote code to recursively create a `LinkedList`.
(2) Press 'Visualize' to run the code. This code ran for 46 steps, where each step is one executed line of code. Go to any step (2a) and see what line of code was being run at that step (2b).
(3) See the frames of all functions/methods on the stack at this step, each of which shows its local variables. Here at step 41 we see `main()` along with 4 recursive calls to `init()`.
(4) See all objects on the heap at the current step. Here it shows a `LinkedList` instance with `first` and `last` fields pointing to its first and last `Node` instances, respectively. Each `Node` has a numerical value and a `next` pointer.

```
execution_trace: list<step>

step:
  current_line: integer
  global_variables: list<name_to_value>
  stack: list<frame>
  heap: list<object>
  print_output: string
  exception_message: string

frame:
  function_name: string
  local_variables: list<name_to_value>
  id: integer              [for closures]
  parent_frame_id: integer [for closures]
  has_exited: boolean      [for closures]

name_to_value:
  name:  string | object [usually string]
  value: object

object:
  memory_address: integer
  type_name: string
  data: primitive | collection
```

```
primitive:  integer | float | string | boolean | null

collection: sequence | set | class | instance | function

sequence:
  contents: list<object>
  dimensions: list<integer> [for multidimensional arrays]

set: list<object>

class:
  name: string
  superclass_names: list<string>
  class_fields: list<name_to_value>

instance:                     [object created from a class]
  class_name: string
  instance_fields: list<name_to_value>

function:
  name: string
  parent_frame_id: integer  [for closures]
  default_arguments: list<name_to_value>
```

Figure 3: The backend runs the user's code and produces an execution trace in a language-independent format specified by this schema. (Notation: `list<X>` means a list of elements of type `X`, alternative types are separated by the `|` delimiter.)

(5) See what has been printed up to this step. Here the print statement in the `Node` constructor (line 5) has run 3 times.

The user can navigate forwards and backwards through all execution steps, and the visualization changes to match the run-time state of the stack and heap at each step. In this example, the user would see their custom `LinkedList` data structure getting incrementally built up one `Node` at a time via recursive calls to `init()` until the base case is reached when `n==0`.

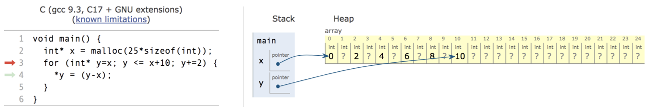
4 DESIGN AND IMPLEMENTATION

Python Tutor is a standard web application. When the user writes code in their web browser and presses 'Visualize,' that code gets sent to the server backend, which runs it using a language-specific execution engine. The engine produces an execution trace formatted in the schema of Figure 3. That trace is serialized as JSON and sent back to the user's browser, where the frontend renders it.

4.1 Backend: Multilingual Run-Time Tracing

Language-independent execution trace: Although this project started out being only for Python [54], as we expanded to other languages we realized that run-time state diagrams for many languages actually look very similar. For instance, although the diagram in Figure 2 was from Java code, it could have just as well been produced by Python, C++, Ruby, or JavaScript code, since all of those languages have the concept of function calls, stack frames, objects, and pointers. Thus, we created a language-independent execution trace format that captures the run-time state of code written in a variety of imperative and object-oriented languages.

Figure 3 shows the schema of our execution trace format. Reading from the top left, an `execution_trace` is a list of steps (e.g., the trace in Figure 2 has 46 steps). Each `step` contains the full run-time state at that step, such as the contents of global variables, the stack, the heap, and what the program has printed out so far. The stack is a list of frames, where each `frame` contains its local variables and IDs for handling *closures* (i.e., nested function scopes) [144]. Each variable is a `name_to_value` mapping: names are usually strings but can be arbitrary objects (e.g., Python dictionary key 'names' can actually be any hashable object [114]). An `object` has a memory


C (gcc 9.3, C17 + GNU extensions)
(known limitations)

```c
void main() {
  int* x = malloc(25*sizeof(int));
  for (int* y=x; y <= x+10; y+=2) {
    *y = (y-x);
  }
}
```

Figure 4: Our C/C++ engine uses Valgrind [98] to track that `x` points to a heap array of size 25 and that only even-numbered elements up to 10 are initialized using `y` to point into the middle of it (the rest have '?' for unknown values).

address, a type name (e.g., 'unsigned long' in C or 'symbol' in JavaScript), and its data (either a `primitive` or a `collection`).

The right half of Figure 3 shows the kinds of data that objects in the execution trace can hold. This includes the usual primitive data types (e.g., integers, floating-point numbers, strings) as well as a variety of collections that are built into many programming languages. For instance, a `sequence` is an ordered collection of objects (e.g., C/Java array or Python list/tuple), a `set` is unordered, and `class` and `instance` are object-oriented programming constructs that hold collections of name-to-value mappings (i.e., fields). Python dicts and Ruby hashes can be represented as instances.

Finally, an essential kind of object is a *pointer*, which is identified by `type_name='pointer'` and whose `data` is an integer representing a memory address of another object. In many languages, what is stored in the stack and in collections are actually *pointers to objects allocated on the heap*, so this representation captures those details.

Language-specific execution engines: An execution engine runs the user's code on the server and produces an execution trace with the schema of Figure 3. We have created five execution engines so far: Python, Java, Ruby, JavaScript, and C/C++. Adding support for more languages is a matter of writing new engines for them.

Most engines are implemented atop the standard debugger interface for each respective language. For instance, the Python engine uses its debugger module called `bdb` [113], Java uses the JDI module (Java Debug Interface) [104], Ruby uses `debug_inspector` [86], and JavaScript uses the Node.js server-side execution engine and debugger protocol [100]. These interfaces all allow an engine to programmatically step through each line of code execution one at a time, inspect the run-time state of all functions/methods on the stack and all in-scope memory objects, and serialize their values to a JSON execution trace in the format of Figure 3.

Our most complex engine is for C/C++ since those languages are not memory-safe [17]. That means given a pointer to a block of memory, it is impossible to tell: a) whether that memory has been initialized or just holds meaningless junk values, b) how many elements of data are located there – i.e., is this a single element or an array, and what size is the array? Thus, a debugger like GDB cannot produce accurate traces of realistic code. Instead, we built our C/C++ engine as a plug-in for Valgrind Memcheck [98], a run-time tool that augments every byte of memory with information about whether it has been allocated and/or initialized. That way, our engine can safely traverse memory to get an accurate trace of the size and contents of objects in both the stack and heap (using a technique similar to our master's thesis [52]). Figure 4 shows Python Tutor accurately visualizing the size and contents of a partially-filled C array, which is not possible to do automatically with GDB.

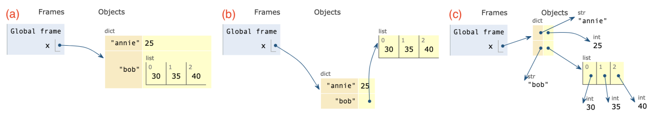


Figure 5: Users control whether to: (a) nest all objects inside of each other (list is nested inside of the dict), (b) do not nest objects [default], (c) do not even nest primitive values.

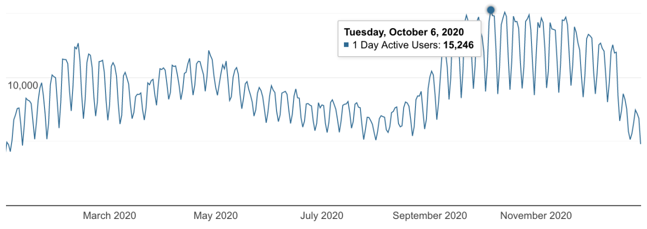
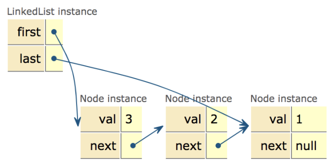


Figure 6: Daily active users in 2020, from Google Analytics

Execution sandbox: To protect the server against resource overuse and security breaches, all execution engines run in a sandbox. We use a combination of Docker [20] and Linux `setrlimit` [111] system calls to create a custom sandbox that limits execution time (< 10 seconds) and memory usage (< 200 MB), forbids file accesses beyond module files that are in the sandbox, and forbids network access to prevent user code from launching network-based attacks.

4.2 Frontend: Execution Trace Visualizer

Data rendering: The frontend translates each element from Figure 3's schema into a visual representation. Thus, it knows how to render built-in data types such as numbers, strings, sequences (e.g., C/C++/Java arrays, Python lists and tuples), classes, instances (e.g., `LinkedList` and `Node` from Figure 2), and pointers to any object.



The frontend renders custom user-defined data structures out of these built-in types. For example, the image on the right is copied from Figure 2. It shows a `LinkedList` instance and three `Node` instances, each with two name-to-value mappings (their instance fields). This visualization looks like a familiar linked list, but Python Tutor has *no knowledge* about what linked lists are (or any custom data structure that is not in Figure 3's schema); here it simply sees instances, fields, and pointers, and renders them using boxes, arrows, and text labels.

Layout algorithm: How does the frontend know how to render the data in the above example so that it looks like a linked list? Instead of using complex algorithms or heuristics, we implemented a simple grid-based layout that works well in practice. Objects are laid out vertically in the order they are created (from top to bottom), and *structurally-identical* objects are laid out horizontally from left to right. In the above example, the `LinkedList` instance appears first, then the first `Node` appears below it. As more `Node` instances get created, the frontend recognizes that they are structurally-identical (same field names), so it lays them out horizontally (this is a common idiom for linked data). Finally, each row gets nudged to the right so that as many pointers as possible point rightward, which helps prevent them from overlapping with objects. That is why the three `Node` instances all appear to the right of the `LinkedList`.

Layout customizations: The user can drag-and-drop any object to move it around the visualization. This makes up for cases where our simple grid-based layout looks messy or occludes certain objects. They can also hide selected variables and object fields, which can remove on-screen clutter when visualizing more complex code.

Object nesting: We provide three nesting options, illustrated in Figure 5: (a) nest all objects, which renders the list inside of the dict in this example, (b) do not nest objects, which renders the list outside the dict (this is the default since it looks the most sensible to us), and (c) do not even nest primitive values such as strings and numbers, which renders them all as objects with pointers to them. This is pedantically the most 'correct' for Python and Ruby since primitives are objects, but it often produces too much clutter.

5 IMPACT OVER THE PAST DECADE

Since this system was released over a decade ago, instead of doing a traditional HCI evaluation [103] we will summarize its impact.

Large-scale worldwide usage: As Figure 1 shows, the most salient impact of Python Tutor has been its scale of worldwide usage: We estimate that over ten million people in over 180 countries have used it so far, based on unique IP addresses[1] that have executed code. Figure 6 shows daily usage patterns in a typical year (2020). There is a weekly rhythm, peaking on weekdays to around 15,000 users per day and dropping on weekends to around 8,000. Summers have the lowest activity since the fewest classes are in session then.

How do users find Python Tutor? A significant percentage of traffic (36% in 2020) comes from Google searches for relevant terms like '`visualize code`', '`python tutor`', '`java tutor`', etc. In addition, the HTTP referer fields [91] in our server logs show that, as of March 2021, at least 28,671 webpages from 8,087 different domain names contain direct URL links to pythontutor.com (filtering out outlier domains with fewer than 10 referral visits). Of those, there are 504 unique `.edu` domain names, which approximates the number of educational institutions that are using it. These inbound links also help our website organically rank well on search engines.

