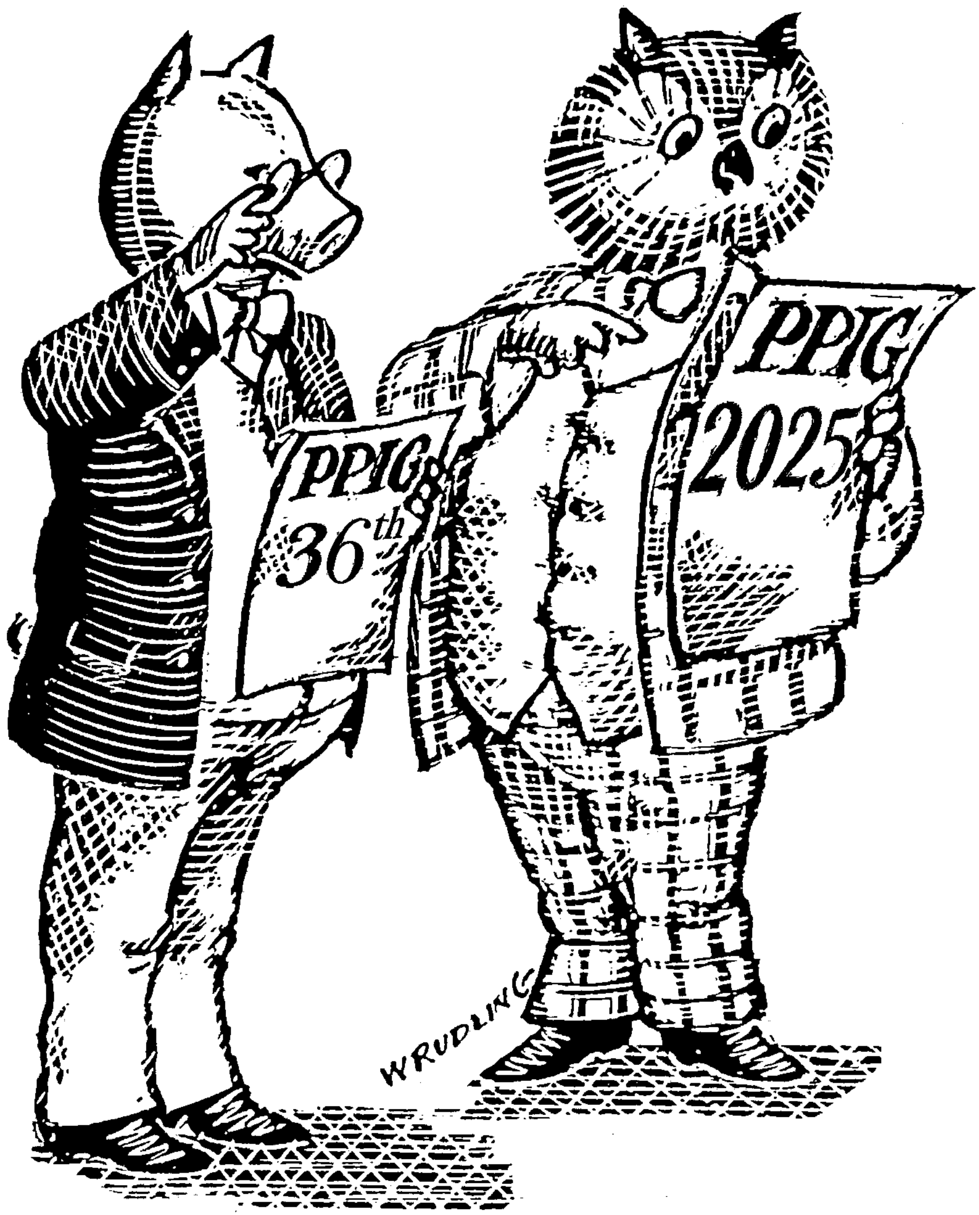
PPIG 2025

Proceedings of the 36th Annual Workshop of the Psychology of Programming Interest Group

8 - 10 September 2025

JetBrains Research, Belgrade, Serbia



Edited by
Ilya Zakharov, Vladimir Kovalenko, Agnia Sergeyuk, Mariana Marasoiu, and Luke Church

PPIG 2025
Programme & Proceedings Index

	Design opportunities for the psychology of programming after AI Clayton Lewis	120
	Vibing: Low viscosity programming with aliens Luke Church	
15:30 - 16:00	*Coffee break*	
16:00 - 17:30	*Industry Panel*	
	Academic Research Contributions to Software Development: Successes and Challenges from the Human-AI Experience Team	
	What is the role of the large-scale developer surveys	
	Increasing Developer Empathy through User Integration in Product Teams	
	Bringing Cognitive Research into Language Design: The Case of Kotlin	
	Open discussion	

Wednesday, 10 September

10:00 - 13:00	**Belgrade graffiti art tour**	
13:00 - 14:00	*Lunch*	
14:00 - 15:00	*Keynote*: **Language and Mind: A Complex Dynamic Relation** Dusica Filipovic Durdevic, University of Belgrade	55
15:00 - 15:30	*Coffee break*	
15:30 - 17:20	*Talks*	
	Towards a Model of Library Use Ava Heinonen	38
	Analogy between Interdisciplinarians and Business Analysts in IT Anna Bobkowska	45
	Purpose, Transparency and Challenge: Self-taught programming in adolescence Jan Dittrich and Felienne Hermans	65
	LUCY-learn: Visualising Image Classifiers for Education in Machine Learning Isabel Millar and Alan Blackwell	78
17:20 - 17:30	*PPIG Prizes & close*	

The Triadic Nature Of Code Review

Alberto Bacchelli
University of Zurich
bacchelli@ifi.uzh.ch

Keynote abstract

This talk explores the multifaceted nature of code review in software development, delving into its three core dimensions: technical, social, and cognitive. While commonly perceived as a straightforward process aimed at bug detection, the talk will show how code review plays a broader role, significantly impacting knowledge transfer, team dynamics, and shared code ownership. In addressing the triadic nature of code review, we will discuss its evolution from a tool for code inspection to a critical element of social software engineering. This talk will present highlights from ten years of investigations on code review, with the goal of informing evidence-based code review tools and practices.

Mixed-Initiative Collaborative Workflow Support for Online Tasks

Sophie Claxton
Computer Laboratory
University of Cambridge
sc2370@cam.ac.uk

Alan F. Blackwell
Computer Laboratory
University of Cambridge
afb21@cam.ac.uk

Abstract
We present a mixed-initiative browser extension for semi-automated support of online tasks, which is designed for joint use by a support-seeker and a workflow-encoder. The support-seeker is an individual with mild cognitive deficits who may struggle to complete infrequently repeated online tasks. The workflow-encoder is able to support them in reviewing their task requirements, and automating elements of the task using a simple domain-specific visual language. The novel contribution of this work is in extending classical mixed-initiative models of programming by example (such as Cypher's Eager, and subsequent application to CoScripter) to accommodate social situations in which end-user programming is achieved jointly by pairs of individuals having distinct needs and abilities.

1. Introduction
Almost a quarter of UK adults have "very low" digital capability (Bank, 2024), with older age the best predictor (Hall, Money, Eost-Telling, & McDermott, 2022). Adults over the age of 75 often lack foundational skills (Bank, 2024) and confidence (ITU, 2021; Ofcom, 2024) to operate online. Despite this, support services increasingly move online, with public services such as healthcare and housing benefits among the least accessible (Bank, 2024; Hall et al., 2022). Many individuals who struggle online ask friends or family for help or to use the internet on their behalf (Ofcom, 2024), with 80% of 16-38-year-olds providing support for online tasks at least monthly. However, when the person providing support lacks time and patience to teach, or takes control of the device, the individual's digital capability does not improve, and fragile self-efficacy can be harmed (Richardson, 2018). Moreover, if that support network becomes unexpectedly unavailable, an individual can be left vulnerable.

We describe a system that helps individuals with low digital capability use online services by effectively reproducing in-person support through an end-user development strategy. Digitally-skilled volunteers use the system to encode a task workflow representation that can be used to provide interactive support at a later time. To meet diverse support needs while offering learning opportunities, the system provides different levels of support. Moreover, as individuals may be too overwhelmed to consider adjusting the level of support, the system helps select the appropriate support level. Because volunteers may make mistakes when encoding task workflows, and those needing support may lack the knowledge or confidence to identify them, the system also helps report problems with the support.

We thus address two distinct groups of users: *Support Seekers* with low digital capability, and digitally-skilled *Workflow Encoders*. We focus in particular on Support Seekers with mild cognitive impairment due to ageing, who may have reduced self-efficacy (Wild et al., 2012), deficits in metacognition (Murphy, Schmitt, Caruso, & Sanders, 1987), and increased cognitive load when accessing online services (Hollender, Hofmann, Deneke, & Schmitz, 2010; Oviatt, 2006). Workflow Encoders are relatively skilled individuals who volunteer their expertise to help others access online services. However, they may have limited availability to provide direct support.

2. Background
Tools already exist that can automate task workflows (BardeenAI, n.d.; Ovadia, 2014; Zapier, n.d.) and help users perform tasks online (Adept, 2022). However, these are designed for users familiar with tasks they wish to automate, and are intended to run unattended. We extend ideas introduced by CoScripter (Leshed, Haber, Matthews, & Lau, 2008), a browser extension that allowed users to

record online actions for playback at a later time. We adopt ideas for end-user programming of scripts (Bogart, Burnett, Cypher, & Scaffidi, 2008), recording re-playable actions in a central repository, whilst introducing new features to meet the needs of individuals with low digital capability. Our system relies on Workflow Encoders creating machine-interpretable representations of how to use online services, which we consider as an end-user development task (Kelleher & Pausch, 2005). We provide a Domain-Specific Visual Programming Language (DSVPL) to capture their expert knowledge of website usage (Mernik, Heering, & Sloane, 2005), while improving accessibility through graphical syntax (Baroth & Hartsough, 1995; Maleki, Woodbury, Goldstein, Breslav, & Khan, 2015).

3. Mixed-Initiative Interaction

Given the high cognitive load Support Seekers face, they may lack the metacognitive resources to assess their support needs or identify problems with the support provided. Therefore, the system acts as an intelligent agent, assisting with these secondary tasks. However, intelligent agents that make inaccurate inferences, do not consider the costs of acting, or do not allow the user to refine or control actions, can be more frustrating than helpful (the "Clippy" effect) (Baym, Shifman, Persaud, & Wagman, 2019). Misaligned system actions could lead to confusion and errors, further damaging low self-efficacy and reducing willingness to engage with the system. Mixed-initiative systems (Horvitz, 1999) combine autonomous capabilities with direct user control, allowing either system or user to initiate interactions (Tecuci, Boicu, & Cox, 2007). When deciding what action to perform, the agent considers uncertainty about the user's goal, and estimates costs and benefits of each possible outcome.

Our system differs from previous MII work where the intelligent agent directly automates actions, instead focusing on whether the level of user support should be increased or decreased. We maintain two models, the first oriented only toward Support Seekers, estimating the level of *Metacognitive Support* they need (Figure 1).

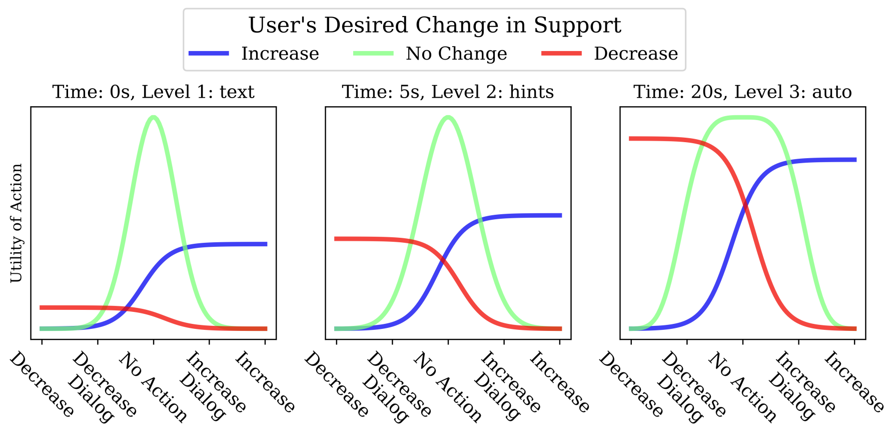


Figure 1 – The Metacognitive Support utility model

The second model is the *Consultation Trigger*, which is novel in modeling costs for two different classes of user (Figure 2). If the Consultation Trigger infers a problem in the support being given, it sends feedback to the Workflow Encoder, who can then fix or improve the workflow. This model consider costs and benefits not only for the Support Seeker, but also the Workflow Encoder and other Support Seekers. For example, it considers the cost to the Workflow Encoder of reporting a problem when there is none, and the future cost to other Support Seekers if a problem is not reported.

4. Implementation

We created a Chrome extension sidebar, providing contextual guidance to Support Seekers for any webpage. A second sidebar provides a DSVPL editor. This front-end was implemented in TypeScript using

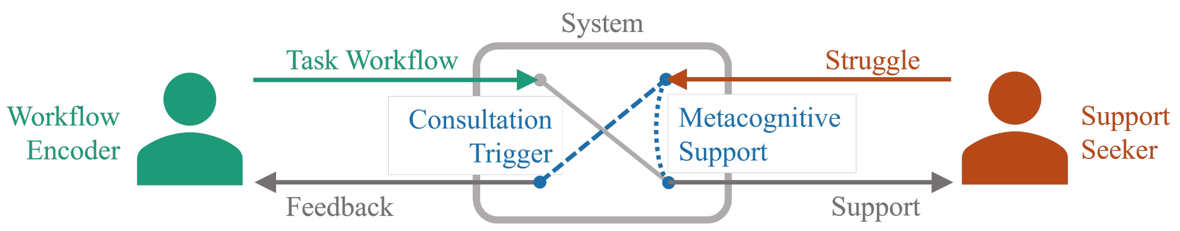
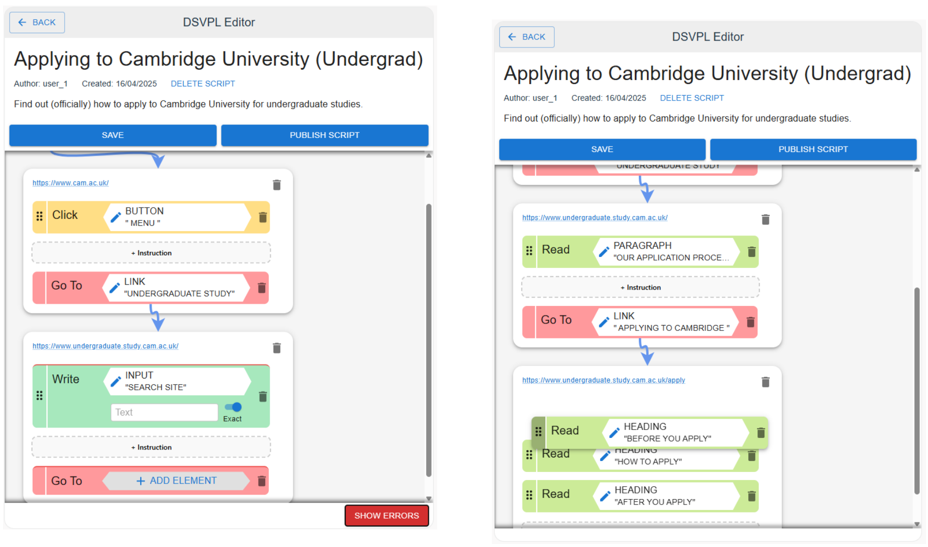


Figure 2 – The two applications of Mixed-Initiative Interaction (blue)

Vite [1], transpiled into JavaScript[2]. The back-end repository for task workflows was implemented using the SQLAlchemy Object Relational Mapper [3] and Python FastAPI framework[4], with task workflows encoded via a REST API[5].

4.1. Domain-Specific Visual Programming Language

The DSVPL serves two functions: an abstract syntax defining the workflow for support, and a concrete syntax edited by Workflow Encoders. Task steps correspond to individual webpage interactions, with sections for different pages, and subsections reflecting user decisions. The abstract syntax includes many step types to reflect the HTML elements that Support Seekers may interact with. Workflow Encoders may be unfamiliar with all HTML elements, so the concrete syntax groups these, for example the `ASTSelectNode`, `ASTCheckNode`, and `ASTRadioNode` in the abstract syntax correspond to a single `CSTSelectNode` for non-text input.