Impact on research: Although widespread usage is personally fulfilling, what is more significant for an academic audience is the impact that Python Tutor has had on generating new research publications. The former facilitates the latter, though, since our work in maintaining this software to last over ten years and serve millions of users has enabled it to be used in research settings far beyond what is feasible with a prototype.

To assess research impact, we performed a Google Scholar search for publications that mentioned '`pythontutor.com`' or cited our 2013 paper [54]. We found around 500 publications with this search and read through them. Out of those, we determined that at least 55 publications *directly used Python Tutor as a part of their research.* These publications came from at least 35 different research groups, which we inferred based on the name and affiliation of each paper's senior author. Although a few of these are colleagues whom we

[1]Note on estimates: Home IP addresses, though technically dynamic, usually remain fixed over several months or more [21]. A single user may use multiple IPs if they change locations. But many users often share a single public IP within school networks.

	# papers	# groups	countries
Building new software systems	24	19	9
Running empirical studies	11	6	1
Evaluating Python Tutor	20	15	8
Total	55	35†	13†

Table 2: Number of papers from Google Scholar that have directly used Python Tutor in three main ways, and number of research groups and countries each came from. (†several groups and countries overlapped between categories.)

shared data with, we did not even know about the majority of these researchers and papers until we performed a Google Scholar search for this evaluation. Table 2 summarizes our bibliometric analysis and shows three main ways that these projects used Python Tutor:

1) Building new software systems: We found 24 papers from 19 research groups that used Python Tutor's open-source codebase to build new research software that either extended its features or integrated it into other platforms: TraceDiff [135], CodeSkulptor [136], VizQuiz [127], CAT-SOOP Detective [61], PILeT [9], UNCode [120], JavelinaCode [152, 153], JaguarCode [151], Ladebug [85], OverCode [50], CodeInk [124], Omnicode [69], Data Theater [83], Trace Table Tutor [122], PCRS [155], OPT+Graph [31], Python Tutor Graph Builder [128], CS Circles [112], IMI Python [32], Runestone CodeLens [42, 93], concept-driven explanations [13], PITON [38]. We were involved in only four of these [50, 69, 83, 124], which indicates that our code is robust enough to enable many other research groups to build new software upon it without our help.

2) Running empirical studies: 11 papers from 6 research groups were empirical studies conducted using Python Tutor's large userbase or its data set of code submissions. These projects include deploying surveys to the website to reach learners around the world [56, 57], measuring learner behavior on the site [33, 137], using a pool of concurrent site visitors to tutor one another [55, 59], crowdsourcing learners to annotate code snippets [51, 58], design-based research with teachers who use it in digital textbooks [43], and analyzing a large corpus of code submissions to develop program analysis tools for fault localization [28] and program repair [39]. These publications (mostly from our own collaborators) show that Python Tutor's widespread usage can provide data to enable novel research that is not as feasible in a smaller-scale lab setting.

3) Evaluating Python Tutor's efficacy: Lastly, even though this paper does not present a user evaluation of Python Tutor, we found 20 papers from 15 groups that independently evaluated Python Tutor across several contexts. We were not involved in any of these evaluations. These papers include evaluating the efficacy of Python Tutor when used in programming courses [11, 70–72, 119], evaluations of Python Tutor embedded within other interactive systems (JavelinaCode [35] and UNCode [121]), how readers interacted with Python Tutor visualizations within digital textbooks [10, 40, 41, 107], comparative evaluations of Python Tutor with other code visualization and tutorial systems [7, 15, 45, 67, 73, 94, 110, 130], and using it as a case study to evaluate a code annotation toolkit [126]. These papers provide evidence that Python Tutor is a compelling target for other researchers to run independent evaluations of it.

6 DESIGN GUIDELINES FOR SCALABLE AND SUSTAINABLE RESEARCH SOFTWARE

In the next three sections we present ten design guidelines that we distilled throughout our past decade of work on Python Tutor. These are intended to help researchers build software that can grow a large userbase and be sustained across many years of development within the resource constraints of a university environment.

Since these guidelines are based on our personal experiences, the usual caveats apply: they may not generalize to different domains, and there are examples of widely-used research software that do not meet some of them. When possible, we will present alternative approaches that can help generalize these guidelines further.

7 USER EXPERIENCE DESIGN

These guidelines show how to build user experiences that attract large numbers of users while under the resource constraints of an academic setting with no full-time software development team.

7.1 Walk-up-and-use

One way to get software to grow a large userbase is to make it a walk-up-and-use experience [138] where people can (virtually) 'walk up' and start using it right away.

No installation or configuration: For software to be walk-up-and-use, first-time users must not need to install or configure anything. At the time of writing (2021) this means creating a web application using APIs that are built into modern browsers (i.e., no plug-ins) rather than a desktop or mobile app. For Python Tutor, users simply visit http://pythontutor.com/, type in their code, and press 'Visualize' to run and visualize it. While a web-based design seems obvious today, back in 2009 when we started developing Python Tutor there were no easily-accessible web-based interfaces for learning to code. The few research prototypes that existed back then were desktop apps. Some used Java to run in the browser, but those were dependent on getting Java Web Start [105] and related technologies installed as browser plug-ins. Thus, even though early versions of Python Tutor had limited functionality, in the early 2010s it was one of the only free ways to run code in a browser in a 'walk-up-and-use' manner, so that attracted many early users.

No user accounts or logins: Some research software requires users to create accounts so that the researchers can track user-specific information for their studies. This requirement severely limits the number of users in-the-wild who would be willing to try out an unknown prototype. It is widely-known that if a user's first impression of a website or mobile app is seeing an account creation or login screen (i.e., a 'login wall'), the vast majority will leave [26]. Implementing accounts via third-party authentication (e.g., Google, Facebook, Twitter) may help, but users may still be wary of giving an unknown app access to their personal accounts. In our experience, not requiring login is the best way to scale to the most users. Thus, Python Tutor has no user accounts or logins so that users can start visualizing code right away.

How does Python Tutor collect user-specific research data without accounts? When a new user visits the site, it generates a new UUID [148] (a random 128-bit number) and stores it in their web browser's persistent localStorage data [92]. When that user returns to the site later, it will re-use the stored UUID from their localStorage. Similarly, for every new browsing session, Python Tutor generates a new UUID and stores it in the browser's sessionStorage. Unlike localStorage, sessionStorage gets erased when the current browser tab or window closes [92]. All user interactions are tagged with user and session UUIDs so we can split server activity logs by users and sessions, respectively. This lightweight technique trades off some precision for user convenience: It requires no user accounts, but it can be imprecise if a user switches browsers. In practice, though, many users stick with the same browser for extended periods of time, or if they switch computers their browser state is synced with features such as Google Chrome sync.

As an alternative, if research software requires more detailed data (e.g., user demographics, survey responses), we suggest prompting users *while* they are using it rather than requiring that data upfront. This way, users can start getting some immediate value right away without first filling out a lengthy personal information form. Another way to get close to walk-up-and-use is to build one's research software as a lightweight extension for popular web apps such as Facebook or Twitter, as many social computing systems do [76]. This technique bootstraps off the built-in audience of these sites and can spread via their social sharing mechanisms.

7.2 Should 'Just Work'

Making software 'just work' as users expect is necessary for growing a large userbase, even though such implementation details are uninteresting from a research perspective (i.e., they do not lead to new publications). But if people are disappointed by their first experience, then it is unlikely that they will return or recommend it to others. And if they like their initial experiences, then they will organically publicize it via word-of-mouth referrals.

Craftsperson's mindset vs. researcher's mindset: HCI research prototypes are often user-tested under well-controlled conditions with the researchers present to either limit the scope of user inputs or to explain away unsupported features. In contrast, Python Tutor is being used without our supervision to visualize over 100,000 pieces of new code every day, so we must ensure it 'just works' for arbitrary unknown code that users happen to enter into it. This means putting in the sustained implementation effort to handle complex and ever-evolving language features, error-inducing inputs, and arcane edge cases that arise at scale. There is no magic here – just a lot of hands-on software crafting work to make our code increasingly more robust, to track bugs, and to build up a detailed regression test suite over time.

In this way we have purposefully adopted a craftsperson's mindset rather than a researcher's mindset: Whereas a research mindset optimizes for building MVPs (Minimum Viable Products) to rapidly validate and publish novel ideas, a craft ethic requires spending long stretches of time to get all of the mundane details right [18, 30]. Staying disciplined about limiting the scope of our software (Section 9) helps to make this effort sustainable across many years.

Use production-grade tools: Another way to help make software 'just work' is to build it upon production-grade tools. While this is standard practice in industry, within academia many researchers instead choose to build their prototypes upon more ad-hoc tools (e.g., those created by fellow researchers) since those can allow them to be more expressive and innovative. As an example in Python Tutor's domain, code visualization and tutoring systems are often built upon "home-grown" frameworks such as hand-written language interpreters or AST transformers [12, 97, 110, 131]. This bespoke approach lets researchers add fine-grained customizations to explore novel ideas such as stepping into expressions within a line of code to visualize and explain their detailed semantics. But the tradeoff is that it makes those systems less likely to 'just work' when users in-the-wild paste in arbitrarily complex code, since researchers do not have the time to implement all the details required to match the many edge cases of real programming languages.

In contrast, building upon widely-used production-grade tools helps make Python Tutor 'just work' on a large variety of real code, at the expense of being less expressive. Specifically, it extends official versions of programming language interpreters, compilers, and debuggers (Section 4.1). We initially experimented with some prototype third-party implementations of Python interpreters [1, 25] to add finer-grained expression-level interactive stepping that related tools have [12, 97, 110, 131] (instead of ordinary line-level stepping that debuggers provide). But the problem we encountered was that those third-party implementations could never match all the details of the official Python interpreter and, more frustratingly for novices, they produced different error messages that often do not match what is written in instructional materials and tutorials.

7.3 Sharing, Not Hosting

Research software that explores online interactions [76] often host their own user-generated content such as user profiles, images, discussion posts, and chat logs. While this makes sense for a time-limited field deployment to collect high-fidelity study data, we have found that it is impractical at the scale of usage that Python Tutor has reached. Thus, we believe academic projects should *not* host any user-generated content if they want to sustain a large userbase.

The main reason why hosting content is impractical at scale is due to the need for moderation to remove objectionable content [49, 109]. Gillespie eloquently conveys this challenge in his book *Custodians of the Internet* [49] (page 9): *"Content moderation is hard. This should be obvious, but it is easily forgotten. Moderation is hard because it is resource intensive and relentless; because it requires making difficult and often untenable distinctions; because it is wholly unclear what the standards should be; and because one failure can incur enough public outrage to overshadow a million quiet successes."*

With millions of users, even on the most wholesome of websites like those for learning to code, it is inevitable that malicious users will post spammy, obscene, abusive, harassing, and even illegal content. For instance, learn-to-code websites often allow users to upload images or animations to put into their coding projects, which then get displayed in public galleries; and they also host on-site discussion forums for learners to help one another. Moderating such user-generated content requires massive amounts of human labor. At one extreme, companies like Facebook and Twitter hire tens of thousands of contractors [34, 99] to do this work. Even though academic projects are not nearly at this scale, if they grow a significant userbase then they will require members of the research team (or outside volunteers) to moderate. For Python Tutor, we did not feel comfortable hiring students for this sort of labor and also did not have the energy to do it on our own.

Thus, we decided not to host any user-generated content on the Python Tutor site, such as user-written code, discussion posts, Q&A, or tutorials. This decision also simplified compliance with policies ranging from university IRBs all the way to international data privacy laws, which made it logistically easier to sustain this project across many years.

We understand that some academic projects need to rely on user-generated content to address their research questions. So how can they get the benefits of user-generated content without the burdens of hosting and moderating it? The approach that has worked well for Python Tutor is to *outsource content hosting to other websites* by letting its visualizations be easily shared and embedded across the web. Thus, our content philosophy is *sharing, not hosting*.

URL sharing and embedding: Users can share the current state of any Python Tutor visualization by generating a URL. This URL encodes their current code, options, and the execution step they are now viewing. Over the past decade users have posted these URLs all over the web when asking and answering programming questions, such as on MOOC discussion forums, Stack Overflow, GitHub Issues threads, and Slack and Discord chat communities. In this way, we outsource the work of content moderation to those sites, which each have their own policies and moderators.

Similarly, instructors can generate an iframe URL to embed Python Tutor visualizations into their own websites, which lets them integrate visualizations into online textbooks, course lessons, and web-based lecture slides. This feature lets us outsource the work of instructional content creation to domain experts rather than doing it ourselves.

Free organic advertising across the web: Our *sharing, not hosting* approach has had another serendipitous benefit: Without any intervention from us, Python Tutor URLs have been posted across over 28,671 webpages from 8,087 unique domains over the past decade (see Section 5). In addition, 16,050 webpages from 1,764 domains have embedded Python Tutor visualizations within them as iframe embeds. All of those webpages are *advertising Python Tutor for free* since their users will see Python Tutor URLs posted there, click on them, and then discover our website.

Imagine if we had decided to host our own user-generated content like many other learn-to-code websites do. Then not only would we be burdened with doing content moderation, but Python Tutor URLs would not have organically spread as far and wide across the web. We credit this simple design decision as the main reason why Python Tutor usage has grown steadily over the past decade without us putting any time or money into advertising. This is critical since we as academics do not have the marketing or PR resources that companies do.

That said, the main downside of our approach is that user-generated content is dispersed across the web on a variety of third-party sites that we do not control. Thus, researchers who take our approach should be prepared to scrape third-party sites and deal with lower-fidelity research data. As an alternative approach, one elegant compromise is to outsource content hosting but to direct users to use a specific website with a dedicated tag. For instance, the creators of D3 [23] ask users to post all questions and discussions on Stack Overflow with a `'d3.js'` tag rather than maintaining their own in-house discussion forum [132]. As another alternative, if content hosting is absolutely necessary, we recommend using an externally-managed cloud service such as an enterprise Stack Overflow [4] or Slack instance [3] to get some protections via those paid platforms. We also recommend *not* having this content be publicly visible or indexed by search engines, since private online communities are less likely to require as much content moderation.

7.4 Minimize User Options

As software gets more widely-used over time, it is inevitable that users will ask for custom options to meet their needs. Thus, it is unsurprising that decades-old software (e.g., Photoshop, MS Word) accumulate thousands of options, often hidden within deeply-nested menus [79, 80]. This level of complexity is unsustainable for academic projects that do not have full-time software development staff. Thus, we suggest minimizing the number of user options.

Throughout the past decade, we have received many requests for options to customize how Python Tutor visualizations look, since there is no single "right" way to display the run-time state of code. We have rejected nearly all of these requests since every added option both harms user experience and increases our software maintenance burden. Options harm user experience in two ways: 1) they overwhelm users, especially novices, with choices to make [5, 78], and 2) they make Python Tutor visualizations harder to comprehend at a glance, since viewers also need to check and understand which options were set when rendering a given visualization. Also, options burden developers by making their code more complex and harder to test; each binary toggle option could double the number of code paths to test.

Ideally, Python Tutor would have no user-specified options so that there would be only one canonical way for it to visualize a given piece of code. We tried to stick to this ideal, but we added three options for early power users (object nesting policy, show/hide exited frames, pointers as arrows vs. text). The weight of those early decisions still burden us today since the code to implement them percolated throughout our codebase. We do not feel comfortable removing those options since that would break backward compatibility. Many people have posted URLs of old Python Tutor visualizations with those options, so we do not want to break them.

One alternative to user options is to use heuristics to activate certain settings. For Python Tutor, we considered heuristics for automatically hiding "uninteresting" elements such as boilerplate objects from modules imported by user code. We also tried heuristics for rendering certain kinds of more advanced data structures with better aesthetic layouts (e.g., binary trees, force-directed graphs). However, those heuristics were hard to design, had too many special cases, and hindered usability since they were completely opaque to users. Instead we opted for a more transparent solution: render all objects using a simple grid-based layout (see Section 4.2). Then we let users customize visualizations by manually dragging objects around the canvas or hiding elements to make their own custom layouts. In sum, along with minimizing user options, we also suggest not having opaque heuristics that feel like "magic" to users, since those can lead to confusing experiences for novices.

8 SOFTWARE ARCHITECTURE DESIGN

We discussed outward-facing user experience issues in the last section, so now we turn inward to consider the technical architecture of software systems. These three guidelines can help research software developed in academia be more scalable and sustainable.

8.1 Be Stateless

Web-based research software often maintains server state, most commonly embodied by CRUD (Create-Read-Update-Delete) web app architectures [146]. In our experience it is hard to scale these web apps to large numbers of users when maintaining them within an academic lab without professional I.T. staff. Thus, we encourage researchers to aim for a stateless architecture by adopting our design guidelines in Section 7 such as no user accounts, storing state in users' browsers if needed, and our *Sharing, Not Hosting* approach of outsourcing content hosting to other websites.

Python Tutor embodies this stateless philosophy: its server does not maintain any persistent state. Users type code into their browser, that code gets sent to the server to run, and then the server returns a visualization to the browser. The server does not store any information about users or sessions (although each user's browser stores UUIDs for logging). Users can also visit specially-generated URLs that contain prewritten code and see the corresponding visualizations (Section 7.3). Since all the state is encapsulated in the URL string, again the server does not need to store any state.

This stateless architecture has made it possible for us to maintain nearly 100% uptime of the Python Tutor web application over many years without hiring I.T. staff (or learning much about I.T. ourselves). Some benefits include:

- Simplicity: no need to maintain database software or to pay for more storage space as usage grows over time.
- Low-cost: easier to find low-cost server hosting, which reduces the need to apply for external funding
- Security: can be more secure since many storage-based attacks (e.g., SQL injection) are not possible.
- Privacy: does not store user accounts or personal data, no worries about data privacy laws in different countries.
- Scale: easy to scale horizontally by replicating the backend code across multiple servers (we currently have four).
- Reliability: no bugs related to server being in an inconsistent state, can easily re-image server to roll back code changes.
- Testing: easy to test since there is no state, test cases simply map user code and options to expected visualizations.
- Bug reporting: users can generate a URL to send reproducible bug reports since all relevant state is in the URL.

If it is absolutely critical for a web app to maintain its own state, we recommend isolating its stateful part into a well-encapsulated component with a clean API [140]. That way, the app's processes remain *"stateless and share-nothing"* [141], which conform to this guideline from well-known industry best practices for scalable web apps: *"Execute the app as one or more stateless processes"* [141]. Also consider paying for a database-as-a-service provider, which outsources some of the logistics of scale and security to specialists.

8.2 Use Old Technologies

Software technologies change rapidly, so there is no guarantee that software written today will still work a few years from now. This is especially true for research software in academia that is written by students, which tends to go unmaintained after they graduate [116, 139]. Think of how hard it is to obtain, compile, install, configure, and run research prototypes from ten or more years ago. Given these realities, in our experience one way to create long-lasting sustainable software in academia is to use older and more stable technologies that have stood the test of time [90].

Specifically, despite massive advances in server-side web frameworks in the 30+ years since the dawn of the web, the Python Tutor backend still uses CGI (Common Gateway Interface): *"Developed in the early 1990s, CGI was the earliest common method available that allowed a Web page to be interactive."* [145]. Its backend is a simple Python CGI script served via the Apache webserver, which has also been around since nearly the start of the web. As a result, Python Tutor can run on practically any hosting provider, even very low-cost ones, since most Linux distributions come with Apache+CGI. Over the past 12 years we have had to migrate Python Tutor across several providers, and Apache+CGI 'just works' right away. Hosting costs started at around $10 USD per month, and it is currently around $50 per month since we added a few backup servers.

The Python Tutor frontend is built with similarly old technologies – most notably base JavaScript with jQuery. When we started this project in 2009, there were no modern frontend frameworks [8] (e.g., React, Angular, Vue). Fortunately, browser developers have been meticulous about maintaining backward compatibility (i.e., *"don't break the web"* [149]) so that plain JavaScript from decades ago still runs today. And so far, native web technologies (HTML, CSS, JavaScript) have outlasted all third-party browser plug-ins (e.g., Flash, Silverlight, Java Web Start) over the past three decades.

One good argument for upgrading to newer technologies is that they can be more scalable [77]. For instance, modern server-side frameworks like Node.js and Deno (which use a single main thread with efficient OS event notifications) can handle many more concurrent requests than old-fashioned Apache+CGI (which forks a separate OS process for every incoming request). While this is important for industry-level workloads, academic projects (even widely-used ones) are unlikely to require such scalability. For instance, Python Tutor gets up to 10 incoming http requests per second, which is up to one million requests per day. That is well within range of what Apache+CGI can comfortably handle on a low-cost shared-hosting provider with no manual performance tuning.

8.3 Minimize Dependencies

Software dependencies: Software often depends on libraries and the underlying operating system that it runs upon. And these dependencies will inevitably upgrade over the years, sometimes in non-backward-compatible ways [29]. As a result, it can be hard to keep software with dependencies functioning properly over the span of a decade or more. Thus, we recommend making research software as self-contained as possible with the fewest dependencies.

For Python Tutor, we have both tried to minimize the number of dependencies and, when they are unavoidable, we have bundled the dependencies directly into our code repository. For instance, we bundle all necessary libraries into our codebase instead of installing them from package managers (e.g., pip, npm). We even go as far as including *specific compiled versions* of programming language implementations (e.g., a specific version of the Java virtual machine, the Ruby interpreter, etc.) to ensure that our visualizer backend will continue running into the foreseeable future. We then protect against operating system upgrades by encapsulating our development tools inside of Docker images [20]. This approach totally goes against software development best practices, since it "pins" our dependencies at older versions that do not receive bug-fix or security updates [16, 142]. We do update every few years, but we value long-term stability over always getting the latest software.

Despite our best efforts, we know dependencies are inevitable, especially when doing web development for modern HCI research. Even a single '`npm install`' command can pull in hundreds of JavaScript libraries from across the internet [29]. To cope with the pervasive use of dependencies, we highly recommend using tools such as Docker to create a reproducible and standardized development environment with specific pinned versions of all dependencies installed inside of Docker images. Always specify the exact version numbers of everything and cache them. This way, research team members can work with the *exact same development environment* whether they are on Mac, Windows, or Linux.