BACK
DSVPL Editor
Applying to Cambridge University (Undergrad)
Author: user_1 Created: 16/04/2025 DELETE SCRIPT
Find out (officially) how to apply to Cambridge University for undergraduate studies.
SAVE
PUBLISH SCRIPT
https://www.cam.ac.uk/
Click BUTTON " MENU "
+ Instruction
Go To LINK "UNDERGRADUATE STUDY"
https://www.undergraduate.study.cam.ac.uk/
Write INPUT "SEARCH SITE"
Text
Exact
+ Instruction
Go To + ADD ELEMENT
SHOW ERRORS
BACK
DSVPL Editor
Applying to Cambridge University (Undergrad)
Author: user_1 Created: 16/04/2025 DELETE SCRIPT
Find out (officially) how to apply to Cambridge University for undergraduate studies.
SAVE
PUBLISH SCRIPT
https://www.undergraduate.study.cam.ac.uk/
Read PARAGRAPH "OUR APPLICATION PROCE...
+ Instruction
Go To LINK " APPLYING TO CAMBRIDGE "
https://www.undergraduate.study.cam.ac.uk/apply
Read HEADING "BEFORE YOU APPLY"
Read HEADING "HOW TO APPLY"
Read HEADING "AFTER YOU APPLY"

(a) Editing the task workflow

(b) Rearranging steps

Figure 3 – The DSVPL editor during task workflow creation

The Workflow Encoder adds steps to a new workflow by clicking the "add instruction" button, followed by the step type. Any step describing webpage interaction must reference the HTML element involved in the interaction. These steps have a slot for an element in the shape of an elongated hexagon (3a). This slot either shows the currently selected element or provides an add-element button. When the Workflow Encoder clicks to add or change an element, the content script activates click listeners on all the webpage elements with tags related to the step type and adds styling to highlight these elements.

[1]https://vite.dev/
[2]using the CRXJS plugin https://crxjs.dev/vite-plugin
[3]https://www.sqlalchemy.org/
[4]https://fastapi.tiangolo.com/
[5]https://fastapi.tiangolo.com/

4.2. Interactive Support

Once a Support Seeker has selected the service they wish to access, the system displays instructions in the side panel, and depending on the level of support, automates actions. The side panel tracks progress through the workflow, with each step translated into natural language based on the step type, including details relevant to that element (for example, when referencing a link, the text content from the `<a>` element is included in the instruction).

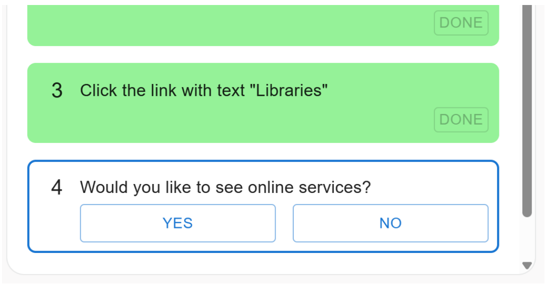

DONE
3 Click the link with text "Libraries"
DONE
4 Would you like to see online services?
YES
NO

Figure 4 – Reaching the decision point in the instructions

Steps are translated sequentially until a user decision node is reached, describing the decision needed, with a yes/no choice (4), to select the next sequence of steps. When an instruction is detected as completed, a green background is added so the Support Seeker can track their progress (4). There are often alternative ways to navigate a website toward the same goal, meaning that a Support Seeker may complete an instruction in a way the extension does not expect and detect. Each instruction therefore provides a "done" button that allows the Support Seeker to report they have completed the task. Elements in the document object model (DOM) may change in ways that break the expected task workflow. For each type of HTML element, attributes are annotated based on whether they can be changed without affecting the user function.

There are three categories of instructions that the content script needs to detect: clicking on an element, scrolling to an element, and providing text input. When a webpage loads, the content script adds a click listener to all elements, so that a detection message can update the Support Seeker's progress. The second category is detected by adding a scroll listener that detects when an element moves entirely into view. The third category is detected by adding input and change event listeners to every input-related element in the DOM. This was straightforward for radio buttons, checkboxes and selections. Autocompleted text inputs require a `MutationObserver` object to detect changes to DOM elements, as do new elements such as popups being added.

The system provides three progressive levels of support:

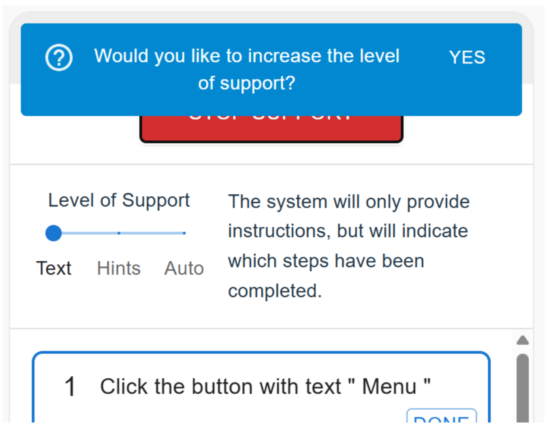
1. Text: written instructions are displayed in the extension's side panel
2. Hints: the system will automatically scroll the webpage to bring the element in the next instruction into view and outline the element with a pink box
3. Auto: the system will perform the interaction described by the next interaction for the Support Seeker, until further decision input is required

The Hints and Auto levels require the extension's content script to modify and interact with the current webpage. The Hints level of support aims to address availability or matching problems between the webpage's options and the Support Seekers' understanding of their goal (Lewis, Polson, Wharton, & Rieman, 1990). The Auto level of support provides a way for Support Seekers to keep progressing with their task, even if they do not understand how to proceed.

4.3. Mixed-Initiative Interaction model

The mixed-initiative interaction model is parameterised by user goals, system actions, observation evidence, and the utility models describing how these are related. Both the Metacognitive Support and

Consultation Trigger components build models in terms of estimated likelihood that the Support Seeker is struggling based on intervals between mouse and scroll events. The side panel tracks the number of steps completed in the last 5 seconds, and the time since the last interaction. If the system determines that the action with the highest expected utility is to change the level of support or initiate a dialog with the Support Seeker, it displays this via a notification. 5 shows the notifications displayed when the system decides to open a dialog about increasing the support and when it decreases the support, respectively. A key principle of MII is that notifications disappear after a few seconds so that the Support Seeker can elect to ignore them.


Would you like to increase the level of support?

YES

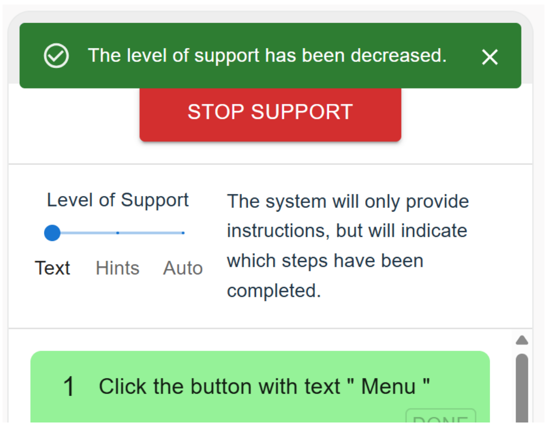
Level of Support

Text Hints Auto

The system will only provide instructions, but will indicate which steps have been completed.

1 Click the button with text " Menu "

(a) Dialog to increase the level of support


The level of support has been decreased.

STOP SUPPORT

Level of Support

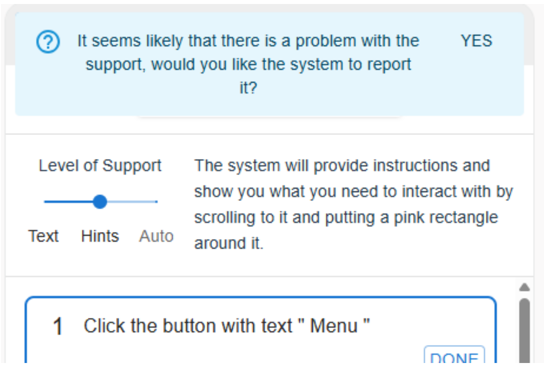
Text Hints Auto

The system will only provide instructions, but will indicate which steps have been completed.

1 Click the button with text " Menu "

(b) Notification for decreased support

Figure 5 – Example Metacognitive Support notifications


It seems likely that there is a problem with the support, would you like the system to report it?

YES

Level of Support

Text Hints Auto

The system will provide instructions and show you what you need to interact with by scrolling to it and putting a pink rectangle around it.

1 Click the button with text " Menu "

DONE

Figure 6 – Dialog to report a problem with the support

The Consultation Trigger model aims to help the Support Seeker detect problems with the support. If the system notes a likely problem, or the Support Seeker reports one, an annotation is recorded in the task workflow, to allow the Workflow Encoder to understand why it may have occurred, as shown in 7. 6 shows the dialog notification displayed to Support Seekers when the Consultation Support determines that a problem is likely and it would be beneficial to first consult the Support Seeker. Notifications for the Consultation Trigger are distinguished from those for Metacognitive Support by styling, but they otherwise behave identically.

5. Evaluation

Two user studies were conducted, one with a sample chosen to represent Support Seekers, and the other representing Workflow Encoders. The studies were approved by the ethics committee of the University of Cambridge Department of Computer Science and Technology. Because Support Seekers potentially belong to vulnerable groups, we took care in the study design to avoid potential distress.

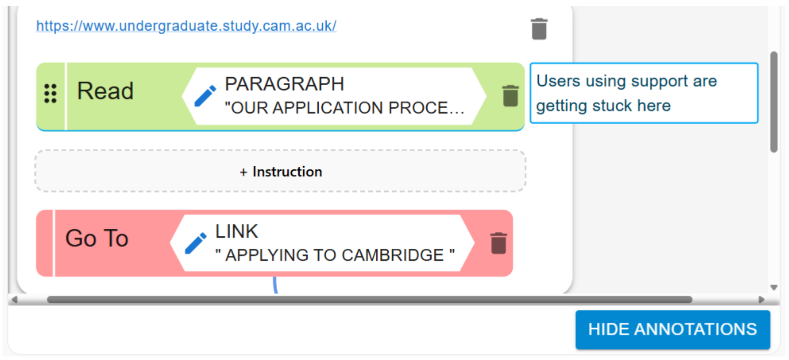

https://www.undergraduate.study.cam.ac.uk/
Read
PARAGRAPH
"OUR APPLICATION PROCE...
Users using support are getting stuck here
+ Instruction
Go To
LINK
" APPLYING TO CAMBRIDGE "
HIDE ANNOTATIONS

Figure 7 – An annotation on an unpublished task workflow

5.1. Participants

Each study involved six participants. Support Seekers (P1-P6) were recruited from the Marple Ramblers walking group and the Marple and District Women's Institute. Screening was based on lack of confidence using online services, and previous experience of receiving assistance. Workflow Encoders (P7-P12) were recruited from students at Cambridge University. Screening selected individuals who were confident using online platforms, had prior experience providing support to others, and were not expert programmers.

5.2. Study Protocol

The main part of the study involved completing tasks while thinking aloud. For Support Seekers, this involved completing an online task, while Workflow Encoders, created a task workflow for that task. The four tasks chosen to represent needs of Support Seekers were:

1. Starting from `www.gov.uk/`, find out the date of the next May bank holiday.
2. Starting from `www.gov.uk/`, fill out[6] the form to apply for a Blue Badge.
3. Starting from `www.ealing.gov.uk/`, either find the digital library services, or find out what opening hours and facilities Ealing Central Library has.
4. Starting from `www.nationalrail.co.uk/`, find out what fast trains are running from Cambridge to London, arriving before 10.00 am on the day of the London marathon.

All participants were walked through the system components relevant to their role. Workflow Encoder participants also completed a simple warm-up task to familiarise themselves with the DSVPL editor. Support Seekers completed two tasks with support and two without (counterbalanced). A pre-task questionnaire asked participants about their experience receiving or providing support, and level of technical expertise. Post-task questionnaires collected feedback on support or the DSVLP editor, as appropriate. Workflow Encoder participants also completed the NASA Task Load Index (NASA-TLX) questionnaire. Qualitative analysis focused on usability issues reported during think-aloud, and feedback in the post-task questionnaire. Texts were segmented and classified using a coding frame defined in advance, with supplementary categories added to analyse recurring themes.

5.3. Interactive Support Effectiveness

Most Support Seekers found the interactive support beneficial, commenting that it was "*very useful for people who haven't got many computer skills*" and that it "*needs extending to other areas* [such as] *passport renewal, and driving licence renewal*" (P4). The "hints" level of support was the most effective, helping participants to locate buttons and links that were not immediately obvious. With the "text" support, half of the participants (P1-P3) focused on completing their task, essentially ignoring the

[6] up to the point where the form redirects to the issuing authority's website

support. In particular, P3 only used the "text" level of support due to their confident attitude towards the tasks and consequently reported being "*not aware when the support assisted* [them]". P1 and P2 subsequently followed up, after having more time to reflect on the support, to say that they thought the approach was "*excellent*" as it provides "*very specific help very easily* [...] *exactly when it's needed so one can get through the procedure not only more quickly but without the stress that almost always occurs*".

When asked if they would prefer to have the interactive support available when navigating websites and whether they thought they would likely use the support in the future, supposing the system was refined for production, the participants' answers ranged from "maybe" to "yes", in line with their digital capability.

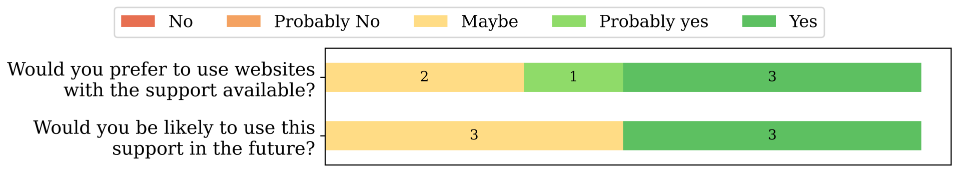


Figure 8 – Preference for using websites with the support available

5.4. Task Workflow Creation Success Rate

Each Workflow Encoder participant was asked to create five task workflows. A task workflow was deemed complete if the system could generate automatic support from it that performed the particular online task (with the necessary user input provided). Every participant successfully created five complete task workflows.

5.5. DSVPL Editor Ease of Use

The NASA-TLX measures workload using six subscales, each rated on a 0-100 scale (the Physical Demand subscale was omitted), as shown in figure 9. Overall workload score (Raw TLX (Hart, 2006)) was calculated as the mean of the subscales. Mean scores for the subscales "Mental Demand", "Effort", and "Frustration" are in the 20th percentile for NASA-TLX results (Hertzum, 2021), and the means for "Temporal Demand", "Performance", and overall TLX are in the 10th percentile. We conclude that the Workflow Encoders experienced low overall workload.

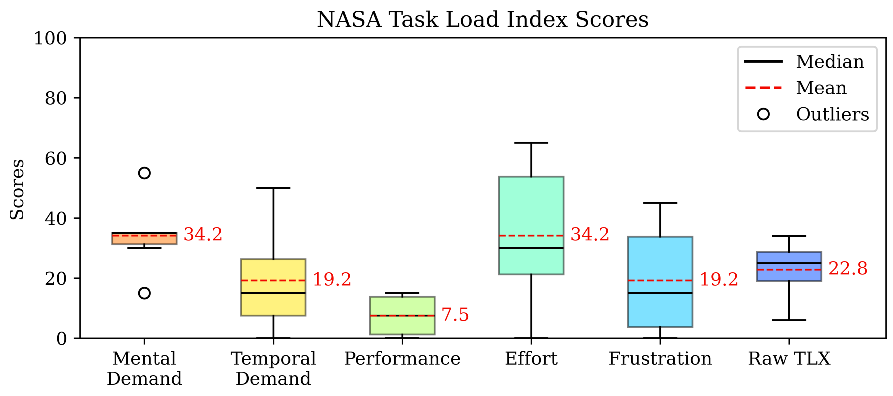


Figure 9 – NASA Task Load Index (TLX) scores

5.6. DSVPL Editor Learnability

After being shown a simple example of a task workflow, all Workflow Encoders spent less than 30 minutes creating the task workflows, without further instruction. The qualitative data collected from the moderated think-aloud protocol and the post-task questionnaire feedback was analysed to assess learnability of the editor. Participants commented that it was "*easy to understand*" (P8) and "*relatively intuitive*" (P12). Initially, participants were unsure which instructions were needed for certain interactions, with all confusing “select” and “click” at least once. Due to the system highlighting relevant website elements when editing an instruction, participants could test out what type of interaction an instruction related to. This was a common strategy, and by the end of the tasks, participants were confident selecting instructions, only becoming unsure when they needed to specify a new type of interaction.

6. Design guidance

Qualitative analysis identified a number of issues that may offer guidance for developers of similar systems in future.

6.1. Support Usability

Ambiguity of ‘Done’ Button. P5 was confused by the purpose of this button, clicking it without having performed the action. P6 had to be prompted to click it, after navigating to the next page via a path the system did not detect. This could be addressed by hiding the button until the system has detected new actions, and animated reveal could prompt users to reflection.