Institutional dependencies: As an analogy to software dependencies, not depending on external institutions also makes it easier for one's research software to sustain over time. For online education software such as Python Tutor, it may be tempting to partner with a university, a popular MOOC provider (e.g., Coursera, edX, Udacity), or educational technology companies. Over the past decade, we have explored collaboration opportunities with all these types of institutions. But ultimately we found that we could be a lot more agile if we operated independently and focused on directly providing value to users on our own website. Institutional partnerships take a lot of bureaucratic and logistical finessing, and they might result in at most thousands of new users. In contrast, making the core service work better and facilitating organic word-of-mouth growth across the web has let us reach several orders of magnitude more users with much less effort. Since we have full control over the Python Tutor website, we can launch new features, experiments, and research studies as we wish without first coordinating with the bureaucracies of any external institutions.

That said, we see the value of academic researchers partnering with external institutions to broaden their reach. But it is important to keep in mind that the priorities of such institutions shift over time and that those shifts are not always favorable for individual researchers who want to publish academic papers. Tread carefully.

9 SOFTWARE DEVELOPMENT WORKFLOWS

Our final three design guidelines are about optimizing the *process* of developing research software within academia.

9.1 Single Developer

Academia is an unfavorable setting for creating software that can reach many users and last across many years because:

- It is hard to hire long-term software development staff using grants, which often fund research and not software.
- Professional developers can get better working conditions in industry (e.g., higher pay, larger cohort of peers).
- Thus, students are often the implementers of research software, and they are by definition short-term. Undergrad and master's students must split their time with a full course load and extracurriculars. Ph.D. students (rightly!) prioritize publishing new papers to build their careers rather than maintaining old software for their advisors.

Given these constraints, the way we have kept Python Tutor development going throughout the past decade is by *not* relying on a team of student or staff developers. Instead, we adopted a single-developer workflow where the project's creator (this paper's author) is the only person who works on the software.

Having a single developer greatly simplifies our workflow since there is no need to coordinate with others. Although one person cannot possibly implement software that is as sophisticated as that of a team, this constraint forces us to simplify our design in ways that let Python Tutor scale well (e.g., no user accounts, few user options, no content hosting, stateless architecture). It also completely avoids "design by committee" [147] and better preserves the classic notion of *conceptual integrity* from Turing Award winner Fred Brooks [24]: *"I will contend that conceptual integrity is the most important consideration in system design. It is better to have a system omit certain anomalous features and improvements, but to reflect one set of design ideas, than to have one that contains many good but independent and uncoordinated ideas."*

Our single-developer workflow also saves on personnel costs, which are usually far more expensive than equipment costs for typical software projects. Hiring even one undergraduate student for a summer costs at least $8,000 USD (an NSF REU [101]), which is more than we have spent *in total over the past decade* on server hosting costs (see Section 8 for how we have kept server costs low). Also, it can be hard to get grant funding for research software development, so one must often find such funding from indirect sources, which can be time-consuming to wrangle. Not needing to hire any personnel means that we spend less time applying for funding and more time on actual software development work.

That said, we have experimented with the more traditional model of supervising students on developing Python Tutor features. But even after multiple attempts we have not yet been able to get students to contribute production-quality code that we felt comfortable deploying to the real site. For instance, several students have attempted to extend Python Tutor to support more languages or to create new kinds of visualizations. In all of those cases, they were able to make a prototype that worked well enough for cool demos, but they could not sustain the long-term effort necessary to make it 'just work' on the many edge cases that come up when deployed to a live website with tens of thousands of daily users. Such tedious bug-fixing work does not advance their careers at all, so rational students do not stick around long enough to see it through when they have much more appealing work opportunities available.

Relatedly, one can foster an open-source community of external contributors. We have also attempted to do this in the past by being open to external contributions via GitHub pull requests. Like our experiences with student contributors, though, we have found it even harder to shepherd unknown volunteers from the internet to make code contributions that can handle the many edge cases required to 'just work' in production. Managing a distributed development effort with volunteers can be far harder than managing a co-located team. More broadly, it can involve setting up formal governance structures [154], managing contributor expectations [65], arbitrating community disputes over project direction [64], and many other time-consuming tasks [37, 48].

The only two exceptions to our single-developer workflow were one colleague adding support for Python 3 (which differs from Python 2) and another creating the Java backend. They were full-time instructors of Python and Java, respectively, who used Python Tutor in their teaching, so both were well-qualified to extend it.

Our single-developer workflow is unusual, even within academia. Its obvious weakness is having a bus factor of 1: this project dies if we stop working on it [143]. Thus, a more typical arrangement for long-running academic software projects is for the PI (e.g., faculty) to become a software architect and supervise successive generations of students on doing the implementation and maintenance work. This can be viable if there are good processes in place to hand off between student generations every few years. Otherwise the default is for a project to languish after the main student graduates. An alternative model is to raise funding to hire (even part-time) staff programmers to ensure better continuity. In all those cases, we recommend having the *smallest possible team* since we as academics are not experts at managing larger software development teams.

9.2 Start Specific

Software-based researchers often strive to build systems containing high-level ideas that are likely to generalize, since those make for more compelling academic papers. However, we believe that trying to be too general actually hinders scale and sustainability. To build long-lasting software that can organically grow a large userbase, one must instead *start specific.*

In 2009 we created Python Tutor with a very specific goal in mind: to provide a convenient way for students and instructors (such as ourselves) to walk through Python code step-by-step and see the values of variables. That first release had no visual affordances such as pointers or data structure renderings – it simply printed out values in plain text in an HTML table! It was basically a convenient web-based version of the built-in Python pdb debugger [113]. But even though that first version was very simplistic, it met a concrete user need and spread quickly via word-of-mouth.

Since Python became popular throughout the 2000s as a language for both teaching and software development, we focused specifically on it to hone in on a fast-growing user population. Our userbase grew throughout the early 2010s as more computer science courses switched to using Python and as more online education initiatives launched. As usage continued to rise we generalized by adding other popular languages (Java, JavaScript, C, C++, Ruby), but we believe Python Tutor initially got widespread usage because we started out so specific.

As a counterfactual, imagine if in 2009 we had tried to design a generalizable system upfront that could visualize code in multiple languages using sophisticated visual representations. Or, even more ambitiously, we could design a *general-purpose toolkit* [46, 84, 88] for building run-time code visualizations. While this type of idea could lead to a viable grant proposal or research paper, in practice it would have taken a tremendous amount of implementation work to even get a first working release out. And since we would need to split our efforts to support multiple programming languages and visual representations, it would be hard to put in the engineering effort required to make it 'just work' in the wild. Finally, we would not be close enough to any specific user population to iterate meaningfully with them.

Thus, it is ironic that a project which has contributed to dozens of research papers (Table 2) would likely not have gotten off the ground if we had tried to design it as generalizable research from the beginning. Instead we started it as a super-specific and non-researchy project to fit a niche user need and then only generalized later as it built momentum. *We definitely did not start this project thinking about how we were going to get papers published from it.*

On a mundane but amusing note, even the ultra-specific name of our software helped grow its userbase. It turned out that the original title of the website – "Python Tutor: visualize Python code execution" – was great for SEO (Search Engine Optimization) since lots of people were searching for Python help in the early 2010s as the language grew in popularity in schools and workplaces. We continued this trend by creating landing pages for the five other languages with equally-boring names like "Java Tutor: visualize Java code execution." For reference, in 2020 (last year) 36% of visitors to pythontutor.com came from Google keyword searches. If we had instead designed a generalizable system at the start, we would have also named it something generic, so it would be hard for people to discover via web searches. Companies spend lots of money on SEO and brand marketing, but we accidentally stumbled upon an approach (specific names like Python Tutor, Java Tutor, C++ Tutor, etc.) that grew our userbase with zero money spent on advertising.

9.3 Ignore Most Users

A central tenet of HCI is user-centered design [6] – working closely with people to discover and meet their needs. We followed this standard process when iterating with a few early users, but as Python Tutor gained a large userbase over the years we found it more important to do exactly the opposite: ignore most users.

Hyperfocus: In order to keep software running for over ten years, especially in a resource-constrained academic setting, it is extremely important to tightly focus its scope. The quoted paragraph at the beginning of Section 3 lays out the hyperfocused scope of Python Tutor: it just emulates what programming instructors manually draw on the board when explaining code execution, nothing more.

Over the past decade we have received many feature requests from users via personal emails, GitHub Issues, and user surveys. And we have ignored most of them since they are outside of this hyperfocused scope, so implementing them would lead to *feature creep* [14] and make our software harder to maintain over time. Some frequent suggestions include turning Python Tutor into a full-blown web-based IDE, supporting visual debugging of larger-scale production code, adding learning management system (LMS) features like managing student assignments, exams, grading, and school I.T. systems integration, and hosting galleries of user-generated content. While these features make sense for a commercial product, they would add too much maintenance burden for an academic project. Instead we felt that our time was much better spent on the craft of making a few core features work well.

More broadly, we have not let user sentiments guide our high-level project direction. This quote from the creator of the Clojure programming language captures our feelings well: *"Soon there were dozens, then hundreds, then thousands of people asking questions, looking for clarifications and guidance, and most challengingly, desiring input into the project. When I open sourced Clojure, what I thought I was doing was sharing something I had created in a way, open source, that would provide no barriers to adoption. What I discovered was that open source engenders presumptions of collaborative development, which can be at odds with maintaining a singularity of vision."* [65] In sum, we encourage HCI researchers to push back against pervasive trends of community-driven and user-centered design. You are all the experts in your own respective domains, so lean harder on that expertise to maintain *your* singularity of vision!

Protecting against harassment: Lastly, even if most people are well-meaning, with millions of users there are bound to be ones who communicate online in negative ways. In our experience, this behavior has ranged from expressions of entitlement (e.g., *"you need to fix MY bug right now!"*) to various forms of online harassment directed at us (e.g., threatening messages, trying to find our personal information, contacting our employer to demand our attention).

In our project's early years we embraced user-centered design: we made our name and email address visible on the Python Tutor website, and we encouraged users to post bug reports and feature requests to its public GitHub Issues page. While this was constructive for the first few years, as our userbase grew over time the incoming noise of user demands became too overwhelming. Here we echo the sentiments of the creator of D3, a widely-used visualization toolkit that started in academia [23], in his ten-year retrospective: *"I have deep reservations about the way GitHub and other platforms enable [public issue threads] by default, establishing the unreasonable expectation that unpaid maintainers must immediately, politely, and substantively respond to any and all requests for help. Yes, I can turn off issues, but as a community we need to rethink our norms if we are serious about addressing maintainer burnout."* [22]

To protect ourselves from increased noise and occasional harassment, in 2020 we removed our name and email address from the Python Tutor website and turned off the public GitHub Issues bug tracker so users can no longer comment in publicly-visible threads. Instead we direct all users to send feedback and bug reports to a private Google Form text box. This lets frustrated users vent in an anonymous text box but prevents abusive messages from being publicly visible or landing in our personal email inbox. Since doing so we have found that we can work on this project quietly at our own pace without a continual stream of incoming user inputs.