Disruptive Automatic Scrolling. P4 was irritated by automatic scrolling while still reading, describing it as “*gone ballistic*”. While reading time can be added programmatically in workflows, Support Seekers may benefit from (configurable) instruction reading time.

Hidden Instructions. The list of instructions did not scroll automatically as the system detected progress, so participants using smaller displays had to scroll manually to see the next instruction. This could be addressed by defaulting to show previous, current, and next instructions. A progress indicator could provide motivation.

Control Difficulties. In all support levels, participants faced basic difficulties. Many struggled with typing speed and accuracy (as addressed in a previous PPIG paper describing MII for older users (Morris & Blackwell, 2023)). Some became unsure how to proceed after mistakes. A few lacked familiarity with standard web layouts, with one struggling to find the “menu” button despite the pink outline.

6.2. DSVPL Usability

Instruction Naming. Names of some instruction types did not match expectations. The ‘click’ instruction was often used for ‘select’ or ‘go-to’. Some participants failed to recognise that date input required ‘write’. They suggested adding explanations for each instruction type. However, raising the abstraction level so names match terminology familiar to Workflow Encoders may address the problem.

Instruction Parts. Certain instructions require Workflow Encoders to specify extra information. Some participants found the purpose of this unclear. P8, P9 and P12 needed to be prompted to use the ‘exact’ toggle on ‘write’ instructions for interactions that could not be precisely specified. P7 and P10-P12 all initially wrote non-‘yes/no’ questions as prompts for the user decision. The first issue could be addressed by lowering abstraction, e.g. separating ‘write’ instruction into one for exact and one for inexact text.

Direct Manipulation. Participants generally decided how to complete each task before developing their workflow. Knowing which elements they wanted, some interacted with the webpage before clicking the “add element” button. To address this, the element selection mechanism could be automatically activated whenever a new instruction is added.

Website Issues. Participants faced some challenges caused by website structure. For example, to find train times on the National Rail site, a user must click a button that looks like a text input. Participants attempting to use a “write” instruction were confused when that did not work. It is not feasible to anticipate all website idiosyncrasies , but heuristics could be added to suggest solutions when these issues arise.

6.3. Study Limitations

The use of a think-aloud protocol during task completion likely affected performance (Chi, De Leeuw, Chiu, & LaVancher, 1994). This was most evident for Support Seekers, where two participants reported post-study that they may have ignored support because they were “*so busy showing [the experimenter] what I could do*”. Our care to avoid possibly vulnerable participants becoming frustrated meant that some tasks were slightly too easy, resulting in ceiling effects when Support Seekers were able to complete tasks without support.

7. Conclusion

We have presented a novel system that leverages task workflows created by digitally-skilled Workflow Encoders to provide interactive support to Support Seekers with low digital capability. This assists Support Seekers to navigate an online service, providing step-by-step instructions and interactive webpage guidance. We employ mixed-initiative interaction techniques to provide support with secondary tasks: Metacognitive Support helps Support Seekers to select the appropriate level of support, and a Consultation Trigger assists with the detection and subsequent notification to the Workflow Encoder of potential problems. A Domain-Specific Visual Programming Language (DSVPL) enables Workflow Encoders to encode their tacit knowledge of how to access online services.

This project makes several contributions. The first is a novel Domain-Specific Visual Programming Language for specifying user interactions with websites. Secondly, it contributes to mixed-initiative interaction by extending the approach to collaborative support, demonstrating new practical approaches for designing utility models.

7.1. Future Work

This system could be extended further to better meet the needs of Support Seekers and Workflow Encoders. Possible extensions could include: voice-based interaction for Support Seekers with vision impairments or using small-screen devices; increased personalisation of Metacognitive Support and Consultation Trigger models based on the Support Seekers' responses to system actions, or via manual settings; and introducing programming by demonstration (Cypher & Halbert, 1993) mechanisms so that task workflows can be encoded by interacting directly with a site.

8. References

Adept. (2022). *Act-1: Transformer for actions.* (`https://www.adept.ai/blog/act-1`)

Bank, L. (2024). *2024 Consumer Digital Index* (Tech. Rep.). Author. (`https://www.ipsos.com/sites/default/files/ct/publication/documents/2025-01/lb-consumer-digital-index-2024-report_1.pdf` (accessed on Feb. 20, 2025))

Bardeen. (n.d.). (The AI Copilot for GTM teams. `https://www.bardeen.ai/`)

Baroth, E., & Hartsough, C. (1995). Experience report: Visual programming in the real world. *Visual Object Oriented Programming, edited by MM Burnett, A. Goldberg & TG Lewis, Manning Publications, Prentice Hall*, 21–42.

Baym, N., Shifman, L., Persaud, C., & Wagman, K. (2019). Intelligent failures: Clippy memes and the limits of digital assistants. *AoIR Selected Papers of Internet Research.*

Bogart, C., Burnett, M., Cypher, A., & Scaffidi, C. (2008). End-user programming in the wild: A field study of coscripter scripts. In *2008 ieee symposium on visual languages and human-centric computing* (pp. 39–46).

Chi, M. T., De Leeuw, N., Chiu, M.-H., & LaVancher, C. (1994). Eliciting self-explanations improves understanding. *Cognitive science*, *18*(3), 439–477.

Cypher, A., & Halbert, D. C. (1993). *Watch what i do: programming by demonstration.* MIT press.

Hall, A., Money, A., Eost-Telling, C., & McDermott, J. (2022). *Older people's access to digitalised services* (Tech. Rep.). NIHR.

Hart, S. G. (2006). Nasa-task load index (NASA-TLX); 20 years later. In *Proceedings of the human factors and ergonomics society annual meeting* (Vol. 50, pp. 904–908).

Hertzum, M. (2021). Reference values and subscale patterns for the task load index (TLX): a meta-analytic review. *Ergonomics*, *64*(7), 869–878.

Hollender, N., Hofmann, C., Deneke, M., & Schmitz, B. (2010). Integrating cognitive load theory and concepts of human–computer interaction. *Computers in Human Behavior*, *26*(6), 1278-1288. Retrieved from `https://www.sciencedirect.com/science/article/pii/`

S0747563210001718 doi: https://doi.org/10.1016/j.chb.2010.05.031

Horvitz, E. (1999). Principles of mixed-initiative user interfaces. In *Proceedings of the sigchi conference on human factors in computing systems* (pp. 159–166).

ITU. (2021). *Ageing in a digital world–from vulnerable to valuable.* International Telecommunications Union Geneva, Switzerland.

Kelleher, C., & Pausch, R. (2005). Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM computing surveys (CSUR)*, *37*(2), 83–137.

Leshed, G., Haber, E. M., Matthews, T., & Lau, T. (2008). Coscripter: automating & sharing how-to knowledge in the enterprise. In *Proceedings of the sigchi conference on human factors in computing systems* (pp. 1719–1728).

Lewis, C., Polson, P. G., Wharton, C., & Rieman, J. (1990). Testing a walkthrough methodology for theory-based design of walk-up-and-use interfaces. In *Proceedings of the sigchi conference on human factors in computing systems* (pp. 235–242).

Maleki, M., Woodbury, R., Goldstein, R., Breslav, S., & Khan, A. (2015). Designing DEVS visual interfaces for end-user programmers. *Simulation*, *91*(8), 715–734.

Mernik, M., Heering, J., & Sloane, A. M. (2005). When and how to develop domain-specific languages. *ACM computing surveys (CSUR)*, *37*(4), 316–344.

Morris, T., & Blackwell, A. (2023). Prompt programming for large language models via mixed initiative interaction in a GUI. In *34th annual workshop of the psychology of programming interest group (ppig 2023).*

Murphy, M. D., Schmitt, F. A., Caruso, M. J., & Sanders, R. E. (1987). Metamemory in older adults: the role of monitoring in serial recall. *Psychology and Aging*, *2*(4), 331.

Ofcom. (2024). *Adults' media use and attitudes* (Tech. Rep.). Author. (https://www.ofcom.org.uk/media-use-and-attitudes/media-habits-adults/adults-media-use-and-attitudes (accessed on Oct. 8, 2024))

Ovadia, S. (2014). Automate the internet with "if this then that"(IFTTT). *Behavioral & social sciences librarian*, *33*(4), 208–211.

Oviatt, S. (2006). Human-centered design meets cognitive load theory: designing interfaces that help people think. In *Proceedings of the 14th acm international conference on multimedia* (pp. 871–880).

Richardson, J. (2018). I am connected: New approaches to supporting people in later life online. *The Centre for Ageing Better and Good Things Foundation*.

Tecuci, G., Boicu, M., & Cox, M. T. (2007). Seven aspects of mixed-initiative reasoning: An introduction to this special issue on mixed-initiative assistants. *AI Magazine*, *28*(2), 11–11.

Wild, K. V., Mattek, N. C., Maxwell, S. A., Dodge, H. H., Jimison, H. B., & Kaye, J. A. (2012). Computer-related self-efficacy and anxiety in older adults with and without mild cognitive impairment. *Alzheimer's & dementia*, *8*(6), 544–552.

Zapier. (n.d.). (The most connected AI orchestration platform. https://zapier.com/)

Defeating Dragons in the Classroom: Role-playing Games for Teaching Communication and Decision Making Skills in Software Engineering

Tom Beckmann
Hasso Plattner Institute
University of Potsdam
tom.beckmann@*

Eva Krebs
Hasso Plattner Institute
University of Potsdam
eva.krebs@*

Marcel Garus
Hasso Plattner Institute
University of Potsdam
marcel.garus@*

Stefan Ramson
Hasso Plattner Institute
University of Potsdam
stefan.ramson@*

Robert Hirschfeld
Hasso Plattner Institute
University of Potsdam
robert.hirschfeld@*

Abstract

Education in software engineering typically employs a variety of educational formats, including coding exercises, project work, case studies, or lectures. One important aspect of software engineering that appears inadequately addressed by these formats is communication—this includes communication with other programmers, clients, managers, or experts of other disciplines.

To address this lack, we prototyped the design of a tabletop role-playing game (TTRPG) that gives players the opportunity to engage in technical or other challenging conversations in a safe, simulated setting. As in a typical TTRPG, a game leader gives two to three players a description of scenarios, acts as the non-player characters, and lets players react, forming a loop of cooperative storytelling. In first playtesting sessions, we find that the game manages to put players in situations that are unfamiliar and sometimes uncomfortable for them but does so in a way that is educational and still feels playful. We discuss avenues to make use of this format in a classroom setting for a large number of groups, as it heavily relies on the improvisational skills and experience of the game leader.

1. Introduction

Role-playing is an often-used method in education (Rao & Stupans, 2012), entertainment, or even health care. At its core, it involves a person pretending to be another person, either acting as that person or describing their actions as that person. This form of disassociation from one's own person allows players to consider a scenario in a different light, for example, by building empathy for the character they themselves act as or that other people act as.

In entertainment, tabletop role-playing games (TTRPGs) are one of the most famous instances of role-playing. Typically, a group of players and one game leader are gathered around a table for the duration of the play session. The players and the game leader tend to have game-mechanical aids that support the scenario that is being played. For example, in the TTRPG "Dungeons and Dragons" (DnD), players have a character sheet that describes their character, including their strengths and weaknesses. A set of polyhedral dice is used throughout the game to determine success of actions in the scenario, which introduces a degree of uncertainty that prompts players to be spontaneous and flexible.

Even for role-playing games where mechanics play an important role, the core of the game experience remains cooperative storytelling. The game leader describes a scene and players describe how they act in response to that scene. Depending on the scene, these actions may involve dialogs exchanged between players or so-called non-player characters (NPCs) that are brought to life through the game leader. As the game should provide a safe space for the players, the play session should allow them to act in ways that they may otherwise be uncomfortable with, such as portraying an outgoing, boisterous character.

*hpi.uni-potsdam.de

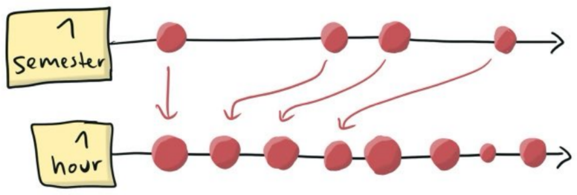


Figure 1 – Compared to a project that students work on over the course of a semester, similar interesting decision points can occur over the course of one hour in our game concept.

In this paper, we explore the players' ability to safely experiment with different roles through a prototype of a role-playing game for teaching software engineering. Players can, for example, assume senior or leadership roles in software engineering settings and are prompted to make decisions that the players may otherwise only get to make in a couple of years' time. Potentially, this may also help them understand and navigate scenarios where they are affected by such decisions in earlier stages of their career.

Communication plays an essential role in software engineering, as much of the engineering aspect encompasses activities that go beyond programming. To illustrate, a software engineering role may need to communicate with non-technical domain experts to understand requirements. Sources of confusion may include a lack of common vocabulary, as the domain experts use domain vocabulary and the software engineer uses software technical vocabulary. Or, goals may be misaligned, as the domain expert emphasizes aspects of the software that require an expensive technical investment that has not been properly accounted for during initial planning. Navigating these situations requires skills of careful communication, as well as an intuition on when to escalate or delegate points of discussion. While best practices and guidelines exist, the opportunities for putting these skills into practice are rare, especially in an educational setting, as illustrated in Figure 1. Our role-playing game prototype, consequently, offers players an opportunity to practice their communication skills and decision-making in a software engineering setting. In alignment with common game development practices, we conducted a first evaluation of our game prototype by performing playtests (Macklin & Sharp, 2016; Schell, 2019).

2. Role Playing in Education

Role-playing exercises are an established educational format (Rao & Stupans, 2012). Such exercises have been used to teach computer science topics that include but are not limited to: Requirements engineering, process management, quality assurance, software architecture, teamwork / soft skills, and software development (Hidalgo, Astudillo, & Castro, 2023). In order to make the most of a role-playing session, a reflection guided by a teacher afterwards is essential (Vykopal, Celeda, Svábenský, Hofbauer, & Horák, 2024). In the following, we will give an overview of how role-playing is used in computer science education, ranging from theater-like performances to scenario evaluations and traditional TTRPGs.

A common form of role-playing is acting as characters in a scene. This form of role-playing has, for instance, been used to teach agile development concepts (Bringula, Elon, Melosantos, & Tarrosa, 2019). Groups of three to four students received a prompt about a software team that uses agile development. The students then had to write their own script and act it out in ten to fifteen minutes. Students learned basic concepts of agile development and liked the cooperative nature of the exercise, but did not learn details about the topic. Additionally, shy students disliked the exercise and other students also noted that role-playing exercises are not an ideal tool for everyone, as they might, e.g., induce anxiety in students with special educational needs such as autism (Hidalgo et al., 2023).

Role-playing can also be used to simulate aspects of project work. For instance, one course aimed to increase communication between students in an online setting and teach about requirements engineering via role-playing (de Macedo, de Lima Fontão, & Gadelha, 2024). Each student team had two roles: they alternated acting as the development team working on the wishes of a customer team, and acting as that customer team for other students. Students reported learning about teamwork and conflict resolution as well as online collaborative tools.

Commercial TTRPGs such as DnD have also been used in education. A course in a game development program included DnD exercises to enhance the students' communication skills (Veldthuis, Koning, & Stikkolorum, 2021). As it was part of a game-oriented program, the lecturers did not alter DnD, but instead embraced its nature as a game to also convey knowledge about storytelling and character creation to students.

In addition to exercises, role-playing has also been used to re-frame entire courses. One computer science course had each student create a character based on traditional TTRPG archetypes such as warriors, rogues, and wizards (Toth & Kayler, 2015). All class activities, including programming exercises, were altered to fit the theme. Students collected experience points from activities that were used for grading. Students liked that some of the new themed tasks, e.g., collecting special items on campus, encouraged them to talk to new students outside of established peer groups. Since the characters were separate from the students, including different names, the course could also add a leaderboard to encourage competition while also keeping students anonymous, if preferred.

Beyond role playing, another format with similar goals presented students with concrete requirements for a software system and they had to implement the interfaces of components by writing on physical index cards and connecting them via strings when they called each other (Blackwell & Arnold, 1997). When requirements change, students would have to untie or cut strings, helping to visualize the value of refactoring and avoiding coupling in system design.

Another alternative to role playing is storytelling—commonly, teaching experiences of how to address complex situations is done through a talk held by an experienced person. These can condense the interesting points and help to learn by example.

Undecided? is a board game that, similarly, highlights points of decision made by a team (Becker, Tsang, Booth, Zhang, & Fagerholm, 2020). By selecting decisions of a set of pre-determined choices, players cause effects that may surface in the short-term or only in the long-term.

3. Game Setup and Rules

Our game prototype aims to bring players into a position where they have to exercise planning, decision-making, and communication skills. We designed the rules for the game through iterative prototyping. We improvised a first session with one game leader and two players who defined a majority of the rules. In two subsequent sessions, we refined the rules and tried out slight variations to arrive at the ruleset we describe below. Notably, a number of problems with the ruleset as described below exist that we discuss in Section 5.

3.1. Setup

Most of the game's setup happens ad-hoc at the table with only a minor burden for preparation on the game leader.

Game Leader Preparation The game leader should prepare the scenario by deciding on or asking the players for themes that the group wants to explore as part of the play session. These themes may include activities such as hiring, crunch time, service outage response, or onboarding. Based on the themes, we recommend that the game leader prepare rough notes for potential complications the players might run into. Attempting to predict solutions to each complication proved to be a useful manner to derive possible further complications.

Role and Character Assignment When the group is gathered, the game leader assigns roles to each player. We recommend that these should imply distinct competencies and potentially also weaknesses, such that it is easier for players to act as their character. For example, if a character possesses extensive

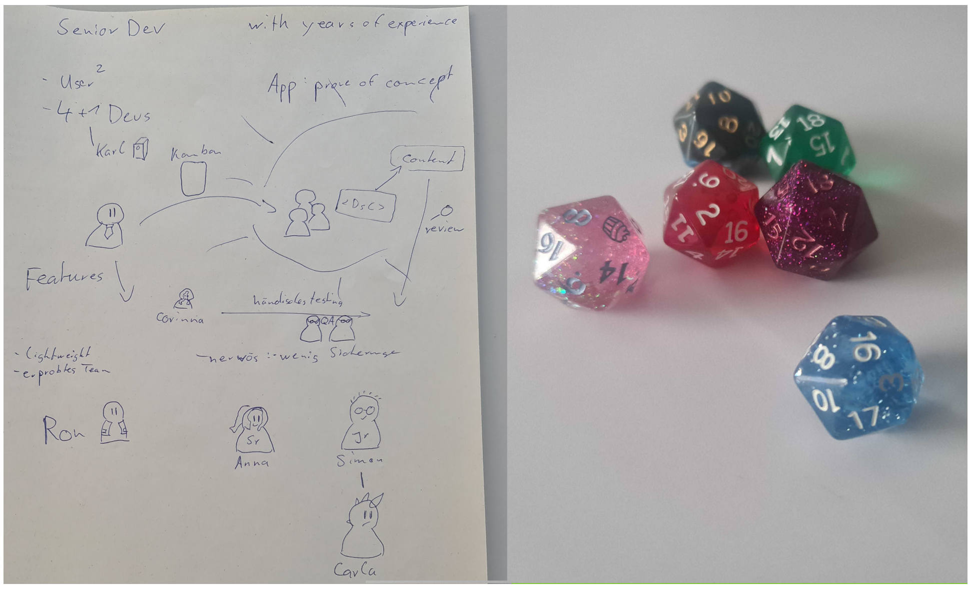


Figure 2 – Left: notes a player took during a play test, including names of characters, their relationships, and processes. Right: 20-sided dice, used in our prototype when outcomes are determined by chance.

experience with migrating cloud infrastructures, this enables the player to act as if they had multiple years of professional work experience in the relevant field without the player themselves knowing all required nuances. Assigning distinct competencies or roles among the players facilitates turn-taking, as players naturally hand the word to other players when their character's area of expertise would be helpful.

Scenario Description Once the roles are established, the game leader proceeds to describe the scenario the players are in: this may include the company they work at, colleagues they might refer to, and the team's current work mode. This description can already imply or state tension that exists in the team's setup, such as limited availability or suboptimal handovers between teams.

3.2. Play

With the characters and scenario established, the game loop begins.

Game Loop The structure of the game, following classic TTRPG rules, can be roughly summarized as follows, also illustrated in Table 1:

1. The game leader describes a setting, consequence, or action.
2. The players ask clarifying questions, if needed.
3. The players either react in-character, describe a reaction, or first discuss among themselves on how to best react, typically out-of-character.
4. The game leader considers the players' reaction and either repeats the loop or asks for a dice roll to determine its outcome before proceeding.

Dice Rolls In our prototype, we used a 20-sided die as seen in Figure 2 to add an element of uncertainty and chance to actions players take. This also supports the game leader, as outcomes to actions that are

Table 1 – Example Sequence of Play

Game Lead establishes scenes, describes "the world"	*"Your colleague from HR comes in and asks what to put into the new tech role's job description."*
Players describe or act out their actions	*"I tell him I'll send a text. I describe the project, tech stack, and I ask the colleague to add text about the company."*
Game Lead calls for dice rolls in important moments	*"To determine who applies, roll a die." (given the preparation, the Lead adds +5)*
Game Lead narrates results	*"It's a 5 (total 10). You get 50 applications." (likely good candidates, but challenging to decide who to interview)*

risky or uncertain do not have to be arbitrarily decided by the game leader. We encourage the game leader to consider the outcome of the dice roll as a tendency, rather than a scale from failure to success. For example, preparation that players took to mitigate risks with the decision should weigh more strongly than a 1 or 20 on the dice roll. Still, a 1, as the worst possible outcome of the dice roll, could present an interesting complication for the players but the game leader should still take care to realistically take into account the stakes of the decision and the players' preparation.

In addition, the game leader may use dice rolls to determine resources or outcomes outside of the players' control. These could include setting the size of a budget if it appears less relevant to the theme the game leader wants to bring across, or the number of applicants to a job posting the players created.

Scenes As described earlier, players and the game leader can either describe how characters act or act out interesting scenes. We encourage the game leader to offer the players opportunities to engage in acted dialog when the game leader thinks the conversation could include interesting challenges but not to force players to do so.

To reach interesting decisions throughout the game, the game leader will need to seek and often improvise adverse actions or circumstances to the players' plans. For example, a supplier may not be on time for an otherwise perfect plan. Or, the players' open communication with a customer may contradict promises another representative of their company has made without their knowledge.

4. Test Runs

We conducted four test runs, also known as playtests, of the game rules.

4.1. First Test Run

During the first test run, three of the co-authors played together, one being the game leader, and two players. The players assumed roles of a CTO at a startup and a Senior Developer who was just hired to bring in expertise for preparing the startup's infrastructure for a much larger scale.

Themes that were touched upon during play included defining roles and distributing or claiming responsibilities, hiring new team members, and mediating personal conflicts during onboarding. In total, the session took 45 minutes, with another 20 minutes afterward for reflection on the play session and the game system.

The players described feeling challenged or uncomfortable with decisions in multiple of scenes or conversations during the game but also that this was likely a good sign, as they appeared at the same time like they might happen in a real work setting. The game thus allowed them to explore the game world's reaction to their decisions without real-world consequences. When asked, the players listed a number of decisions they took throughout the game and indicated that the game encouraged reflection on these decisions.

4.2. Second Test Run

For the second test run, the same game leader led the game for three students in their last Bachelor's semester. This time, the players assumed the roles of three experienced software developers at an IT consultancy. Their current project involved writing drivers and control software for an expensive installation at a zoo. During the project, another lucrative project popped up and their management asked the team if they could create a proof-of-concept on a short timeline.

Themes included testing setups for hardware that is not on-site, communicating timelines with customers, and team organization when faced with the spontaneous new project.

The players appeared less confident than in the previous run. On multiple occasions, they were recalling best practices from software engineering lectures they had recently taken. The game leader made attempts to probe the understanding of the best practices by asking high-level aspects of their applicability and implementation in the scenario. Player feedback on the session and learning success was generally positive but players were struggling to state what they learned when prompted.

4.3. Third Test Run

For our third test run, a different co-author took the game leader role, and three students were selected as players. The players assumed roles of experienced software developers in a larger company that wanted to branch out and develop a new digital communication solution.

Themes included challenging technical requirements, pressure for marketing teams, and coordination or delegating work across departments, as they moved into a product lead role early in the game.

Feedback from the players was mostly concerned with game mechanics and structure of the storyline, which we will address in the next section.

4.4. Workshop Test Run

As part of the PPIG'25 workshop, we were able to let participants of the workshop try the game concept and include the resulting insights as a fourth test run. We asked them to form groups of three, select a game leader among them, and gave them a prepared scenario to run over the course of 45 minutes. In total, we had three groups, of which two played in person and one online.

Feedback was generally positive but it was noted that the involved players were all experienced programmers who had doubts that it would work as well with less experienced programmers. One group was initially struggling with the vague guidelines that our game rules provide but eventually got comfortable with the setting.

For this test run, we gave additional inputs to the players as a one-page website: a summarization of the rules, a scale to help game leaders judge the outcome of dice rolls, and a more explicit framework for accounting for preparation. The additional scale for preparation suggested applying adjustments to dice rolls, ranging from -10 for no preparation and a risky task to $+5$ for a well-prepared team or a well-known task. It was difficult to judge the effect this additional hint had but one team noted that they found it unclear how to use these adjustments. The same team also noted that it is not obvious when dice rolls should be called for.

Another point of difficulty concerned our prepared materials: we suggested separate roles of a "CTO" and "senior developer"—which appeared too similar, such that it was unclear how responsibilities should be split up. One team circumvented the tasks' complexity by simply stating that they would use "AI" for everything. The game leader in that team was unsure how to properly react to that, highlighting the dependency on improvisational skill or technical expertise with the technologies that players want to employ.

5. Discussion

Based on observations and player feedback, we derived a number of observations on the game rules and their use as a teaching tool.

5.1. Dependency on Game Leader

The most relevant issue that became apparent was a strong dependency on the game leader: they would have to be comfortable with improvising reactions of characters and sufficiently knowledgeable of software engineering processes to create realistic scenes for players to engage with.

To attempt to lessen the burden, we created a set of "events". These events comprise an inciting incident, potential consequences for the players, and a model approach for how the players may address the issue, such that the game leader could prepare potential setbacks as the players attempt to resolve the event. In practice, it was difficult to apply the events while the game was running. In the third session, players had started exploring implications of integration with a third-party platform, but the game leader felt unprepared to follow that thread and shifted to a prepared event. Feedback from the sessions suggested creating even more comprehensive scenarios that would allow the game leader to "look up" what should happen in response to a player's unexpected suggestion, and, if missing, at least find inspiration in similar situations. These events could also feature instructions for dice rolls, to serve as examples of what situations may be significant enough for rolling dice.

Similarly, it became clear that it was difficult to improvise interesting and relevant complications to the players' actions. Without those, the game ended up as fast-paced enumeration of scenes where typically one or at most two decisions had to be taken by players before it was resolved. One notable difference we drew from a comparison to TTRPGs like DnD was the lack of movement through space that often prompts the game leader to add "fluff"—details that set the mood of a scene or describe the setting. These often inconsequential details can help the game leader to gain valuable seconds to consider the next important event and how they want to present it to the players. In our game, as the narrative tended to jump from important scene to important scene without much "fluff", the game leader was constantly put in a position of improvising the next events without much buffer.

This dependency on the game leader's improvisational skill and software engineering experience is especially relevant for the game's use in an educational setting, as it makes it difficult to run a session with only few people as game leaders available. Further, the experience and learning outcomes will thus heavily depend on the game leader's familiarity with the game session's themes, whereas for use in a classroom, a degree of consistency would be desirable.

5.2. Impact of Dice Rolls

As described earlier in Section 3.2, dice rolls should only imply a tendency. Throughout the three game sessions, both the game leaders and players struggled with a suitable interpretation of dice rolls. As the range from 1 to 20 suggests a linear scale from worst possible to best possible outcome, numbers 1 through 5 were commonly perceived to yield potentially catastrophic outcomes. As rolling a 1 through 5 has a 25% chance of occurring, this led to some jarring and unrealistic developments in the story.

We attribute this expectation in part to prior experience with other TTRPGs, where dice rolls are typically combined with so-called modifiers that model the character's aptitude for the desired action. In, for example, DnD, it is not uncommon to add a $+10$ to rolls that characters are experts in, skewing the scale heavily toward success.

As we wanted to keep the ruleset light, we omitted any sort of modifiers or character attributes. Rather, the game leader is encouraged to consider the character's prior experiences as preparation for a roll and adjust outcomes accordingly. A future iteration of the game could consider adding a flat bonus to all roles to skew the perception toward success or adopt a different means of adding an element of chance to outcomes. An example may be a die with textual labels as opposed to numbers, or placing a labeled scale on the table during play that assigns, e.g., "minor setback" to the numbers 2 through 6 and thus helps adjust player expectations.

While it would also be possible to drop dice rolls completely, we kept them in our test runs for two reasons. First, as described before, they remove some pressure from the game leader to take decisions that "hurt" the players, shifting the blame to random chance. Second, players unanimously reported that the dice added a feeling of excitement to the game. Notably, we did not try a test run without dice rolls, so the excitement and degree of uncertainty could potentially also be gained just from the unpredictable narration of the game leader.

5.3. Player Decisions

The core purpose of the game revolves around presenting players with interesting decisions and their communication. These decisions may concern who they want to talk to, how they address an issue, or what preparation they take. At several points, players noted that they were missing knowledge to articulate the steps they want to take. For example, in the first test run, the player acting as the senior developer tasked with preparing the infrastructure for scaling was initially struggling to play as that role, as it was unclear to them how to best proceed. In that instance, the game leader offered to present some general steps that they could consider and they proceeded to fill them out with how they figured it might make sense to proceed. Ultimately, while the game leader had to step in to help shape the decision, the player questioned and adapted the proposed steps, leading to interesting insights. This is in contrast to the second test run, where, in a similar situation, the players just said that they want to implement the steps offered by the game leader as suggested. It thus appears to be both a challenge for the game leader to volunteer information at the right level of abstraction and for the players to show willingness and interest to engage with the information. Still, it is clear that participants require a minimum level of experience to adequately derive actions. As one participant put it: the participants "don't know what [they] don't know".

As another concern, players and the game leader had a tendency to resolve conflicts quickly, which prevented interesting decisions from arising. Suggestions from the workshop to address this behavior included preparing "antagonists" in the story that may have conflicting interests with the players or giving players roles that initially appear to follow conflicting goals.

5.4. Other Limitations

In our test runs, we have only played the game with people who had already been working with one another for at least several months and trusted each other. In a classroom setting, it is unlikely that this will always be given. It may thus be necessary to add rules or mechanisms that protect players and ensure that they can act openly and experiment with decisions. Commonly, TTRPGs recommend players to conduct a so-called "Session Zero", where, among other things, themes and topics are collected that should not appear in the game's story. Further, a Session Zero often introduces safety tools that allow players to interrupt the game and discuss elements of the story that they felt uncomfortable with.

6. Conclusion

In this paper, we presented a concept and prototype for a game designed to let students experiment with communication and decisions in a software engineering context in the safe context of a game. Findings from three test runs indicate that the game is effective at engaging players with important decisions and placing them in situations that they would usually only encounter at later career stages. For actual use in a classroom setting, the most relevant unsolved challenge remains to support the game leader to improvise interesting consequences and scenarios, as well as provide them with sufficient knowledge of the software engineering themes they want to explore in the game.

7. References

Becker, C., Tsang, T., Booth, R., Zhang, E., & Fagerholm, F. (2020). Undecided? a board game about intertemporal choices in software projects. In *Proceedings of the 31st annual meeting of the psychology of programming interest group* (pp. 122–133). Retrieved from `https://ppig.org/papers/2020-ppig-31st-becker/`

Blackwell, A. F., & Arnold, H. L. (1997). Simulating a software project: The pop guns go to war. In *Proceedings of the 9th annual meeting of the psychology of programming interest group* (pp. 53–60). Retrieved from `https://www.cl.cam.ac.uk/~afb21/publications/PPIG97.html`

Bringula, R., Elon, R., Melosantos, L., & Tarrosa, J. R. (2019). Teaching agile methodology through role-playing: What to expect and what to watch out. In *Proceedings of the 3rd international conference on education and multimedia technology* (p. 355–359). New York, NY, USA: Association for Computing Machinery. Retrieved from `https://doi.org/10.1145/3345120.3352733` doi: 10.1145/3345120.3352733

de Macedo, G. T., de Lima Fontão, A., & Gadelha, B. (2024). Building soft skills through a role-play based approach for requirements engineering remote education. *J. Braz. Comput. Soc.*, *30*(1), 1–16. Retrieved from `https://doi.org/10.5753/jbcs.2024.3071` doi: 10.5753/JBCS.2024.3071

Hidalgo, M., Astudillo, H., & Castro, L. M. (2023). Challenges to use role playing in software engineering education: A rapid review. In H. Florez & M. León (Eds.), *Applied informatics - 6th international conference, ICAI 2023, guayaquil, ecuador, october 26-28, 2023, proceedings* (Vol. 1874, pp. 245–260). Springer. Retrieved from `https://doi.org/10.1007/978-3-031-46813-1\_17` doi: 10.1007/978-3-031-46813-1_17

Macklin, C., & Sharp, J. (2016). *Games, design and play*. Boston, MA: Addison-Wesley Educational.