10 CONCLUSION: PERSONAL REFLECTIONS

I'll switch to first-person to end on a more personal note. As a mid-career academic (I am now an associate professor), one of the most common questions I get from students who work on HCI and computing-related research is: *"Should I spend the time to make my research software more robust, usable, scalable, etc., and aim for widespread adoption?"* The obvious tradeoff is that the students who spend more time polishing up their software will have less time and energy to devote to their next paper-producing projects. And what ultimately counts for career advancement within academia is producing generalizable knowledge via research publications, not creating widely-used software. Thus, the candid answer I give is: *"Probably not. You should get prototypes working well enough to validate your research ideas and maybe make them a bit more robust so you can build on them for follow-up papers. But don't aim for widespread adoption from the start, since there's no proven formula for getting users. Focus on what will help you make the best discoveries."*

Note that even I didn't aim for widespread adoption from the start (see Section 9.2: Start Specific). I created Python Tutor in late 2009 as a procrastination side project during my Ph.D., which was totally unrelated to my dissertation research [53]. Almost nobody used it for the first few years that I put it online. But then a series of lucky external factors contributed to organic user growth, most notably the launch of MOOCs and other popular learn-to-code initiatives around 2012. Only then did I think about making the software more robust and scalable to ride that huge incoming wave.

However, that was also the time when I finished my Ph.D. and started a tenure-track career path. From 2012–2020 I was a postdoc then an assistant professor, which meant that throughout most of my past decade of work on Python Tutor, I had to carefully balance my time spent on software development versus time spent working with my students toward new research. I was well-aware that it would both be unwise for my own career to spend too much time on software development, and it would also be unfair to my students who wanted to publish new papers to launch their own careers. ***The ten design guidelines in this paper helped me strike a fine balance between maintaining widely-used research software and sustaining an early-stage academic career.***

My pragmatic compromise was to diversify my project portfolio while keeping Python Tutor running in the background without too much extra effort (aided by these ten design guidelines). As a result, I was able to use it as a platform to launch new research projects when serendipitous collaboration opportunities arose (see Section 5), but I did not solely rely on it for building my early-faculty career. When I applied for tenure around 2020, only 25% of the papers that I published as an assistant professor were based on Python Tutor, so most of my lab's projects were on different topics.

Reflecting back on the past decade, the way I've been able to keep this project going for so long is by 'sneaking it into' a more traditional Ph.D., postdoctoral, and early faculty career path. There would be no way for me back as a grad student to somehow get a magical ten-year grant to build Python Tutor since the idea wasn't at all novel – code visualization tools had existed for nearly 30 years even back then [130]. In fact, I doubt that I could even get long-term funding nowadays to work on a new research software project with similar ideals – focused on providing direct value to users rather than on producing publishable papers right away.

So why try to do this in academia at all? Wouldn't this kind of 'product-oriented' software development work be better done at a company? No. In my experience, there are many kinds of highly-useful software that do not have marketable value, so companies do not fund their development. A code visualization tool like Python Tutor is one of them. Over the past decade I have been closely tracking what technology companies have developed in terms of tools for learning to code. Even with billions of dollars of collective funding (within big companies, from VCs for startups, and from philanthropists for nonprofits like Khan Academy), to my knowledge *no company has ever built its own code visualization tool.* (But a few have integrated Python Tutor into their products.) This suggests that such tools are not marketable and will likely not be developed in industry. But clearly they have proven to be very useful for both learners and researchers, as Python Tutor has shown with its millions of users and dozens of papers built upon it. I believe if Python Tutor wasn't developed in academia, then it wouldn't exist. An alternative is to develop it as an open-source side project while working in an industry job (which I seriously considered), but such indie efforts can be immensely hard to sustain long-term [37].

And that's why I think academia could become the best place to support *long-term hybrid research+product work* like Python Tutor. But we're not nearly there yet since current mechanisms for funding, publication, and career advancement make this path tricky to navigate. In the coming years, I hope we can work toward incentivizing such projects because I believe academia can lead the way in fostering this sort of impactful long-term work that is hard to sustain in typical industry or open-source settings.

More broadly, zooming out beyond my own story: Should we even aim to design research software for scale and sustainability? Or are shorter-term prototypes better for demonstrating exciting new ideas that industry can later pick up on if there is enough market interest? How much should academia value research software that gains a large userbase? Specifically, how should we weigh the more tangible impacts of widespread usage versus the more intellectual impacts of the underlying research ideas? Likewise, should we make software-related contributions count more in hiring and promotion decisions? Or is it actually unfair (and a sign of technocentrism [106]) to privilege the role of software so much, since many types of valuable research contributions [150] are not as software-driven? There are no simple answers to these questions, but we should definitely discuss them now as software continues to pervade more and more domains of research in the coming years. Students, postdocs, and lab staff across many fields are working day-to-day as *research software developers*, so they deserve our thoughtful consideration on these hard questions.

ACKNOWLEDGMENTS

Long before starting Python Tutor, I spent my formative years as a student surrounded by mentors who valued making high-quality research software that scaled and sustained over time. My undergraduate research advisors Eric Klopfer and Michael Ernst have maintained long-running software in their labs for over two decades, and they instilled in me the value of making even my early projects 'just work' for actual users. My Ph.D. advisor Dawson Engler led by example as well and always pushed me to get my software analysis tools working on real code (the title of this paper is an homage to his 2010 paper, *"A Few Billion Lines of Code Later ..."* [19]). Also, my other Ph.D. committee members Margo Seltzer and Jeffrey Heer are both well-known for pioneering widely-used open-source software in their respective fields. All of these role models have tremendously influenced my research tastes as a student, even as I ended up branching out to very different lines of work.

Special thanks to all the people who encouraged me to get Python Tutor off the ground in its fragile early years. Brad Miller, Suzanne Rivoire, and Peter Wentworth were amongst the first instructors who used it and gave me hope that others might like it too. Peter Wentworth and his students made an early port to Python 3, and John DeNero worked closely with me to get it working well enough for widespread deployment in 2012. David Pritchard and Will Gwozdz created the Java visualizer backend shortly afterward. Peter Norvig, Anant Agarwal, and Rob Miller generously gave me the resources, time, and support to keep this project going during my critical postdoctoral years before I started my first faculty job.

Thanks to all the Python Tutor users around the world who gave me feedback and encouraged me to keep at it through the years. Aside from everyone above, I also want to thank Ned Batchelder, Jennifer Campbell, Irene Chen, Frédo Durand, David Evans, Paul Gries, Adam Hartz, Sean Lip, Tomás Lozano-Pérez, Bertram Ludäscher, Fernando Pérez, Andrew Petersen, and Guido van Rossum.

More broadly, discussions with Nadia Eghbal about her work [37] have deeply influenced my thinking and helped me reflect on the challenges of open-source software maintenance. Professors such as John Regehr and Cristian Cadar have inspired me with how they have maintained widely-used research software in their labs over the past decade.

Thanks to Ian Drosos, Jim Hollan, and Sam Lau for feedback on drafts. Finally, thanks to the UIST reviewers and PC members for supporting this very nontraditional paper.

This material is based upon work supported by the National Science Foundation under Grant No. NSF IIS-1845900.

REFERENCES

[1] 2015. Skulpt: Python. Client Side. https://skulpt.org/. Accessed: 2021-04-01.
[2] 2021. Scratch Foundation. https://www.scratchfoundation.org/. Accessed: 2021-04-01.
[3] 2021. Slack for Enterprises. https://slack.com/enterprise. Accessed: 2021-04-01.
[4] 2021. Stack Overflow for Teams Pricing and Plans. https://stackoverflow.com/teams/pricing. Accessed: 2021-04-01.
[5] Dan Abramov. 2017. The melting pot of JavaScript. https://increment.com/development/the-melting-pot-of-javascript/. Accessed: 2021-04-01.
[6] Chadia Abras, Diane Maloney-Krichmar, Jenny Preece, et al. 2004. User-centered design. *Bainbridge, W. Encyclopedia of Human-Computer Interaction. Thousand Oaks: Sage Publications* 37, 4 (2004), 445–456.
[7] Michel Adam, Moncef Daoud, and Patrice Frison. 2019. Direct Manipulation versus Text-Based Programming: An Experiment Report. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education* (Aberdeen, Scotland Uk) *(ITiCSE '19)*. Association for Computing Machinery, New York, NY, USA, 353–359. https://doi.org/10.1145/3304221.3319738
[8] Kamran Ahmed. [n.d.]. Developer Roadmaps: Step by step guides and paths to learn different tools or technologies. https://roadmap.sh/. Accessed: 2021-04-01.
[9] Bedour Alshaigy, Samia Kamal, Faye Mitchell, Clare Martin, and Arantza Aldea. 2015. PILeT: An Interactive Learning Tool To Teach Python. In *Proceedings of the Workshop in Primary and Secondary Computing Education* (London, United Kingdom) *(WiPSCE '15)*. Association for Computing Machinery, New York, NY, USA, 76–79. https://doi.org/10.1145/2818314.2818319
[10] Christine Alvarado, Briana B. Morrison, Barbara Ericson, Mark Guzdial, Brad Miller, and David L. Ranum. 2012. *Performance and use evaluation of an electronic book for introductory Python programming.* Technical Report GT-IC-12-02. Georgia Institute of Technology.
[11] Luís Alves, Dušan Gajić, Pedro Rangel Henriques, Vladimir Ivančević, Vladimir Ivković, Maksim Lalić, Ivan Luković, Maria João Varanda Pereira, Srđan Popov, and Paula Correia Tavares. 2020. C Tutor usage in relation to student achievement and progress: A study of introductory programming courses in Portugal and Serbia. *Computer Applications in Engineering Education* 28, 5 (2020), 1058–1071.
[12] Aivar Annamaa. 2015. Introducing Thonny, a Python IDE for Learning Programming. In *Proceedings of the 15th Koli Calling Conference on Computing Education Research* (Koli, Finland) *(Koli Calling '15)*. Association for Computing Machinery,

New York, NY, USA, 117–121. https://doi.org/10.1145/2828959.2828969

[13] Mohammadreza Azadmanesh and Matthias Hauswirth. 2017. Concept-Driven Generation of Intuitive Explanations of Program Execution for a Visual Tutor. In *2017 IEEE Working Conference on Software Visualization (VISSOFT)*. IEEE Computer Society, Los Alamitos, CA, USA, 64–73. https://doi.org/10.1109/VISSOFT.2017.22

[14] Nick Babich. 2020. Feature Creep: What Causes It and How to Avoid It. https://www.shopify.com/partners/blog/feature-creep. Accessed: 2021-04-01.

[15] Valerie Barr and Deborah Trytten. 2016. Using Turing's Craft Codelab to Support CS1 Students as They Learn to Program. *ACM Inroads* 7, 2 (May 2016), 67–75. https://doi.org/10.1145/2903724

[16] Niccolo Belli. 2019. How should you pin dependencies and why? https://the-guild.dev/blog/how-should-you-pin-dependencies-and-why. Accessed: 2021-04-01.

[17] Emery D. Berger and Benjamin G. Zorn. 2006. DieHard: Probabilistic Memory Safety for Unsafe Languages. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Ottawa, Ontario, Canada) *(PLDI '06)*. Association for Computing Machinery, New York, NY, USA, 158–168. https://doi.org/10.1145/1133981.1134000

[18] Ron Berger. 2003. *An Ethic of Excellence: Building a Culture of Craftsmanship with Students*. Heinemann.