Rao, D., & Stupans, I. (2012). Exploring the potential of role play in higher education: development of a typology and teacher guidelines. *Innovations in Education and Teaching International*, *49*(4), 427–436. Retrieved from `https://doi.org/10.1080/14703297.2012.728879` doi: 10.1080/14703297.2012.728879

Schell, J. (2019). *The art of game design: A book of lenses* (3rd ed.). London, England: Routledge.

Toth, D., & Kayler, M. (2015). Integrating role-playing games into computer science courses as a pedagogical tool. In A. Decker, K. Eiselt, C. Alphonce, & J. L. Tims (Eds.), *Proceedings of the 46th ACM technical symposium on computer science education, SIGCSE 2015, kansas city, mo, usa, march 4-7, 2015* (pp. 386–391). ACM. Retrieved from `https://doi.org/10.1145/2676723.2677236` doi: 10.1145/2676723.2677236

Veldthuis, M., Koning, M., & Stikkolorum, D. R. (2021). A quest to engage computer science students: Using dungeons & dragons for developing soft skills. In *CSERC '21: The 10th computer science education research conference, virtual event, the netherlands, november 22 - 23, 2021* (pp. 5–13). ACM. Retrieved from `https://doi.org/10.1145/3507923.3507927` doi: 10.1145/3507923.3507927

Vykopal, J., Celeda, P., Svábenský, V., Hofbauer, M., & Horák, M. (2024). Research and practice of delivering tabletop exercises. In M. Monga, V. Lonati, E. Barendsen, J. Sheard, & J. H. Paterson (Eds.), *Proceedings of the 2024 on innovation and technology in computer science education v. 1, iticse 2024, milan, italy, july 8-10, 2024.* ACM. Retrieved from `https://doi.org/10.1145/3649217.3653642` doi: 10.1145/3649217.3653642

Students' Feedback Requests and Interactions with the SCRIPT Chatbot: Do They Get What They Ask For?

Andreas Scholl
Computer Science
Nuremberg Tech
andreas.scholl@th-nuernberg.de

Natalie Kiesler
Computer Science
Nuremberg Tech
natalie.kiesler@th-nuernberg.de

Abstract

Building on prior research on Generative AI (GenAI) and related tools for programming education, we developed SCRIPT, a chatbot based on ChatGPT-4o-mini, to support novice learners. SCRIPT allows for open-ended interactions and structured guidance through predefined prompts. We evaluated the tool via an experiment with 136 students from an introductory programming course at a large German university and analyzed how students interacted with SCRIPT while solving programming tasks with a focus on their feedback preferences. The results reveal that students' feedback requests seem to follow a specific sequence. Moreover, the chatbot responses aligned well with students' requested feedback types (in 75%), and it adhered to the system prompt constraints. These insights inform the design of GenAI-based learning support systems and highlight challenges in balancing guidance and flexibility in AI-assisted tools.

1. Introduction

Ever since the broad availability of GenAI, we have seen an increasing number of application contexts in computing education, and particularly introductory programming (Prather et al., 2023, 2025). This development is not surprising, as GenAI is capable of passing introductory programming exercises and courses (Geng, Zhang, Pientka, & Si, 2023; Kiesler & Schiffner, 2023a; Savelka, Agarwal, Bogart, & Sakr, 2023). They can generate programming error messages (Leinonen et al., 2023; Sarsa, Denny, Hellas, & Leinonen, 2022; MacNeil et al., 2023), exercises (Jacobs, Peters, Jaschke, & Kiesler, 2025), and formative feedback (Bengtsson & Kaliff, 2023; Kiesler, Lohr, & Keuning, 2024; Jacobs & Jaschke, 2024; Roest, Keuning, & Jeuring, 2023; Lohr, Keuning, & Kiesler, 2025). For example, it has been shown that it is possible to elicit certain types of feedback for programming novices by using specific prompts (Lohr et al., 2025). Moreover, recent GenAI tools have advanced rapidly, and some of them can generate precise, structured, and personalized feedback for novice programmers (Azaiz, Kiesler, & Strickroth, 2024; Phung et al., 2023; Scholl & Kiesler, 2024).

Due to GenAI's potential for learners, several prototypes and tools have been developed to integrate GenAI into programming education. Context-specific tools with feedback for novice learners of programming comprise, for example, CodeAid (Kazemitabaar et al., 2024), Codehelp (Liffiton, Sheese, Savelka, & Denny, 2024), LLM Hint Factory (Xiao, Hou, & Stamper, 2024), and the StAP-tutor (Roest et al., 2023). Other chatbots, such as the CS50 Duck (Liu et al., 2024, 2025) and the CodeTutor (Lyu, Wang, Chung, Sun, & Zhang, 2024) can provide context-aware feedback. Tutor Kai (Jacobs, Peters, et al., 2025) can generate comprehensive programming tasks, including problem descriptions, code skeletons, unit tests, and model solutions (Jacobs, Peters, et al., 2025) to help students practice and gain experience while also getting feedback (Jacobs, Kempf, & Kiesler, 2025). Generally speaking, all of these tools are designed to support novice programmers struggling with various challenges. Among them are cognitively complex tasks in introductory programming (e.g., problem understanding, designing and writing algorithms, debugging, and understanding error messages (Kiesler, 2024; Luxton-Reilly et al., 2018; Ebert & Ring, 2016; Spohrer & Soloway, 1986; Du Boulay, 1986)). Educators' high and unrealistic expectations (Luxton-Reilly, 2016; Luxton-Reilly et al., 2018; Whalley, Clear, & Lister, 2007; Kiesler, 2022), a low student-educator ratio (Petersen, Craig, Campbell, & Tafliovich, 2016), and an increasingly diverse student body in higher education (e.g., due to diverse learners and educational biographies) add to the list of novices' challenges.

Considering the potential of GenAI and the challenging nature of introductory programming education, it is crucial to keep advancing tutorial systems capable of providing the feedback computing students need and want. Despite the availability of certain tools for novice programmers (e.g., CodeAid, etc.), there is no one-size-fits-all solution and no consensus on how to design a chatbot with pedagogical guardrails yet. Prior research analyzed students' interactions with GenAI tools, such as ChatGPT (Scholl, Schiffner, & Kiesler, 2024). It revealed diverse usage patterns, e.g., superficial engagement, such as seeking quick solutions, but also extensive and iterative dialogues. The evaluation of the student perspective indicates their appreciation of GenAI's continuous availability and immediate responses (Scholl & Kiesler, 2024). At the same time, students expressed their concerns about over-reliance, inconsistent responses, and a lack of pedagogical guidance (Scholl & Kiesler, 2024). Motivated by these insights and in line with the lessons learned from the design of similar tools, we developed a *Supportive Chatbot for Resolving Introductory Programming Tasks* (SCRIPT) for novice learners of programmers. It is based on ChatGPT-4o-mini, and integrates task description and context into the system prompt. Computing students are offered predefined prompts for specific feedback types (Keuning, Jeuring, & Heeren, 2018; Narciss, 2006) based on related work (Lohr et al., 2025). In addition, students can enter free-form input, and receive step-by-step responses (Liffiton et al., 2024) without giving away (model) solutions.

In this research paper, the **goal** is to investigate students' interactions with SCRIPT in terms of requested feedback types. In addition, we evaluate to what extent the generated responses match students' requests and whether the output adheres to the system prompt constraints. The results **contribute** to increasing our understanding of GenAI-based chatbots for introductory programming education and how to design them to align with learners' needs. They particularly inform us of learners' needs in terms of AI-generated feedback for programming tasks. These findings thus have implications for tool developers and educators, but also computing students applying respective tools.

2. Related Work

In the past few years, numerous application contexts were evaluated to show the potential of GenAI for computing education, and introductory programming in particular. For example, GenAI and related tools (e.g., ChatGPT, Codex, Copilot, or Llama) have been shown to successfully solve introductory programming tasks and pass respective exams (Wermelinger, 2023; Geng et al., 2023; Kiesler & Schiffner, 2023a; Savelka et al., 2023). Other studies revealed they can enhance programming error messages (Leinonen et al., 2023), and effectively generate code explanations (MacNeil et al., 2023; Sarsa et al., 2022). Explanations by GenAI were found to be capable of analyzing student code and providing instruction on how to fix errors (Phung et al., 2023; Zhang et al., 2022). Even elaborate feedback types were identified in GenAI output in response to student code (Kiesler et al., 2024; Azaiz et al., 2024). In the following, we present relevant research on GenAI feedback, and customized GenAI tools that informed the development of our system (SCRIPT), and the research design.

2.1. Feedback Potential of GenAI

The generation of feedback is considered a huge potential of GenAI, especially with the rapid advancements of the underlying large language models. A study by Phung et al. (Phung et al., 2023) investigated the use of LLMs to fix syntax errors in Python programs. They developed a technique to receive high-precision feedback from Codex. By using qualitative methods, several other studies explored the feedback characteristics of GenAI in response to student solutions containing errors (Kiesler et al., 2024; Azaiz et al., 2024). For example, Kiesler et al. (Kiesler et al., 2024) identified the following feedback elements: stylistic advice, textual explanations of the cause of errors and their fix, examples, meta-cognitive and motivational elements. However, they recognized misleading information and uncertainty in the model's output, depending on the task. ChatGPT-3.5 also requested more information in a few cases. A qualitative evaluation of GPT4 Turbo's feedback showed notable improvements (Azaiz et al., 2024). The outputs were more structured, consistent, and always personalized (Azaiz et al., 2024).

Lohr et al. (Lohr et al., 2025) investigated whether they can generate specific feedback for introductory programming tasks using GenAI (i.e., ChatGPT). Following an iterative approach, they designed prompts to elicit specific feedback types, such as knowledge about mistakes, or knowledge on how to proceed (cf. (Narciss, 2008; Keuning et al., 2018)). They qualitatively evaluated the generated output with human intelligence and determined its feedback type to check if the prompts had elicited the expected feedback. As a result, they present prompts for the generation of different types of feedback suitable for introductory programming tasks and student submissions (Lohr et al., 2025). They also noted that misleading information occurred less frequently compared to related work (Kiesler et al., 2024) (Lohr et al., 2025).

2.2. Customized GenAI Tools for Programming Education

Due to their feedback potential, GenAI models are being integrated into educational tools and systems to offer novice-friendly explanations tailored to individual student errors. The dcc - -help tool, for example, integrates ChatGPT 3.5 into the Debugging C Compiler (DCC). It produces context-aware explanations in response to compiler errors in DCC (Taylor, Vassar, Renzella, & Pearce, 2024). A more recent conversational AI extension to the GenAI-enhanced C/C++ compiler DCC is DCC Sidekick (Renzella, Vassar, Lee Solano, & Taylor, 2025). It generates pedagogical programming error explanations without integrating student input, and by applying the Socratic method.

Another example of a programming assistant is CodeAid (Kazemitabaar et al., 2024). Its implementation avoids the generation of code solutions to support students' thinking and learning. CodeAid offers help with general questions, inline code explorations, questions from code, help to fix code, code explanations, and help in writing code. Yet, the developers did not distinguish or implement any well-known and context-specific feedback typology (cf. (Keuning et al., 2018)). Kazemitabaar et al. (2024) summarize four design considerations for AI coding assistants: (1) exploiting unique advantages of AI, (2) designing the AI querying interface, (3) balancing the directness of AI responses, and (4) supporting trust, transparency and control.

CodeHelp is another GenAI-powered tool designed to provide guardrails and on-demand programming assistance (Liffiton et al., 2024). Its features comprise a simple interface for students' help requests, a system prompt avoiding the generation of complete code solutions, and a sufficiency check to catch ambiguous or incomplete student queries. CodeHelp was evaluated by 52 students over a 12-week period. The results show students value the availability and help of the tool, and that it complements the efforts of the instructor (Liffiton et al., 2024).

Xiao et al. (2024) followed a somewhat different approach by investigating whether and how different hints support students' problem-solving and learning processes in the LLM Hint Factory (Xiao et al., 2024). The latter is a system providing four levels of hints, from natural language to concrete code assistance. Via a think-aloud study with 12 programming novices, they identified recommendations for feedback design. For example, hints about the next step or how to solve syntax issues should be concise and personalized to students' requests to meet their needs. Otherwise, students may switch to ChatGPT to receive full solutions according to the authors of the study (Xiao et al., 2024). Similarly, Roest et al. 2023 developed the StAP-tutor; a GenAI-based tutoring system for next-step hints (Roest et al., 2023). The system was designed by exploring various prompts and evaluating the feedback. The best output was generated for inputs with a problem description and the words “student” and “hint”. Another recommendation is to increase the temperature parameters of the used model. Through a student and expert evaluation, the feedback was evaluated as concise and personalized, but it still contained misleading information for OpenAI's GPT-3.5-turbo model. Thus, future work is required to evaluate more recent models (Roest et al., 2023).

There are some other recent applications and studies on the use of GenAI in computing. For example, Yeh et al. (2025) investigated an interactive approach to teach students how to write better prompts so they can eventually generate working code (Yeh et al., 2025). Bhowmick and Li (2025) experimented with LANTERN to teach students relational query processing as part of a database course (Bhowmick & Li, 2025). Similarly, Riazi and Rooshenas (2025) explored GenAI feedback to support students' conceptual design competencies (e.g., via entity-relationship diagrams) in a database systems course (Riazi & Rooshenas, 2025). Forden et al. (2025) developed an automated assessment tool trying to cultivate students' coding style and foster timely submissions (Forden, Schneider, Gebhard, Islam Molla, & Brylow, 2025). A last example is SENSAI, an AI-powered tutoring system for teaching cybersecurity (Nelson, Doupé, & Shoshitaishvili, 2025). It utilizes the learner's workspace, i.e., active terminals and edited files as input, highlighting the importance of context in the system prompt.

2.3. Research Gap

It is crucial to keep investigating newly emerging large language models (e.g., ChatGPT 4o, 4.5, and OpenAI o1, o3), which is due to the fast pace of this technology (Prather et al., 2023). This is particularly relevant as research data (e.g., for benchmarking) are usually not available (Kiesler & Schiffner, 2023b; Prather et al., 2023). Few studies utilize the same dataset for evaluating or benchmarking recent models. Moreover, the general limitations of GenAI models are well-known. Among them are inaccuracies, hallucinations, misleading information (especially for novice learners), data and privacy concerns, reproduction of bias and stereotypes, lack of accessibility, and their in-transparency by design (Gill et al., 2024; Prather et al., 2023; Zhai, 2022; Kiesler et al., 2024, 2025; Alshaigy & Grande, 2024). Hence, we need to critically reflect upon the use of GenAI, and how to provide pedagogical instruction and guardrails for students. The same applies to the development and design of tools based on GenAI models. A major challenge in this design is finding a balance between individual student support and avoiding the extensive generation of model solutions. This can be addressed by restricting the chatbot from generating any code at all (cf. (Liffiton et al., 2024)).

Our goal, however, is to enable more flexible interactions, where the chatbot can produce helpful code snippets without giving away complete solutions. We also postulate that students can benefit from individual feedback and help, and code snippets may be helpful for some of them. In addition, we want to provide exemplary prompts leading to specific types of feedback, e.g., indicating and explaining error(s), or the next steps to solve them, as successful prompting can be challenging for novices as well. Moreover, we need to integrate students into the discussion to qualitatively evaluate how students apply GenAI tools (customized or not). Therefore, a critical aspect of this work is whether AI-generated feedback addresses students' informational needs.