[19] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. 2010. A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World. *Commun. ACM* 53, 2 (Feb. 2010), 66–75. https://doi.org/10.1145/1646353.1646374

[20] Carl Boettiger. 2015. An Introduction to Docker for Reproducible Research. *SIGOPS Oper. Syst. Rev.* 49, 1 (Jan. 2015), 71–79. https://doi.org/10.1145/2723872.2723882

[21] Dana Bojcic. [n.d.]. How often do IP addresses change? (Example). https://www.vicimediainc.com/often-ip-addresses-change/. Accessed: 2021-04-01.

[22] Mike Bostock. 2021. 10 Years of Open-Source Visualization. https://observablehq.com/@mbostock/10-years-of-open-source-visualization. Accessed: 2021-04-01.

[23] Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. 2011. D^3 data-driven documents. *IEEE transactions on visualization and computer graphics* 17, 12 (2011), 2301–2309.

[24] Frederick P. Brooks. 1975. *The Mythical Man-Month: Essays on Software Engineering* (1st ed.). Addison-Wesley Longman Publishing Co., Inc., USA.

[25] Brython. [n.d.]. A Python 3 implementation for client-side web programming. https://brython.info/. Accessed: 2021-04-01.

[26] Raluca Budiu. 2014. Login Walls Stop Users in Their Tracks. https://www.nngroup.com/articles/login-walls/. Accessed: 2021-04-01.

[27] Parmit K. Chilana, Amy J. Ko, and Jacob Wobbrock. 2015. From User-Centered to Adoption-Centered Design: A Case Study of an HCI Research Innovation Becoming a Product. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems* (Seoul, Republic of Korea) *(CHI '15)*. Association for Computing Machinery, New York, NY, USA, 1749–1758. https://doi.org/10.1145/2702123.2702412

[28] Benjamin Cosman, Madeline Endres, Georgios Sakkas, Leon Medvinsky, Yao-Yuan Yang, Ranjit Jhala, Kamalika Chaudhuri, and Westley Weimer. 2020. PABLO: Helping Novices Debug Python Code Through Data-Driven Fault Localization. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education* (Portland, OR, USA) *(SIGCSE '20)*. Association for Computing Machinery, New York, NY, USA, 1047–1053. https://doi.org/10.1145/3328778.3366860

[29] Russ Cox. 2019. Surviving Software Dependencies. *Commun. ACM* 62, 9 (Aug. 2019), 36–43. https://doi.org/10.1145/3347446

[30] Matthew B. Crawford. 2009. *Shop Class as Soulcraft: An Inquiry into the Value of Work*. Penguin Books.

[31] Habibie Ed Dien and Yudistira Dwi Wardhana Asnar. 2018. OPT+Graph: Detection of Graph Data Structure on Program Visualization Tool to Support Learning. In *2018 5th International Conference on Data and Software Engineering (ICoDSE)*. IEEE.

[32] Marija Djokic-Petrovic, David Pritchard, Milos Ivanovic, and Vladimir Cvjetkovic. 2016. IMI Python: Upgraded CS Circles Web-Based Python Course. *Comput. Appl. Eng. Educ.* 24, 3 (May 2016), 464–480. https://doi.org/10.1002/cae.21724

[33] Ian Drosos, Philip J. Guo, and Chris Parnin. 2017. HappyFace: Identifying and Predicting Frustrating Obstacles for Learning Programming at Scale. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC '17)*. 171–179. https://doi.org/10.1109/VLHCC.2017.8103465

[34] Elizabeth Dwoskin, Jeanne Whalen, and Regine Cabato. 2019. Content moderators at YouTube, Facebook and Twitter see the worst of the web — and suffer silently. Washington Post – https://www.washingtonpost.com/technology/2019/07/25/social-media-companies-are-outsourcing-their-dirty-work-philippines-generation-workers-is-paying-price/. Accessed: 2021-04-01.

[35] Brandon Earwood, Jeong Yang, and Young Lee. 2016. Impact of static and dynamic visualization in improving object-oriented programming concepts. In *2016 IEEE Frontiers in Education Conference (FIE)*. IEEE.

[36] Florian Echtler and Maximilian Häußler. 2018. Open Source, Open Science, and the Replication Crisis in HCI. 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, 1–8. https://doi.org/10.1145/3170427.3188395

[37] Nadia Eghbal. 2020. *Working in Public: The Making and Maintenance of Open Source Software*. Stripe Press.

[38] Elvina Elvina, Oscar Karnalim, Mewati Ayub, and Maresha Caroline Wijanto. 2018. Combining program visualization with programming workspace to assist students for completing programming laboratory task. *Journal of Technology and Science Education* 8, 4 (2018), 268–280. https://doi.org/10.3926/jotse.420

[39] Madeline Endres, Georgios Sakkas, Benjamin Cosman, Ranjit Jhala, and Westley Weimer. 2019. InFix: Automatically Repairing Novice Program Inputs. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering* (San Diego, California) *(ASE '19)*. IEEE Press, 399–410. https://doi.org/10.1109/ASE.2019.00045

[40] Barbara Ericson, Steven Moore, Briana Morrison, and Mark Guzdial. 2015. Usability and Usage of Interactive Features in an Online Ebook for CS Teachers. In *Proceedings of the Workshop in Primary and Secondary Computing Education* (London, United Kingdom) *(WiPSCE '15)*. Association for Computing Machinery, New York, NY, USA, 111–120. https://doi.org/10.1145/2818314.2818335

[41] Barbara J. Ericson, Mark J. Guzdial, and Briana B. Morrison. 2015. Analysis of Interactive Features Designed to Enhance Learning in an Ebook. In *Proceedings of the Eleventh Annual International Conference on International Computing Education Research* (Omaha, Nebraska, USA) *(ICER '15)*. Association for Computing Machinery, New York, NY, USA, 169–178. https://doi.org/10.1145/2787622.2787731

[42] Barbara J. Ericson and Bradley N. Miller. 2020. Runestone: A Platform for Free, On-Line, and Interactive Ebooks. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education* (Portland, OR, USA) *(SIGCSE '20)*. Association for Computing Machinery, New York, NY, USA, 1012–1018. https://doi.org/10.1145/3328778.3366950

[43] Barbara J. Ericson, Kantwon Rogers, Miranda Parker, Briana Morrison, and Mark Guzdial. 2016. Identifying Design Principles for CS Teacher Ebooks through Design-Based Research. In *Proceedings of the 2016 ACM Conference on International Computing Education Research* (Melbourne, VIC, Australia) *(ICER '16)*. Association for Computing Machinery, New York, NY, USA, 191–200. https://doi.org/10.1145/2960310.2960335

[44] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. 2015. The Racket Manifesto. In *1st Summit on Advances in Programming Languages (SNAPL 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[45] Sally Fincher, Johan Jeuring, Craig S. Miller, Peter Donaldson, Benedict du Boulay, Matthias Hauswirth, Arto Hellas, Felienne Hermans, Colleen Lewis, Andreas Mühling, Janice L. Pearce, and Andrew Petersen. 2020. Notional Machines in Computing Education: The Education of Attention. In *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education* (Trondheim, Norway) *(ITiCSE-WGR '20)*. Association for Computing Machinery, New York, NY, USA, 21–50. https://doi.org/10.1145/3437800.3439202

[46] James Fogarty. 2017. Code and Contribution in Interactive Systems Research. In *CHI Workshop on HCITools: Strategies and Best Practices for Designing, Evaluating and Sharing Technical HCI Toolkits*.

[47] Eric Fouh, Ville Karavirta, Daniel A. Breakiron, Sally Hamouda, Simin Hall, Thomas L. Naps, and Clifford A. Shaffer. 2014. Design and architecture of an interactive eTextbook–The OpenDSA system. *Science of computer programming* 88 (2014), 22–40.

[48] R. Stuart Geiger, Dorothy Howard, and Lilly Irani. 2021. The Labor of Maintaining and Scaling Free and Open-Source Software Projects. *Proc. ACM Hum.-Comput. Interact.* 5, CSCW1, Article 175 (April 2021), 28 pages. https://doi.org/10.1145/3449249

[49] Tarleton Gillespie. 2018. *Custodians of the Internet: Platforms, content moderation, and the hidden decisions that shape social media*. Yale University Press.

[50] Elena L. Glassman, Jeremy Scott, Rishabh Singh, Philip J. Guo, and Robert C. Miller. 2015. OverCode: Visualizing Variation in Student Solutions to Programming Problems at Scale. *ACM Trans. Comput.-Hum. Interact.* 22, 2, Article 7 (March 2015), 35 pages. https://doi.org/10.1145/2699751

[51] Mitchell Gordon and Philip J. Guo. 2015. Codepourri: Creating visual coding tutorials using a volunteer crowd of learners. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC '15)*. 13–21. https://doi.org/10.1109/VLHCC.2015.7357193

[52] Philip J. Guo. 2006. *A Scalable Mixed-Level Approach to Dynamic Analysis of C and C++ Programs*. Master's thesis. MIT Department of Electrical Engineering and Computer Science, Cambridge, MA.

[53] Philip J. Guo. 2012. *Software Tools to Facilitate Research Programming*. Ph.D. Dissertation. Stanford University.

[54] Philip J. Guo. 2013. Online Python Tutor: Embeddable Web-based Program Visualization for CS Education. In *Proceedings of the 44th ACM Technical Symposium on Computer Science Education* (Denver, Colorado, USA) *(SIGCSE '13)*. ACM, New York, NY, USA, 579–584. https://doi.org/10.1145/2445196.2445368

[55] Philip J. Guo. 2015. Codeopticon: Real-Time, One-To-Many Human Tutoring for Computer Programming. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology* (Charlotte, NC, USA) *(UIST '15)*. Association for Computing Machinery, New York, NY, USA, 599–608. https://doi.org/10.1145/2807442.2807469

[56] Philip J. Guo. 2017. Older Adults Learning Computer Programming: Motivations, Frustrations, and Design Opportunities. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems* (Denver, Colorado, USA) *(CHI '17)*. Association for Computing Machinery, New York, NY, USA, 7070–7083. https://doi.org/10.1145/3025453.3025945

[57] Philip J. Guo. 2018. Non-Native English Speakers Learning Computer Programming: Barriers, Desires, and Design Opportunities. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems* (Montreal QC, Canada) *(CHI '18)*. Association for Computing Machinery, New York, NY, USA, 1–14. https://doi.org/10.1145/3173574.3173970

[58] Philip J. Guo, Julia M. Markel, and Xiong Zhang. 2020. Learnersourcing at Scale to Overcome Expert Blind Spots for Introductory Programming: A Three-Year Deployment Study on the Python Tutor Website. In *Proceedings of the Seventh ACM Conference on Learning @ Scale* (Virtual Event, USA) *(L@S '20)*. Association for Computing Machinery, New York, NY, USA, 301–304. https://doi.org/10.1145/3386527.3406733

[59] Philip J. Guo, Jeffery White, and Renan Zanelatto. 2015. Codechella: Multi-user program visualizations for real-time tutoring and collaborative learning. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC '15)*. 79–87. https://doi.org/10.1109/VLHCC.2015.7357201

[60] F. Maxwell Harper and Joseph A. Konstan. 2015. The MovieLens Datasets: History and Context. *ACM Trans. Interact. Intell. Syst.* 5, 4, Article 19 (Dec. 2015), 19 pages. https://doi.org/10.1145/2827872

[61] Adam J. Hartz. 2012. *CAT-SOOP: A Tool for Automatic Collection and Assessment of Homework Exercises.* Master's thesis. Massachusetts Institute of Technology.