3. Methodology

In this study, we present SCRIPT, a chatbot based on ChatGPT-4o-mini. It is designed to support novice learners of programming seeking feedback. The evaluation of SCRIPT is guided by the following research questions:

RQ1 *How do students interact with SCRIPT in the context of introductory programming tasks?*

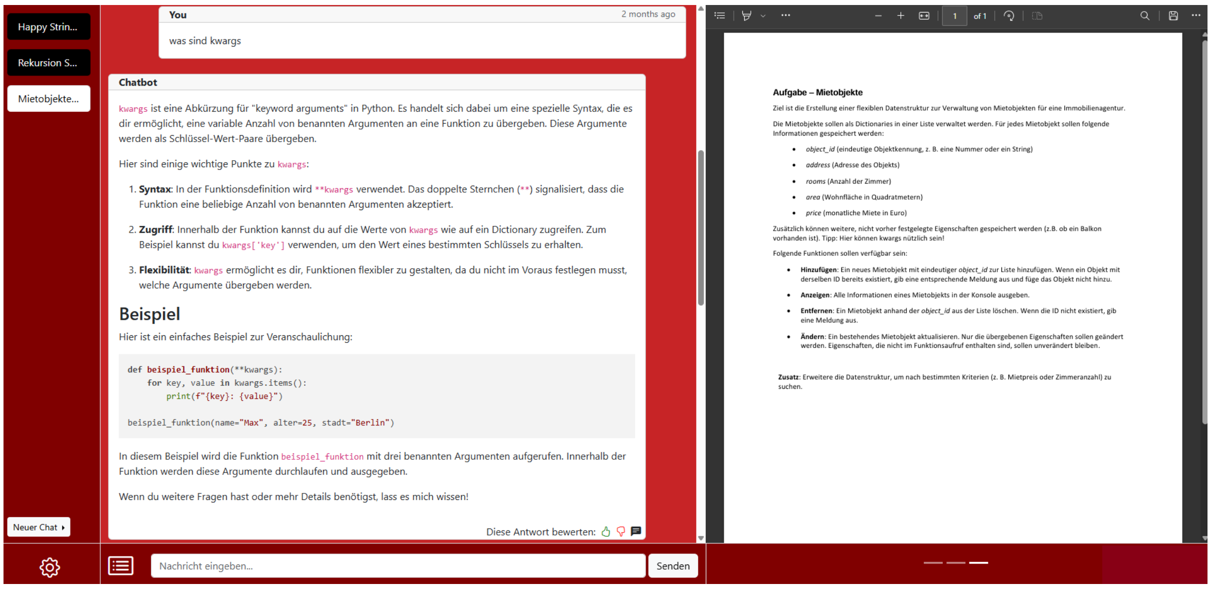
RQ2 *To what extent does the generated output match students' requests?*

RQ3 *To what extent do the generated outputs adhere to the system prompt constraints?*

To address these RQs, we utilize empirical data from students (n=136) enrolled in an introductory programming class at a large German university. Students were asked to solve programming exercises at home while having access to the SCRIPT chatbot. The chat protocols were automatically recorded, and students were asked to rate the GenAI-powered responses and leave comments (all voluntarily). In the following subsections, we introduce SCRIPT, the course context, selected tasks, and data analysis methods. Our research data (e.g., tasks, system prompt, interactions, etc.) are available in an online repository (Scholl & Kiesler, 2025a).

3.1. Introducing SCRIPT

SCRIPT (Scholl & Kiesler, 2025b) is implemented as a standalone web application, which is accessible via a link in the respective university's Moodle course (no additional sign-in needed). The setup enables anonymous data gathering (via anonymous IDs). The user interface (UI) is depicted in Figure 1. It is similar to the structure of ChatGPT's interface. Each chat session is displayed at the very left of the UI. It is task-specific, meaning a conversation is dedicated to solving one programming problem. Students can initiate multiple conversations, including revisiting the same task. The chatbot receives both the task description (displayed on the right) and a reference solution as context (i.e., as part of the system prompt (Scholl & Kiesler, 2025a)), eliminating the need for students to manually copy and paste these elements into the chat, which is supposed to ease its use (Kazemitabaar et al., 2024). Students can rate each GenAI response with a thumbs-up or down feedback, and via textual comments (all optional). The backend architecture relies on a Node.js environment with an Express/Socket.io server. The data management is handled via an SQL database (MariaDB). Student interactions and their feedback (thumbs-up/down ratings, and open input) are logged.



You 2 months ago

was sind kwargs

Chatbot

`kwargs` ist eine Abkürzung für "keyword arguments" in Python. Es handelt sich dabei um eine spezielle Syntax, die es dir ermöglicht, eine variable Anzahl von benannten Argumenten an eine Funktion zu übergeben. Diese Argumente werden als Schlüssel-Wert-Paare übergeben.

Hier sind einige wichtige Punkte zu `kwargs`:

1. **Syntax**: In der Funktionsdefinition wird `**kwargs` verwendet. Das doppelte Sternchen (`**`) signalisiert, dass die Funktion eine beliebige Anzahl von benannten Argumenten akzeptiert.
2. **Zugriff**: Innerhalb der Funktion kannst du auf die Werte von `kwargs` wie auf ein Dictionary zugreifen. Zum Beispiel kannst du `kwargs['key']` verwenden, um den Wert eines bestimmten Schlüssels zu erhalten.
3. **Flexibilität**: `kwargs` ermöglicht es dir, Funktionen flexibler zu gestalten, da du nicht im Voraus festlegen musst, welche Argumente übergeben werden.

Beispiel

Hier ist ein einfaches Beispiel zur Veranschaulichung:

```
def beispiel_funktion(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {value}")

beispiel_funktion(name="Max", alter=25, stadt="Berlin")
```

In diesem Beispiel wird die Funktion `beispiel_funktion` mit drei benannten Argumenten aufgerufen. Innerhalb der Funktion werden diese Argumente durchlaufen und ausgegeben.

Wenn du weitere Fragen hast oder mehr Details benötigst, lass es mich wissen!

Diese Antwort bewerten:

Aufgabe – Mietobjekte

Ziel ist die Erstellung einer flexiblen Datenstruktur zur Verwaltung von Mietobjekten für eine Immobilienagentur.

Die Mietobjekte sollen als Dictionaries in einer Liste verwaltet werden. Für jedes Mietobjekt sollen folgende Informationen gespeichert werden:

- *object_id* (eindeutige Objektkennung, z. B. eine Nummer oder ein String)
- *address* (Adresse des Objekts)
- *rooms* (Anzahl der Zimmer)
- *area* (Wohnfläche in Quadratmetern)
- *price* (monatliche Miete in Euro)

Zusätzlich können weitere, nicht vorher festgelegte Eigenschaften gespeichert werden (z.B. ob ein Balkon vorhanden ist). Tipp: Hier können kwargs nützlich sein!

Folgende Funktionen sollen verfügbar sein:

- **Hinzufügen**: Ein neues Mietobjekt mit eindeutiger *object_id* zur Liste hinzufügen. Wenn ein Objekt mit derselben ID bereits existiert, gib eine entsprechende Meldung aus und füge das Objekt nicht hinzu.
- **Anzeigen**: Alle Informationen eines Mietobjekts in der Konsole ausgeben.
- **Entfernen**: Ein Mietobjekt anhand der *object_id* aus der Liste löschen. Wenn die ID nicht existiert, gib eine Meldung aus.
- **Ändern**: Ein bestehendes Mietobjekt aktualisieren. Nur die übergebenen Eigenschaften sollen geändert werden. Eigenschaften, die nicht im Funktionsaufruf enthalten sind, sollen unverändert bleiben.

Zusatz: Erweitere die Datenstruktur, um nach bestimmten Kriterien (z. B. Mietpreis oder Zimmeranzahl) zu suchen.

Figure 1 – Overview of the SCRIPT User Interface

SCRIPT provides two distinct prompting options, which are designed to exploit the unique advantages of GenAI while balancing guidance and flexibility (Kazemitabaar et al., 2024; Yeh et al., 2025). *Closed prompts* are predefined based on recent research on AI-generated feedback (Lohr et al., 2025) and an established feedback typology for programming contexts (Keuning et al., 2018; Narciss, 2006). These closed prompts guide students through problem-solving steps and issues, including understanding task constraints (KTC), identifying required programming concepts (KC), recognizing mistakes (KM), determining how to proceed (KH), assessing performance levels (KP), and evaluating code correctness relative to a reference solution (KR). These closed prompts are available for students next to the input field. They are supposed to encourage students' structured engagement with the chatbot, and reduce the challenges of starting with the problem-solving process, or formulating a prompt (cf. (Scholl & Kiesler, 2024; Scholl et al., 2024)).

Open prompts are common when using a chatbot, and are comparable to those expected by CodeHelp (Liffiton et al., 2024). They denote students' free-form textual input. When students enter text, SCRIPT prevents the generation of complete code as output. Instead, it fosters step-by-step problem-solving by identifying errors in student code without directly correcting them. Thus, the tool provides stepwise hints and no complete solutions in the form of code. It also offers examples (unrelated to the specific task) and templates (incomplete code structures with only comments in key sections). SCRIPT is available at any time, so students can work at their own pace, with all tasks being accessible. This is supposed to support mastery learning where learners can repeatedly engage with a task until achieving proficiency (Szabo et al., 2025; Keller, 1968). The dual-mode interaction structure with closed and open prompts allows students to choose between guided problem-solving and a more exploratory learning approach. The system prompts for both modes are available in the online repository (Scholl & Kiesler, 2025a).

3.2. Course Context and Task Selection

SCRIPT was designed to support students in an introductory programming course for first-year computing students ($N = 666$) at Goethe University Frankfurt (Germany). The present study was conducted in the winter term 2024/25. The majority of students were enrolled in the bachelor's degree program in Computer Science (CS). Only some were enrolled in other disciplines or chose CS as a minor. Prior programming experience was not required to participate. A Moodle course provided access to learning materials, including SCRIPT. The course structure included a weekly two-hour lecture for all students. Tutorial sessions (20 students each) accompanied the course. Tutorial sessions typically last two hours every week. A key component of the tutorials is the weekly or biweekly exercises, so students can gain hands-on programming experience. Students work individually on tasks and earn exercise points for their submissions. These contribute to the final exam score.

For this study, we designed three programming tasks for students to be completed using SCRIPT. The tasks were supposed to be solved within one week, starting on January 13, 2025. Students had the opportunity to engage with one or more tasks. Students were not instructed on how to use SCRIPT. Participation was voluntary, but students could earn four exercise points as an incentive. Furthermore, the purpose of the study and its procedures were introduced to students during the lecture preceding their tutorial sessions. Table 1 summarizes the tasks used in this study and the concepts they address. The given tasks were aligned with the course curriculum and the student's assumed/expected level of expertise. The course's facilitator also ensured that the tasks were relevant and adequate. The full task descriptions are available along with the other research data (Scholl & Kiesler, 2025a).

Task	Description	Concepts
Happy Strings	Compute the number of "happy" strings within all sub-strings of a given string of digits. A string is considered "happy" if the digits can be rearranged to repeat twice. The following steps are required: (a) Test for "happy" property; (b) iterate and test all substrings.	recursion, functions, lists, conditionals, string manipulation
Recursion Snippets	Determine recursion type, return value and number of function calls of 4 functions: (a) sum of digits; (b) list reversal; (c) multiplication (d) Ackermann function.	
Rental Properties	Implement a data structure to manage rental properties using dictionaries in a list. Provide functions to add, display, remove, and update properties based on unique IDs. Extend functionality to allow searching by criteria.	functions, lists, dictionaries, keyword parameters, data structures

Table 1 – Selected tasks with short description and required programming concepts.

3.3. Data Analysis

To examine students' interactions with SCRIPT, we collected and analyzed chat sessions from 136 students with their input and SCRIPT's output. The chat logs were stored as individual sessions (without any personal data or identifier). They also include students' ratings of SCRIPTs' responses, and textual feedback (if any). The chat logs constitute the data basis for RQ1, RQ2, and RQ3.

For RQ1 and RQ2, we particularly focused on the student prompts and SCRIPTs responses to them. All of these questions and answers were analyzed and categorized regarding the requested and generated feedback type(s): knowledge of result (KR), knowledge of correct result (KCR), knowledge of performance (KP), knowledge about task constraints (KTC), knowledge about concepts (KC), knowledge about mistakes (KM), knowledge on how to proceed (KH), and knowledge about meta-cognition (KMC) (Keuning et al., 2018). The analysis followed a structured coding approach. We started with approximately 15% of the chat sessions to confirm the adequacy of the deductive coding scheme based on the literature (Keuning et al., 2018). It applied to almost all feedback requests and responses. However, some additional requests and responses were noticed and coded inductively based on the material. For example, students' prompts were categorized as: technical (TEC), social interaction (SoI), answer to GenAI question (ANS), clarification question (WHAT), off-topic (OFT), new task request (TR), incomprehensive (IN), and prompt injection (HACK). Similarly, SCRIPTs' responses were coded with these additional categories: marked as offensive (OFF), denied request (DENY), social (SoI), technical

(TEC), off-topic (OFT), technical error (TE), and new task (TR). (Definitions and anchor examples of the additional categories are available in the data publication (Scholl & Kiesler, 2025a).) As part of this rule-guided coding process, multiple codes (a maximum of 3) were applied to a coding unit (i.e., a student input and a generated response each was considered a single coding unit). A student's entire chat session was considered a context unit, in case of uncertainties (e.g., regarding the intentions of a student trying to elicit feedback or have a nice chat). After the initial round, deductive and inductive categories were applied to the remaining material. All of the collected chat data were coded twice by the same coder (first author of this work) to ensure internal consistency and reliability. The coder had prior experience in qualitative coding.

To answer RQ1, the student requests and chatbot responses were aggregated and processed into a flowchart to represent the interactions and their frequencies. Regarding RQ2, we compared the feedback type request of every student question with the feedback category/categories evident in SCRIPTs' responses (both in an aggregated and pair-wise form). We present the number of matches, over-matches (i.e., a match of requested and generated feedback types, plus additional feedback categories in the response), partial matches, or mismatches.

The last research question (RQ3) had the goal of evaluating the quality of SCRIPTs' outputs in terms of correctness, and step-wise hints. We addressed those aspects through the following indicators (cf. (Azaiz et al., 2024; Azaiz, Kiesler, Strickroth, & Zhang, 2025; Roest et al., 2023)): (1) *Number of problem-solving steps* SCRIPT provides in a response. Per system prompt, a single step per response should be provided. (2) *Solution* provided to the task (partial or complete), whereas no full solution should be generated. (3) *Code examples*; SCRIPT is instructed to provide only simple examples. (4) *Code templates*; SCRIPT may provide code templates to students, consisting of structural frameworks for a function, loop, or conditional headers. (5) *Code Corrections*; SCRIPT should correct students' code by pointing out the mistake and providing hints. Hence, we evaluate whether SCRIPT provides corrected code. (6) *Correctness of Response*; we determine if a response from SCRIPT is correct, partially correct, or incorrect.

4. Results

In total, 136 students engaged with SCRIPT, generating 241 chat sessions. Across these interactions, students submitted 1,409 prompts, and SCRIPT generated 1,409 responses. The distribution of prompts per student varied, though, with a mean of 10.36 (SD = 10.86) and a median of 7. Among the student prompts were 207 predefined "closed" prompts, requesting the following feedback types: KC (54), KTC (48), KR (49), KM (26), KP (17), and KH (13).

4.1. Students' Interactions with SCRIPT (RQ1)

To answer RQ1, we analyzed the feedback requests and generated responses w.r.t. their feedback types. The resulting feedback (and other) categories of inputs and outputs were aggregated in a flowchart, as displayed in Figure 2. It illustrates common interaction patterns between students and SCRIPT. Nodes indicate the feedback types of students' prompts, and edges represent the feedback generated by SCRIPT. The flowchart includes loops, such as for KH, which illustrate repeated requests for the same feedback type. Next to the feedback type, we always provide the number of occurrences. Figure 2 only represents those interaction patterns observed at least ten times, and thus were more common. It is important to note that the figure represents the results in an aggregated form and does not display individual student paths. A more detailed flowchart and the raw data are available online (Scholl & Kiesler, 2025a).

In the following, we describe the most frequent requests and responses as depicted in Figure 2. After a default greeting by SCRIPT, students began their session by asking questions about the task constraints (KTC) in 71 cases. This was often followed by questions (19) regarding the necessary programming concepts (KC). Next, students (46) engaged in multiple follow-up inquiries, requesting further explanations or clarifications of concepts. Then they submitted their (partial) solution and asked SCRIPT to evaluate its correctness (KR, 14). This process often occurred iteratively, with students refining their code based on SCRIPT's feedback (28). Instead of simply providing a binary correct/incorrect response, SCRIPT typically offered reasoning about the student's performance (KP, 16), clarifying key concepts (KC, 12), or suggesting how to approach corrections (KH, 16). However, when students used predefined *closed prompts*, SCRIPT adhered to a stricter response format, delivering only KR feedback (i.e., correct/incorrect). This often led students to seek additional clarification about their mistakes through *open prompts*. Another interaction pattern was students requesting the correct solution (KCR) at the beginning of the session. In 18 of the 23 cases, SCRIPT explicitly denied these solution requests.