[62] Neil T. Heffernan and Cristina Lindquist Heffernan. 2014. The ASSISTments ecosystem: Building a platform that brings scientists and teachers together for minimally invasive research on human learning and teaching. *International Journal of Artificial Intelligence in Education* 24, 4 (2014), 470–497.

[63] Joseph M. Hellerstein, Jeffrey Heer, and Sean Kandel. 2018. Self-Service Data Preparation: Research to Practice. *IEEE Data Eng. Bull.* 41, 2 (2018), 23–34.

[64] Rich Hickey. 2018. Open Source is Not About You. https://gist.github.com/richhickey/1563cddea1002958f96e7ba9519972d9. Accessed: 2021-04-01.

[65] Rich Hickey. 2020. A History of Clojure. *Proc. ACM Program. Lang.* 4, HOPL, Article 71 (June 2020), 46 pages. https://doi.org/10.1145/3386321

[66] Mark D. Hill. 2016. The "Tire Tracks" Diagram Corrected and Humanized by National Academy Workshop Report. https://cccblog.org/2016/07/27/the-tire-tracks-diagram-corrected-and-humanized-by-national-academy-workshop-report/. Accessed: 2021-04-01.

[67] Ryosuke Ishizue, Kazunori Sakamoto, Hironori Washizaki, and Yoshiaki Fukazawa. 2018. PVC: Visualizing C Programs on Web Browsers for Novices. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education* (Baltimore, Maryland, USA) *(SIGCSE '18)*. Association for Computing Machinery, New York, NY, USA, 245–250. https://doi.org/10.1145/3159450.3159566

[68] Ralph Johnson and John Vlissides. 1995. Design patterns. *Elements of Reusable Object-Oriented Software Addison-Wesley, Reading* (1995).

[69] Hyeonsu Kang and Philip J. Guo. 2017. Omnicode: A Novice-Oriented Live Programming Environment with Always-On Run-Time Value Visualizations. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology* (Québec City, QC, Canada) *(UIST '17)*. Association for Computing Machinery, New York, NY, USA, 737–745. https://doi.org/10.1145/3126594.3126632

[70] Oscar Karnalim and Mewati Ayub. 2017. The Effectiveness of a Program Visualization Tool on Introductory Programming: A Case Study with PythonTutor. *CommIT (Communication and Information Technology) Journal* 11, 2 (2017), 67–76.

[71] Oscar Karnalim and Mewati Ayub. 2017. The use of Python Tutor on programming laboratory session: Student perspectives. *Kinetik: Game Technology, Information System, Computer Network, Computing, Electronics, and Control* (2017), 327–336.

[72] Oscar Karnalim and Mewati Ayub. 2018. A Quasi-Experimental Design to Evaluate the Use of PythonTutor on Programming Laboratory Session. *International Journal of Online Engineering* 14, 2 (2018).

[73] Ada S. Kim and Amy J. Ko. 2017. A Pedagogical Analysis of Online Coding Tutorials. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education* (Seattle, Washington, USA) *(SIGCSE '17)*. Association for Computing Machinery, New York, NY, USA, 321–326. https://doi.org/10.1145/3017680.3017728

[74] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian E. Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica B. Hamrick, Jason Grout, Sylvain Corlay, et al. 2016. *Jupyter Notebooks-a publishing format for reproducible computational workflows.* Vol. 2016.

[75] Michael Kölling, Bruce Quig, Andrew Patterson, and John Rosenberg. 2003. The BlueJ system and its pedagogy. *Computer Science Education* 13, 4 (2003), 249–268.

[76] Robert E. Kraut and Paul Resnick. 2012. *Building successful online communities: Evidence-based social design.* The MIT Press.

[77] KRAZAM. 2020. Microservices. https://www.youtube.com/watch?v=y8OnoxKotPQ. Accessed: 2021-04-01.

[78] Steve Krug. 2014. *Don't Make Me Think, Revisited: A Common Sense Approach to Web Usability* (3rd ed.). New Riders Publishing, USA.

[79] Benjamin Lafreniere, Andrea Bunt, and Michael Terry. 2014. Task-Centric Interfaces for Feature-Rich Software. In *Proceedings of the 26th Australian Computer-Human Interaction Conference on Designing Futures: The Future of Design* (Sydney, New South Wales, Australia) *(OzCHI '14)*. Association for Computing Machinery, New York, NY, USA, 49–58. https://doi.org/10.1145/2686612.2686620

[80] Benjamin Lafreniere, Parmit K. Chilana, Adam Fourney, and Michael A. Terry. 2015. These Aren't the Commands You're Looking For: Addressing False Feedforward in Feature-Rich Software. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology* (Charlotte, NC, USA) *(UIST '15)*. Association for Computing Machinery, New York, NY, USA, 619–628. https://doi.org/10.1145/2807442.2807482

[81] James Landay. 2009. I give up on CHI/UIST. http://dubfuture.blogspot.com/2009/11/i-give-up-on-chiuist.html. Accessed: 2021-04-01.

[82] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization* (Palo Alto, California) *(CGO '04)*. IEEE Computer Society, USA, 75.

[83] Sam Lau and Philip J. Guo. 2020. Data Theater: A Live Programming Environment for Prototyping Data-Driven Explorable Explanations. In *Workshop on Live Programming (LIVE '20)*.

[84] David Ledo, Steven Houben, Jo Vermeulen, Nicolai Marquardt, Lora Oehlberg, and Saul Greenberg. 2018. Evaluation Strategies for HCI Toolkit Research. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems* (Montreal QC, Canada) *(CHI '18)*. Association for Computing Machinery, New York, NY, USA, 1–17. https://doi.org/10.1145/3173574.3173610

[85] Andrew Luxton-Reilly, Emma McMillan, Elizabeth Stevenson, Ewan Tempero, and Paul Denny. 2018. Ladebug: An Online Tool to Help Novice Programmers Improve Their Debugging Skills. In *Proceedings of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education* (Larnaca, Cyprus) *(ITiCSE 2018)*. Association for Computing Machinery, New York, NY, USA, 159–164. https://doi.org/10.1145/3197091.3197098

[86] John Mair. 2013. debug_inspector: A Ruby wrapper for the MRI 2.0 debug_inspector API. https://github.com/banister/debug_inspector. Accessed: 2020-10-10.

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

[88] Nicolai Marquardt, Steven Houben, Michel Beaudouin-Lafon, and Andrew D. Wilson. 2017. HCITools: Strategies and Best Practices for Designing, Evaluating and Sharing Technical HCI Toolkits. In *Proceedings of the 2017 CHI Conference Extended Abstracts on Human Factors in Computing Systems* (Denver, Colorado, USA) *(CHI EA '17)*. Association for Computing Machinery, New York, NY, USA, 624–627. https://doi.org/10.1145/3027063.3027073

[89] Steve McConnell. 2004. *Code Complete, Second Edition.* Microsoft Press, USA.

[90] Dan McKinley. 2018. Choose Boring Technology. http://boringtechnology.club/. Accessed: 2021-04-01.

[91] MDN Web Docs. 2021. referer. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Referer. Accessed: 2021-04-01.

[92] MDN Web Docs. 2021. Web Storage API. https://developer.mozilla.org/en-US/docs/Web/API/Web_Storage_API. Accessed: 2021-04-01.

[93] Bradley N. Miller and David L. Ranum. 2012. Beyond PDF and EPub: Toward an Interactive Textbook. In *Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education* (Haifa, Israel) *(ITiCSE '12)*. Association for Computing Machinery, New York, NY, USA, 150–155. https://doi.org/10.1145/2325296.2325335

[94] Pedro Moraes and Leopoldo Teixeira. 2019. Willow: A Tool for Interactive Programming Visualization to Help in the Data Structures and Algorithms Teaching-Learning Process. In *Proceedings of the XXXIII Brazilian Symposium on Software Engineering* (Salvador, Brazil) *(SBES 2019)*. Association for Computing Machinery, New York, NY, USA, 553–558. https://doi.org/10.1145/3350768.3351303

[95] Brad Myers. 2021. CMU Myers Group Videos: Videos created by Brad Myers and his students at the HCII, SCS, CMU. https://www.youtube.com/playlist?list=PL3856C8FlIWfr_tX8CMUhOJvl34ylClgb. Accessed: 2021-04-01.

[96] National Academies of Sciences, Engineering, and Medicine. 2020. *Information Technology Innovation: Resurgence, Confluence, and Continuing Impact.* The National Academies Press, Washington, DC. https://doi.org/10.17226/25961

[97] Greg L. Nelson, Benjamin Xie, and Amy J. Ko. 2017. Comprehension First: Evaluating a Novel Pedagogy and Tutoring System for Program Tracing in CS1. In *Proceedings of the 2017 ACM Conference on International Computing Education Research* (Tacoma, Washington, USA) *(ICER '17)*. Association for Computing Machinery, New York, NY, USA, 2–11. https://doi.org/10.1145/3105726.3106178

[98] Nicholas Nethercote and Julian Seward. 2007. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Diego, California, USA) *(PLDI '07)*. Association for Computing Machinery, New York, NY, USA, 89–100. https://doi.org/10.1145/1250734.1250746

[99] Casey Newton. 2019. The Trauma Floor: The Secret Lives of Facebook Moderators in America. The Verge – https://www.theverge.com/2019/2/25/18229714/cognizant-facebook-content-moderator-interviews-trauma-working-conditions-arizona. Accessed: 2021-04-01.

[100] Node.js v15.13.0 documentation. [n.d.]. Debugger. https://nodejs.org/api/debugger.html. Accessed: 2021-04-01.

[101] NSF. 2020. Dear Colleague Letter: Research Experiences for Undergraduates (REU) and Research Experiences for Teachers (RET) Supplemental Funding in Computer and Information Science and Engineering. https://www.nsf.gov/pubs/2021/nsf21028/nsf21028.jsp. Accessed: 2021-04-01.

[102] Society of Research Software Engineering. [n.d.]. https://society-rse.org/. Accessed: 2021-04-01.

[103] Dan R. Olsen. 2007. Evaluating User Interface Systems Research. In *Proceedings of the 20th Annual ACM Symposium on User Interface Software and Technology* (Newport, Rhode Island, USA) *(UIST '07)*. Association for Computing Machinery, New York, NY, USA, 251–258. https://doi.org/10.1145/1294211.1294256

[104] Oracle Java Documentation. 2020. Java Debug Interface. https://docs.oracle.com/javase/7/docs/jdk/api/jpda/jdi/index.html?overview-summary.html. Accessed: 2021-04-01.

[105] Oracle Java Documentation. 2021. Java Web Start. https://docs.oracle.com/javase/8/docs/technotes/guides/javaws/. Accessed: 2021-04-01.

[106] Seymour Papert. 1988. A critique of technocentrism in thinking about the school of the future. In *Children in the information age*. Elsevier, 3–18.