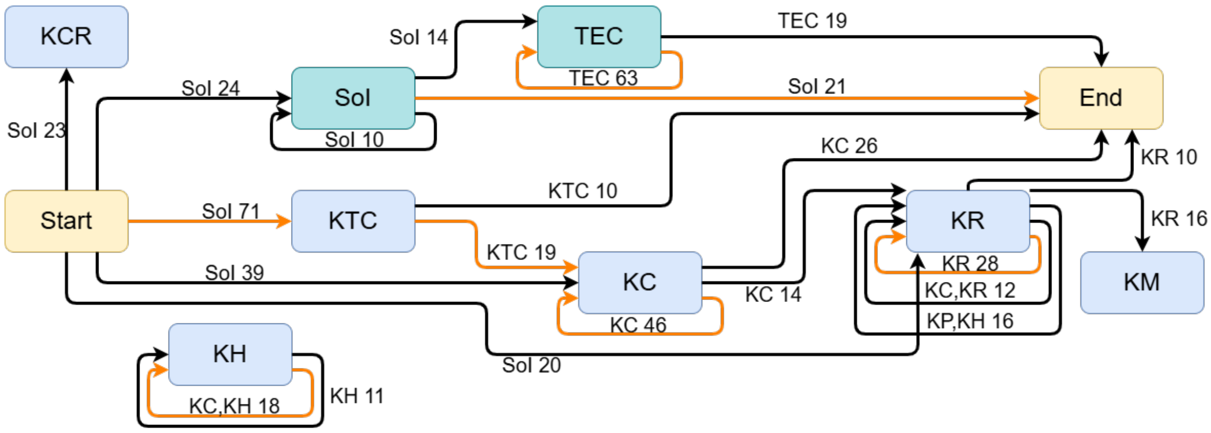


Figure 2 – Flowchart of students' interactions. Nodes mark students' feedback requests, edges represent SCRIPTs' feedback categories. Only interactions that occurred at least 10 times are represented.

In 24 chat sessions, students initiated conversations with social interactions, sometimes (10) followed by additional social exchanges. In 14 sessions, this led students to ask technical questions about SCRIPT before engaging with the task. Some sessions (19) ended at this step without students attempting to solve the problem. Regardless of how a student started or concluded their chat, students asked how to proceed (KH) at different stages of the problem-solving process (see Figure 2). KH loops have occurred at least 10 times (KH, 11; KC, KH, 18). Thus, the number of outgoing requests from KH to other feedback types was less than 10, which is why the KH node is isolated in the aggregated flowchart.

SCRIPT typically responded with step-by-step explanations, often adding conceptual knowledge (e.g., KC, KH 18) to guide students through the task and next steps. This iterative exchange generally continued until the student reached a solution. However, a few students relied solely on requesting the next steps without adding their code as input. In such cases, SCRIPT constructed the solution incrementally, guiding the student through each step of the process.

4.2. Matches of Student Requests and SCRIPT's Responses (RQ2)

To understand the extent to which the generated output matched students' requests, we first of all aggregated all requested and provided feedback types in Table 2a. It shows the total number of occurrences of requested and generated feedback types across all student prompts and responses. For most feedback categories, the numbers align closely, indicating that SCRIPT generally generated what students requested. For some requests, SCRIPT supplemented its responses with additional conceptual explanations (KC, 111) and guidance on how to proceed (KH, 106), particularly when students asked for feedback for their code (i.e., had requested KM (27), or KR (37)). For this reason, the number of feedback types in the responses exceeds the number of requests for KC, and KH (see Table 2a).

We also evaluated to what extent the student feedback requests and the immediate responses from SCRIPT were aligned (i.e., in the question-response pairs). It should be noted though that not all student inputs requested feedback, which is why we only evaluated 891 question-response-pairs. In 47% of cases (417 out of 891), the chatbot's feedback directly matched what students requested. We also observed so-called "over-matching". That is, SCRIPT provided additional feedback types, and not

Feedback Type	No. Requests	No. Responses
KTC	158	147
KC	295	424
KH	159	352
KM	84	81
KP	36	71
KR	209	149
KCR	100	120
KMC	6	6

(a) Feedback Types

Additional Cat.	No. in Prompts	No. in Responses
TEC	165	152
SoI	101	106
ANS	141	-
WHAT	71	-
OFT	48	45
TR	15	15
IN	14	-
HACK	26	-
DENY	-	42
OFF	-	9
TE	-	58

(b) Additional Categories

Table 2 – Total number of requested and generated (a) feedback types and (b) additional categories (both not pair-wise); prompts and responses may contain multiple types each.

just those that were explicitly requested. Considering these, the total alignment increased to 75%, i.e., an additional 255 out of the 891 pairs were over-matching. Overmatching primarily occurred for KC and KH categories, where SCRIPT offered extra explanations or next-step guidance beyond what was explicitly requested. In 22% of question-response pairs (195 out of 891), the response did not match the intended feedback type (mismatch). Notably, 25% of these mismatches (49 out of 195) involved students requesting direct solutions (KCR), which SCRIPT was designed to reject. This indicates that while mismatches occurred, a quarter resulted from SCRIPT correctly adhering to its constraints – rather than failing to provide the desired feedback. 58 technical errors (TE) occurred when SCRIPT refused to answer requests for KTC and KC feedback, due to missing student code (KTC and KC did not require student code to be generated, though).

Students themselves provided 151 thumbs-ratings (130 up, 21 down) and 29 short written comments. Positive remarks highlighted SCRIPT's clear explanations and helpful guidance ("Providing a template without giving everything away is helpful"). Critical comments mostly addressed incorrect evaluations, overly confident, or brief responses ("Here, the AI was confidently wrong.").

4.3. SCRIPT's Adherence to System Prompt Constraints (RQ3)

To evaluate SCRIPT's adherence to the system prompt constraints, we evaluated (1) the number of problem-solving steps, the generation of (2) solutions, (3) code examples, (4) templates, (5) code corrections, and (6) the correctness of responses of all 1409 generated responses (see Table 3). After restricting the responses from SCRIPT to problem-related answers (removing the categories TEC, SoI, DENY, OFF, OFT, TE, and TR), 1016 responses remained for analysis.

(1) In 51%, SCRIPT adhered to the constraint of providing only one step at a time in the problem-solving process. 21% of the responses comprised an overview containing multiple steps but included a clear starting point or next step (see Table 3). (2) Across all tasks, students made 100 KCR requests. SCRIPT provided solutions in 42 cases. These were usually granted after a stepwise generation of problem-solving steps. The success rate of solution requests varied across tasks, with "Recursion Snippets" having the highest proportion of fulfilled requests (71%), followed by "Happy Strings" (38%) and "Rental Properties" (25%). (3) Simple code examples were generated 104 times, and complex examples were observed in 3 responses. (4) Regarding template generation, SCRIPT correctly provided 167 templates in the intended format, offering general code structures without implementations. However, in 95 responses, it eventually provided completed templates, after progressively building up the solution and providing several partial templates. (5) SCRIPT corrected the code of students 34 times and displayed it, despite the constraint not to do so. (6) Overall, the correctness of SCRIPT's responses was high, with 81% of answers being fully correct. 14% of responses were at least partially correct, leaving only 5% of answers classified as completely incorrect.

(1) Number of solving steps in response			(6) Correctness of response		
Single Step	514	51%	Correct	822	81%
Multiple Steps	289	28%	Partially Correct	141	14%
Multiple Steps, Explicit Next-Step	213	21%	Incorrect	53	5%

(2) Solution given		(3) Given Examples		(4) Code templates		(5) Code corrections	
Partial	102	Simple	104	Provided	167	Corrected	34
Complete	129	Complex	3	Completed	95		

Table 3 – Indicators for SCRIPT's adherence to the system prompt constraints (for 1016 responses).

5. Discussion

Our findings revealed interaction patterns of introductory programming students using SCRIPT. When using the chatbot, students seemed to follow a certain (problem-solving) sequence, that has not been revealed in previous research: understanding the task constraints (KTC), identifying relevant programming concepts (KC), formulating a solution approach (KH), debugging errors (KM), verifying results (KP, KR), and ultimately comparing with the correct solution (KCR). While most interactions focused on problem-solving, some students also engaged in technical inquiries (TEC) and social exchanges, highlighting a broader spectrum of engagement beyond the mere task. Notably, KH inquiries and follow-up questions from the chatbot led to more student interactions (i.e., follow-up questions), with SCRIPT guiding students step by step.

Related to RQ2, the chatbot's responses aligned well with students' needs in most cases, particularly in clarifying task constraints (KTC), which were addressed with good accuracy. Concept explanations (KC) were generally good but occasionally too complex or lacking specific details. This suggests future improvements of the chatbot in tailoring explanations more precisely to the task. Performance feedback (KP) showed a high degree of variation and seemed random. KR feedback in *closed prompts* was mostly binary, so students usually continued with additional *open prompts* to elicit KM feedback. This may indicate that students perceived it as insufficient or had different expectations towards the generated response. Solution requests (KCR) resulted in the highest mismatch rate. This was expected as the chatbot was designed to reject such queries. Respective students tried to circumvent this via step-wise questions and gradually reconstructing the solution.

While SCRIPT generally adhered to system prompt constraints, some inconsistencies were observed. Especially for the *Recursion Snippets* task, we found that solutions were often provided too early. Code examples were consistently brief and simple. Code templates initially adhered to the guidelines but tended to be complete after several student inquiries. Direct solution requests were blocked as expected. Yet, in step-wise interactions, partial solutions emerged sooner or later, depending on the task. From a pedagogical perspective, this may not necessarily be a problem – as long as students actively process the feedback. In general, it seems recommendable to balance direct AI responses (Kazemitabaar et al., 2024), and prevent the generation of complete code solutions (Liffiton et al., 2024). Importantly, SCRIPT demonstrated robustness against prompt injection attempts, successfully resisting nearly all manipulation efforts, except for one.

An additional observation concerns the chatbot's responses to *closed prompts* (based on (Lohr et al., 2025)), which were more formal, often writing in the third-person. In contrast, students' *open* inputs exhibited a more conversational tone. We perceived *closed* and *open prompts* as two distinct conversational modes, with some abrupt transitions between the two. This observation may be attributed to the varying perspectives SCRIPT is required to adopt: *closed prompts* request structured, restricted guidance, while *open prompts* may shift the chatbot's role to a personal assistant – depending on the student input.

In general, the *closed prompts* from prior work seemed useful (Lohr et al., 2025), which is reflected in students' use of them (207 times) and the general match between student requests and generated responses. It may also be helpful for students to get started or formulate a specific prompt, e.g., due to language barriers, anxiety, or insecurity about how to use technical terms. At the same time, it seems advisable to offer users multiple options, i.e., both *closed* and *open prompts*, thereby bridging guidance and flexibility, as suggested in related work (Yeh et al., 2025).

Finally, we noted that the feedback typology from prior work (Keuning et al., 2018) was sufficient to describe the requested and generated feedback items. This is worth discussing, as it was constructed based on the existing learning environment at the time, not chatbots and/or GenAI. However, due to the new presentation (i.e., modality, adaptation (Narciss, 2006)) of the feedback via the chatbot, we identified additional categories, such as social interactions (SoI), and many others. As GenAI and related tools advance rapidly, we expect to see new feedback categories, or changed presentation modes, etc., in the near future.

5.1. Limitations

Some of the study's limitations should be noted. For example, students knew their interactions were being analyzed, potentially influencing their behavior (Roethlisberger & Dickson, 1939). SCRIPT was also used in an unsupervised setting, allowing students to engage freely. Few students experimented with the tool, using social or technical prompts without attempting to solve the task, which may have affected interaction patterns. The study also relies on a researcher-driven analysis. Thus, students were not explicitly asked to categorize their feedback types. They might have had slightly other intentions or informational needs than those resembled in their prompts. Yet, we tried to mitigate this limitation by allowing student feedback (via thumbs, and open input field) to each generated response. Finally, this study was conducted in a single university course in one country, limiting the generalizability of the findings. While the number of participants and responses strengthens its representativeness (Boddy, 2016), results may not be transferable to different curricula or institutions.

6. Conclusions and Future Work

In this study, we investigated students' interactions with SCRIPT, a GenAI-based chatbot developed to support problem-solving in introductory programming education. The evaluation involved 136 students, who solved dedicated programming tasks using SCRIPT in an unsupervised, self-paced setting. Chat sessions were analyzed regarding students' interactions and feedback requests, prompt-response alignment, and the system's adherence to constraints. Our analysis showed that students seemed to follow a certain sequence of feedback requests: KTC, KC, KH, KM, KP and KR, and, finally, KCR. Overall, SCRIPT provided correct responses in alignment with students' requests in 75% of its responses. Moreover, the system prompt was suitable for guiding students through step-by-step problem-solving without revealing full solutions. The chatbot remained robust and followed the instructional design.

The study's findings can inform the design of better prompts and scaffolded feedback sequences to help students work more independently with such tools. Future work should explore the use of GenAI tools with a broader range of task types and learning contexts, and continuously improve these early tools and prompts. In particular, closing the gap between *open* and *closed prompts* remains a key challenge, as it requires balancing structure and flexibility while managing the shift between the educator's role of restricting responses and supporting student exploration. This will be one of the next steps in advancing SCRIPT, to make the conversation flow more naturally. Moreover, we will use the students' feedback to improve the technical implementation, User Interface, and interaction. Continuing this work will help support novice programmers seeking feedback by SCRIPT. We also encourage other CS researchers, educators, and tool developers to keep exploring educational tools so we can leverage the potential of GenAI for good.

7. References

Alshaigy, B., & Grande, V. (2024). Forgotten again: Addressing accessibility challenges of generative ai tools for people with disabilities. In *Adjunct proceedings of the 2024 nordic conference on human-computer interaction.* New York, NY, USA: Association for Computing Machinery. doi: 10.1145/3677045.3685493

Azaiz, I., Kiesler, N., & Strickroth, S. (2024). *Feedback-generation for programming exercises with gpt-4.* New York, NY, USA: Association for Computing Machinery. doi: 10.1145/3649217.3653594

Azaiz, I., Kiesler, N., Strickroth, S., & Zhang, A. (2025). *Open, small, rigmarole – evaluating llama 3.2 3b's feedback for programming exercises.* (accepted to the International Journal of Engineering Pedagogy (iJEP; eISSN: 2192-4880)) doi: 10.48550/arXiv.2504.01054

Bengtsson, D., & Kaliff, A. (2023). *Assessment accuracy of a large language model on programming assignments.* Retrieved from `https://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-331000`

Bhowmick, S., & Li, H. (2025). Experience report on using lantern in teaching relational query processing. In *Proceedings of the 56th acm technical symposium on computer science education v. 1* (p. 123–129). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/3641554.3701812

Boddy, C. R. (2016). Sample size for qualitative research. *Qualitative market research: An international journal*, *19*(4), 426–432.

Du Boulay, B. (1986). Some difficulties of learning to program. *Journal of Educational Computing Research*, *2*(1), 57–73. doi: 10.2190/3LFX-9RRF-67T8-UVK9

Ebert, M., & Ring, M. (2016). A presentation framework for programming in programing lectures. In *Proc. educon* (pp. 369–374).

Forden, J., Schneider, M., Gebhard, A., Islam Molla, M. T., & Brylow, D. (2025). Unlocking student potential with ta-bot: Timely submissions and improved code style. In *Proceedings of the 56th acm technical symposium on computer science education v. 1* (p. 346–352). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/3641554.3701955

Geng, C., Zhang, Y., Pientka, B., & Si, X. (2023). *Can ChatGPT Pass An Introductory Level Functional Language Programming Course?*

Gill, S. S., Xu, M., Patros, P., Wu, H., Kaur, R., Kaur, K., . . . Buyya, R. (2024). Transformative effects of chatgpt on modern education: Emerging era of ai chatbots. *Internet of Things and Cyber-Physical Systems*, *4*, 19-23. doi: 10.1016/j.iotcps.2023.06.002

Jacobs, S., & Jaschke, S. (2024, May). Evaluating the Application of Large Language Models to Generate Feedback in Programming Education. In *2024 IEEE Global Engineering Education Conference (EDUCON)* (pp. 1–5). New York: IEEE. doi: 10.1109/EDUCON60312.2024.10578838

Jacobs, S., Kempf, M., & Kiesler, N. (2025). That's not the feedback i need! – student engagement with genai feedback in the tutor kai.. Retrieved from `https://arxiv.org/abs/2506.20433`

Jacobs, S., Peters, H., Jaschke, S., & Kiesler, N. (2025). *Unlimited practice opportunities: Automated generation of comprehensive, personalized programming tasks.* (accepted at ITiCSE 2025, https://doi.org/10.1145/3724363.3729089) doi: 10.48550/arXiv.2503.11704

Kazemitabaar, M., Ye, R., Wang, X., Henley, A. Z., Denny, P., Craig, M., & Grossman, T. (2024). Codeaid: Evaluating a classroom deployment of an llm-based programming assistant that balances student and educator needs. In *Proceedings of the chi conference on human factors in computing systems.* New York, USA: ACM. doi: 10.1145/3613904.3642773

Keller, F. S. (1968). Good-bye, teacher... *Journal of applied behavior analysis*, *1*(1), 79.