[107] Miranda C. Parker, Kantwon Rogers, Barbara J. Ericson, and Mark Guzdial. 2017. Students and Teachers Use An Online AP CS Principles EBook Differently: Teacher Behavior Consistent with Expert Learners. In *Proceedings of the 2017 ACM Conference on International Computing Education Research* (Tacoma, Washington, USA) *(ICER '17)*. Association for Computing Machinery, New York, NY, USA, 101–109. https://doi.org/10.1145/3105726.3106189

[108] Roger D. Peng and Stephanie C. Hicks. 2020. Reproducible Research: A Retrospective. arXiv:2007.12210 [stat.OT]

[109] Whitney Phillips. 2015. *This is why we can't have nice things: Mapping the relationship between online trolling and mainstream culture.* The MIT Press.

[110] Josh Pollock, Jared Roesch, Doug Woos, and Zachary Tatlock. 2019. Theia: Automatically Generating Correct Program State Visualizations. In *Proceedings of the 2019 ACM SIGPLAN Symposium on SPLASH-E* (Athens, Greece) *(SPLASH-E 2019)*. Association for Computing Machinery, New York, NY, USA, 46–56. https://doi.org/10.1145/3358711.3361625

[111] POSIX Programmer's Manual. [n.d.]. setrlimit. https://linux.die.net/man/3/setrlimit. Accessed: 2021-04-01.

[112] David Pritchard and Troy Vasiga. 2013. CS Circles: An in-Browser Python Course for Beginners. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education* (Denver, Colorado, USA) *(SIGCSE '13)*. Association for Computing Machinery, New York, NY, USA, 591–596. https://doi.org/10.1145/2445196.2445370

[113] Python 3.9.2 documentation. [n.d.]. Python bdb – debugger framework. https://docs.python.org/3/library/bdb.html. Accessed: 2021-04-01.

[114] Python 3.9.2 documentation. [n.d.]. Python Built-in Types: Mapping Types – dict. https://docs.python.org/3/library/stdtypes.html#typesmapping. Accessed: 2021-04-01.

[115] Python Tutor. 2021. FAQ and Unsupported Features. http://pythontutor.com/faq.html. Accessed: 2021-04-01.

[116] Roman Rädle and Clemens Nylandsted Klokmose. 2017. Paper accepted, toolkit abandoned. In *CHI Workshop on HCITools: Strategies and Best Practices for Designing, Evaluating and Sharing Technical HCI Toolkits.*

[117] Eric S. Raymond. 2001. *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary.* O'Reilly & Associates, Inc., USA.

[118] Katharina Reinecke and Krzysztof Z. Gajos. 2015. LabintheWild: Conducting Large-Scale Online Experiments With Uncompensated Samples. In *Proceedings of the 18th ACM Conference on Computer Supported Cooperative Work & Social Computing* (Vancouver, BC, Canada) *(CSCW '15)*. Association for Computing Machinery, New York, NY, USA, 1364–1378. https://doi.org/10.1145/2675133.2675246

[119] Ruan Reis, Gustavo Soares, Melina Mongiovi, and Wilkerson L Andrade. 2019. Evaluating Feedback Tools in Introductory Programming Classes. In *2019 IEEE Frontiers in Education Conference (FIE)*. IEEE.

[120] Felipe Restrepo-Calle, Jhon J Ramírez-Echeverry, and Fabio A González. 2018. UNCode: Interactive System for Learning and Automatic Evaluation of Computer Programming Skills. In *EDULEARN18 Proceedings* (Palma, Spain) *(10th International Conference on Education and New Learning Technologies)*. IATED, 6888–6898. https://doi.org/10.21125/edulearn.2018.1632

[121] Felipe Restrepo-Calle, Jhon J Ramírez-Echeverry, and Fabio A González. 2020. Using an interactive software tool for the formative and summative evaluation in a computer programming course: an experience report. *Global Journal of Engineering Education* 22, 3 (2020).

[122] Zak Risha and Peter Brusilovsky. 2020. Making it Smart: Converting Static Code into an Interactive Trace Table. In *Proceedings of Sixth SPLICE Workshop.*

[123] 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 (Jan. 2017), 341–350. https://doi.org/10.1109/TVCG.2016.2599030

[124] Jeremy Scott, Philip J. Guo, and Randall Davis. 2014. A Direct Manipulation Language for Explaining Algorithms. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC '14)*. IEEE, 45–48.

[125] Ben Shneiderman. 2017. The Growth of HCI and User Interface/Experience Design: Presented as a Tire-Tracks Diagram. https://medium.com/@benbendc/a-tire-tracks-diagram-for-e75be51b9bda. Accessed: 2021-04-01.

[126] Teemu Sirkiä. 2018. Jsvee & Kelmu: Creating and tailoring program animations for computing education. *Journal of Software: Evolution and Process* 30, 2 (2018), e1924.

[127] Rebecca Smith, Terry Tang, Joe Warren, and Scott Rixner. 2019. Auto-Generating Visual Exercises for Learning Program Semantics. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education* (Aberdeen, Scotland Uk) *(ITiCSE '19)*. Association for Computing Machinery, New York, NY, USA, 360–366. https://doi.org/10.1145/3304221.3319741

[128] Diogo Soares, Maria João Varanda Pereira, and Pedro Rangel Henriques. 2021. Integrating a Graph Builder into Python Tutor. In *Second International Computer Programming Education Conference (ICPEC 2021) (Open Access Series in Informatics (OASIcs), Vol. 91)*, Pedro Rangel Henriques, Filipe Portela, Ricardo Queirós, and Alberto Simões (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 6:1–6:15. https://doi.org/10.4230/OASIcs.ICPEC.2021.6

[129] Juha Sorva. 2020. personal communication.

[130] Juha Sorva, Ville Karavirta, and Lauri Malmi. 2013. A Review of Generic Program Visualization Systems for Introductory Programming Education. *ACM Trans. Comput. Educ.* 13, 4, Article 15 (Nov. 2013), 64 pages. https://doi.org/10.1145/2490822

[131] Juha Sorva and Teemu Sirkiä. 2010. UUhistle: A Software Tool for Visual Program Simulation. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research* (Koli, Finland) *(Koli Calling '10)*. Association for Computing Machinery, New York, NY, USA, 49–54. https://doi.org/10.1145/1930464.1930471

[132] StackOverflow. [n.d.]. Questions tagged [d3.js]. https://stackoverflow.com/tags/d3.js/. Accessed: 2021-04-01.

[133] William Stein. 2019. Should I Resign From My Full Professor Job To Work Fulltime On Cocalc? https://blog.cocalc.com/2019/04/12/should-i-resign-from-my-full-professor-job-to-work-fulltime-on-cocalc.html. Accessed: 2021-04-01.

[134] Chris Stolte, Diane Tang, and Pat Hanrahan. 2008. Polaris: A System for Query, Analysis, and Visualization of Multidimensional Databases. *Commun. ACM* 51, 11 (Nov. 2008), 75–84. https://doi.org/10.1145/1400214.1400234

[135] Ryo Suzuki, Gustavo Soares, Andrew Head, Elena Glassman, Ruan Reis, Melina Mongiovi, Loris D'Antoni, and Bjoern Hartmann. 2017. TraceDiff: Debugging unexpected code behavior using trace divergences. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC '17)*. IEEE, 107–115.

[136] Terry Tang, Scott Rixner, and Joe Warren. 2014. An Environment for Learning Interactive Programming. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education* (Atlanta, Georgia, USA) *(SIGCSE '14)*. Association for Computing Machinery, New York, NY, USA, 671–676. https://doi.org/10.1145/2538862.2538908

[137] Kyle Thayer, Philip J. Guo, and Katharina Reinecke. 2018. The Impact of Culture on Learner Behavior in Visual Debuggers. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC '18)*.

[138] The Glossary of Human Computer Interaction. [n.d.]. Walk-up-and-use system. https://www.interaction-design.org/literature/book/the-glossary-of-human-computer-interaction/walk-up-and-use-system. Accessed: 2021-04-01.

[139] Chat Wacharamanotham, Lukas Eisenring, Steve Haroz, and Florian Echtler. 2020. Transparency of CHI Research Artifacts: Results of a Self-Reported Survey. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems* (Honolulu, HI, USA) *(CHI '20)*. Association for Computing Machinery, New York, NY, USA, 1–14. https://doi.org/10.1145/3313831.3376448

[140] Adam Wiggins. 2017. The Twelve-Factor App: IV. Backing services. https://12factor.net/backing-services. Accessed: 2021-04-01.

[141] Adam Wiggins. 2017. The Twelve-Factor App: VI. Processes. https://12factor.net/processes. Accessed: 2021-04-01.

[142] Gentoo Linux Wiki. 2021. Why not bundle dependencies. https://wiki.gentoo.org/wiki/Why_not_bundle_dependencies. Accessed: 2021-04-01.

[143] Wikipedia. 2021. Bus factor. https://en.wikipedia.org/wiki/Bus_factor. Accessed: 2021-04-01.

[144] Wikipedia. 2021. Closure (computer programming). https://en.wikipedia.org/wiki/Closure_(computer_programming). Accessed: 2021-04-01.

[145] Wikipedia. 2021. Common Gateway Interface. https://en.wikipedia.org/wiki/Common_Gateway_Interface. Accessed: 2021-04-01.

[146] Wikipedia. 2021. Create, read, update and delete. https://en.wikipedia.org/wiki/Create,_read,_update_and_delete. Accessed: 2021-04-01.

[147] Wikipedia. 2021. Design by committee. https://en.wikipedia.org/wiki/Design_by_committee. Accessed: 2021-04-01.

[148] Wikipedia. 2021. Universally unique identifier. https://en.wikipedia.org/wiki/Universally_unique_identifier. Accessed: 2021-04-01.

[149] Allen Wirfs-Brock and Brendan Eich. 2020. JavaScript: The First 20 Years. *Proc. ACM Program. Lang.* 4, HOPL, Article 77 (June 2020), 189 pages. https://doi.org/10.1145/3386327

[150] Jacob O. Wobbrock. 2016. Research Contribution Types in Human-Computer Interaction. *The Information School, University of Washington* (2016).

[151] Jeong Yang, Young Lee, and Kai H. Chang. 2018. Evaluations of JaguarCode: A web-based object-oriented programming environment with static and dynamic visualization. *Journal of Systems and Software* 145 (2018), 147–163. https://doi.org/10.1016/j.jss.2018.07.037

[152] Jeong Yang, Young Lee, and David Hicks. 2016. Synchronized static and dynamic visualization in a web-based programming environment. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*. IEEE.

[153] Jeong Yang, Young Lee, David Hicks, and Kai H Chang. 2015. Enhancing object-oriented programming education using static and dynamic visualization. In *2015 IEEE Frontiers in Education Conference (FIE)*. IEEE.

[154] Amy X. Zhang, Grant Hugh, and Michael S. Bernstein. 2020. PolicyKit: Building Governance in Online Communities. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*. Association for Computing Machinery, New York, NY, USA, 365–378. https://doi.org/10.1145/3379337.3415858

[155] Daniel Zingaro, Yuliya Cherenkova, Olessia Karpova, and Andrew Petersen. 2013. Facilitating Code-Writing in PI Classes. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education* (Denver, Colorado, USA) *(SIGCSE '13)*. Association for Computing Machinery, New York, NY, USA, 585–590. https://doi.org/10.1145/2445196.2445369