Keuning, H., Jeuring, J., & Heeren, B. (2018, 9). A systematic literature review of automated feedback generation for programming exercises. *ACM Trans. Comput. Educ.*, *19*(1). doi: 10.1145/3231711

Kiesler, N. (2022). *Kompetenzförderung in der programmierausbildung durch modellierung von kompetenzen und informativem feedback* (Dissertation). Johann Wolfgang Goethe-Universität, Frankfurt am Main. (Fachbereich Informatik und Mathematik)

Kiesler, N. (2024). *Modeling programming competency: A qualitative analysis.* Cham: Springer

International Publishing. doi: 10.1007/978-3-031-47148-3

Kiesler, N., Lohr, D., & Keuning, H. (2024). Exploring the potential of large language models to generate formative programming feedback. In *2023 ieee frontiers in education conference (fie)* (p. 1-5). doi: 10.1109/FIE58773.2023.10343457

Kiesler, N., & Schiffner, D. (2023a). *Large language models in introductory programming education: Chatgpt's performance and implications for assessments.* doi: 10.48550/arXiv.2308.08572

Kiesler, N., & Schiffner, D. (2023b). Why We Need Open Data in Computer Science Education Research. In *Proceedings of the 2023 conference on innovation and technology in computer science education v. 1* (p. 348–353). New York: ACM. doi: 10.1145/3587102.3588860

Kiesler, N., Smith, J., Leinonen, J., Fox, A., MacNeil, S., & Ihantola, P. (2025). *The role of generative ai in software student collaboraition.* New York, NY, USA: Association for Computing Machinery. doi: 10.1145/3724363.3729040

Leinonen, J., Hellas, A., Sarsa, S., Reeves, B., Denny, P., Prather, J., & Becker, B. A. (2023, March). Using large language models to enhance programming error messages. In *Proc. sigcse.* ACM. doi: 10.1145/3545945.3569770

Liffiton, M., Sheese, B. E., Savelka, J., & Denny, P. (2024). Codehelp: Using large language models with guardrails for scalable support in programming classes. In *Proceedings of the 23rd koli calling international conference on computing education research.* New York: ACM. doi: 10.1145/3631802.3631830

Liu, R., Zenke, C., Liu, C., Holmes, A., Thornton, P., & Malan, D. J. (2024). Teaching cs50 with ai: Leveraging generative artificial intelligence in computer science education. In *Proceedings of the 55th acm technical symposium on computer science education v. 1* (p. 750–756). doi: 10.1145/3626252.3630938

Liu, R., Zhao, J., Xu, B., Perez, C., Zhukovets, Y., & Malan, D. J. (2025). Improving ai in cs50: Leveraging human feedback for better learning. In *Proceedings of the 56th acm technical symposium on computer science education v. 1* (p. 715–721). New York, NY, USA: ACM. doi: 10.1145/3641554.3701945

Lohr, D., Keuning, H., & Kiesler, N. (2025). You're (Not) My Type – Can LLMs Generate Feedback of Specific Types for Introductory Programming Tasks? *Journal of Computer Assisted Learning*. doi: 10.1111/jcal.13107

Luxton-Reilly, A. (2016). Learning to Program is Easy. In *Proc. ITiCSE* (pp. 284–289). doi: 10.1145/2899415.2899432

Luxton-Reilly, A., Simon, Albluwi, I., Becker, B. A., Giannakos, M., Kumar, A. N., ... Szabo, C. (2018). Introductory Programming: A Systematic Literature Review. In *Proc. ITiCSE* (pp. 55–106). New York: ACM. doi: 10.1145/3293881.3295779

Lyu, W., Wang, Y., Chung, T. R., Sun, Y., & Zhang, Y. (2024). Evaluating the effectiveness of llms in introductory computer science education: A semester-long field study. *arXiv:2404.13414*.

MacNeil, S., Tran, A., Hellas, A., Kim, J., Sarsa, S., Denny, P., ... Leinonen, J. (2023). Experiences from Using Code Explanations Generated by Large Language Models in a Web Software Development E-Book. In *Proc. sigcse ts* (p. 931–937). doi: 10.1145/3545945.3569785

Narciss, S. (2006). *Informatives tutorielles feedback: Entwicklungs- und evaluationsprinzipien auf der basis instruktionspsychologischer erkenntnisse*. Münster: Waxmann Verlag.

Narciss, S. (2008). Feedback strategies for interactive learning tasks. *Handbook of research on educational communications and technology*, *3*, 125–144.

Nelson, C., Doupé, A., & Shoshitaishvili, Y. (2025). Sensai: Large language models as applied cybersecurity tutors. In *Proceedings of the 56th acm technical symposium on computer science education v. 1* (p. 833–839). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/3641554.3701801

Petersen, A., Craig, M., Campbell, J., & Tafliovich, A. (2016). Revisiting why students drop cs1. In *Proc. koli calling* (pp. 71–80). doi: 10.1145/2999541.2999552

Phung, T., Cambronero, J., Gulwani, S., Kohn, T., Majumdar, R., Singla, A., & Soares, G. (2023). *Gen-*

erating High-Precision Feedback for Programming Syntax Errors using Large Language Models.

Prather, J., Denny, P., Leinonen, J., Becker, B. A., Albluwi, I., Craig, M., … Savelka, J. (2023). The robots are here: Navigating the generative ai revolution in computing education. In *Proceedings of the 2023 working group reports on innovation and technology in computer science education* (p. 108–159). New York: ACM. doi: 10.1145/3623762.3633499

Prather, J., Leinonen, J., Kiesler, N., Gorson Benario, J., Lau, S., MacNeil, S., … Zingaro, D. (2025). Beyond the hype: A comprehensive review of current trends in generative ai research, teaching practices, and tools. In *2024 working group reports on innovation and technology in computer science education* (p. 300–338). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/3689187.3709614

Renzella, J., Vassar, A., Lee Solano, L., & Taylor, A. (2025). Compiler-integrated, conversational ai for debugging cs1 programs. In *Proceedings of the 56th acm technical symposium on computer science education v. 1* (p. 994–1000). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/3641554.3701827

Riazi, S., & Rooshenas, P. (2025). Llm-driven feedback for enhancing conceptual design learning in database systems courses. In *Proceedings of the 56th acm technical symposium on computer science education v. 1* (p. 1001–1007). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/3641554.3701940

Roest, L., Keuning, H., & Jeuring, J. (2023). Next-Step Hint Generation for Introductory Programming Using Large Language Models. In *Proceedings of the 26th Australasian Computing Education Conference* (pp. 144–153). Sydney, Australia: ACM. doi: 10.1145/3636243.3636259

Roethlisberger, F. J., & Dickson, W. J. (1939). *Management and the Worker.* Cambridge: Harvard University Press.

Sarsa, S., Denny, P., Hellas, A., & Leinonen, J. (2022, August). Automatic generation of programming exercises and code explanations using large language models. In *Proc. icer.* ACM. doi: 10.1145/3501385.3543957

Savelka, J., Agarwal, A., Bogart, C., & Sakr, M. (2023). *Large language models (gpt) struggle to answer multiple-choice questions about code.*

Scholl, A., & Kiesler, N. (2024). How novice programmers use and experience chatgpt when solving programming exercises in an introductory course. In *2024 ieee frontiers in education conference (fie)* (p. 1-9). doi: 10.1109/FIE61694.2024.10893442

Scholl, A., & Kiesler, N. (2025a, Jul). *Data: Students' feedback requests and interactions with the script chatbot - do they get what they ask for?* OSF. doi: 10.17605/OSF.IO/TG5R3

Scholl, A., & Kiesler, N. (2025b). Script - supportive chatbot for resolving introductory programming tasks. In *Proceedings of the 30th acm conference on innovation and technology in computer science education v. 2* (p. 759). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/3724389.3730786

Scholl, A., Schiffner, D., & Kiesler, N. (2024). Analyzing chat protocols of novice programmers solving introductory programming tasks with chatgpt. In *Proc. delfi 2024* (pp. 63–79). doi: 10.18420/delfi2024_05

Spohrer, J. C., & Soloway, E. (1986). Novice mistakes: Are the folk wisdoms correct? *Communications of the ACM*, *29*(7), 624–632. doi: 10.1145/6138.6145

Szabo, C., Parker, M. C., Friend, M., Jeuring, J., Kohn, T., Malmi, L., & Sheard, J. (2025). Models of mastery learning for computing education. In *Proceedings of the 56th acm technical symposium on computer science education v. 1* (p. 1092–1098). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/3641554.3701868

Taylor, A., Vassar, A., Renzella, J., & Pearce, H. (2024). dcc –help: Transforming the role of the compiler by generating context-aware error explanations with large language models. In *Proceedings of the 55th acm technical symposium on computer science education v. 1* (p. 1314–1320). New York: ACM. doi: 10.1145/3626252.3630822

Wermelinger, M. (2023). Using github copilot to solve simple programming problems. In *Proceedings* *of the 54th acm technical symposium on computer science education v. 1* (p. 172–178). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/3545945.3569830

Whalley, J., Clear, T., & Lister, R. (2007). The many ways of the Bracelet project. *BACIT*.

Xiao, R., Hou, X., & Stamper, J. (2024). Exploring how multiple levels of gpt-generated programming hints support or disappoint novices. In *Extended abstracts of the 2024 chi conference on human factors in computing systems.* New York, USA: ACM. doi: 10.1145/3613905.3650937

Yeh, T. Y., Tran, K., Gao, G., Yu, T., Fong, W. O., & Chen, T.-Y. (2025). Bridging novice programmers and llms with interactivity. In *Proceedings of the 56th acm technical symposium on computer science education v. 1* (p. 1295–1301). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/3641554.3701867

Zhai, X. (2022). Chatgpt user experience: Implications for education. doi: http://dx.doi.org/10.2139/ssrn.4312418

Zhang, J., Cambronero, J., Gulwani, S., Le, V., Piskac, R., Soares, G., & Verbruggen, G. (2022). Repairing bugs in python assignments using large language models. *arXiv preprint arXiv:2209.14876*.

Towards a Model of Library Use

Ava Heinonen
Department of Computer Science
Aalto University
Ava.Heinonen@aalto.fi

Abstract

Programmers often struggle when using libraries because of difficulties in understanding how the library can be used to achieve their desired outcome. Much of the existing literature has focused on API usability and documentation. However, limited research has been done to gain insight into the processes we seek to support with documentation and usable APIs — the processes of understanding and using libraries.

In this work-in-progress paper, we present initial results of a study examining programmers' cognitive processes and mental model development as they refactor an open-source web application to use a new library. We discuss the analysis of the pilot protocol, and the initial insights gained through this analysis. The initial insights suggest that developers form understanding of the library and the solution not only through seeking information but also through interaction with the library by implementing and testing code.

1. Introduction

Software libraries can be difficult to use (Robillard & DeLine, 2011; Samudio & LaToza, 2022). One study on the problems programmers face when developing web applications indicated that many of the problems were caused by programmers struggling to understand how to use libraries and debug solutions implemented using libraries (Samudio & LaToza, 2022). Studies on library-related questions and help requests from programmers show that these problems are not limited to web programming (Wang & Godfrey, 2013; Hou & Li, 2011). As modern software development is highly based on the use of libraries and other reusable assets, finding ways to help programmers understand how to use libraries becomes increasingly important (Taivalsaari, Mikkonen, & Mäkitalo, 2019).

Research on learning and using libraries has focused on API usability and documentation — finding solutions for the problem of understanding and using libraries. Studies have sought assessed programmer difficulties with API documentation (Robillard & DeLine, 2011), evaluated ways to create effective documentation (Meng, Steinhardt, & Schubert, 2020), and developed ways to assess API usability (Piccioni, Furia, & Meyer, 2013). However, limited research has been conducted to analyze the processes we seek to support with API design and documentation — the problem of understanding and using libraries.

In recent years there have been efforts to bridge this gap in the literature. Studies have sought to model and analyze programmers' information seeking behavior when developing solutions using an unfamiliar library (Kelleher & Ichinco, 2019; Kelleher & Brachman, 2023; Sparmann & Schulte, 2023). One approach was the COIL model, that sought to model programmers' information seeking behavior when learning to understand new libraries (Kelleher & Ichinco, 2019; Sparmann & Schulte, 2023). The model describes three stages of the library use process: Information collection stage consisting of seeking and acquiring relevant information from online information sources, Information organization stage where the acquired information is organized to form a solution, and solution testing stage where the solution is implemented and tested (Kelleher & Ichinco, 2019). Another approach was the utilization of data-frame theory of sensemaking to investigate the cognitive processes underlying programmers information seeking (Kelleher & Brachman, 2023). Focusing on online information seeking behavior, they analyzed programmers search terms and navigation behavior to identify different stages in the sensemaking process (Kelleher & Brachman, 2023). These studies bring insight into developer behavior when developing the required understanding to implement solutions using unfamiliar libraries. However, these studies do not aim to analyze the motivations underlying the behavior. Therefore, the question of what programmers seek to achieve, and what is the understanding they seek to form to achieve it remains.

While the recent studies have focused on programmer behavior, literature from the early years of software libraries and software reuse developed theories on the tasks of using libraries and the understanding required to conduct these tasks (Krueger, 1992; Détienne, 2002; Fischer, 1987). These studies identified what a programmers has to do to successfully use a library. They theorize, that library use would include locating and selecting suitable artifacts to use (Krueger, 1992), coordinating artifacts to form solutions (Fischer, 1987), modifying artifact behavior through arguments to achieve the desired behavior (Krueger, 1992; Détienne, 2002), and integrating artifacts into the program code (Krueger, 1992). Early literature on reuse also theorizes about the types of knowledge programmers would need to conduct reuse tasks (Fischer, 1987; Krueger, 1992). This includes knowledge about the library, the artifacts it provides, and how those artifacts can be configured into solutions (Krueger, 1992; Fischer, 1987). It also includes knowledge about the function and interface of the individual artifacts (Krueger, 1992; Fischer, 1987). While it is quite possible that these theories apply to modern software reuse as well, technology has rapidly changed in the last 30 years. Therefore, assessing the degree to which these theories hold in modern programming environments is required.

In this paper, we discuss an ongoing study investigating the tasks of library use in modern software development environments. The study aims to analyze the tasks involved in using and selecting libraries, and the understanding required to complete these tasks. Specifically we seek to answer the following research questions:

RQ1 What are the tasks included in using and selecting libraries?

RQ2 What do developers need to understand about the library and the task situation to conduct these tasks?

RQ3 How do developers form this understanding?

- RQ3.1 What information do developers seek?
- RQ3.2 What activities do programmers conduct to acquire and process information?

RQ4 How do programmers move between the tasks during a library use situation?

To this end, we analyze the think-aloud protocols of professional developers learning to use a library for a refactoring task. In this work-in-progress paper, we discuss the study methodology and the ongoing data analysis process. Furthermore, we discuss our initial observations.

2. Methodology

We used the think-aloud method to study developers' cognitive processes when refactoring an open-source web application to use a new library. The think-aloud method is a well-established technique used to study cognitive processes in various fields (Ericsson & Simon, 1980; Panadero, Pinedo, & Ruiz, 2025).

2.1. Study Task and Study Process

In the study, participants worked on refactoring the front-end of an open source Web Application Habitica[1] to use a new time and date library instead of Moment.js. Each participant had one hour to work on selecting a library to use and beginning to refactor the Habitica front-end. The participants were assured that they were not expected to complete the task. Participants were allowed to select which parts of the front-end code to refactor. However, they were instructed that one possible starting point could be the calculateTimeTillDue() method in the task.vue file. This was done so that participants could focus on the selection and refactoring tasks and not on deciding on a suitable starting point.

[1]https://github.com/HabitRPG/habitica

Participants were provided a laptop. The laptop had Visual Studio Code IDE with the Habitica source code already open. Habitica was already running in the background and the participants could view the Habitica application in the browser. The browser window with Habitica had developer tools already open. In another browser tab, we had a list of potential Moment replacements provided by the Moment team open [2].

Before the study, participants were asked to respond to a background survey. The survey contained eight questions about the participant's employment, development experience, and familiarity with the technologies used in the study.

The study procedure was divided into two phases. In the first phase, participants were informed about the study and the study task. Participants received verbal and textual descriptions of the study task and the opportunity to ask questions. Participants were also instructed about think-aloud.

In the second phase, participants worked on a refactoring task. The laptop screen was recorded, the participant was filmed, and their voice was recorded during the task. If the participant requested not to be filmed, only their vocalizations were recorded.

A researcher sat with the participant, making notes, and asking the participant to vocalize their thoughts if they were silent for more than 30 seconds.

2.2. Participants

Twelve participants participated in the data collection. All participants were, at the time of data collection, employed in software development roles. All participants had some experience with JavaScript and web development.

Participants were allowed to speak either Finnish or English during the think-aloud. All participants were required to have good enough English or Finnish skills to communicate their thoughts in either language. Two of the participants spoke Finnish, and eight spoke English. However, from the English speakers all spoke English as a second language.

Due to technical difficulties, the data collected from two participants could not be used. In these two cases, no sound was recorded, and thus the think-aloud data was lost. Therefore, the final data set contains data from 10 participants.

Participants were recruited through posters and contacts in companies and other organizations. Participants received a 10 euro gift card to a coffee shop for participation.

2.3. Data Analysis

Data analysis is currently in progress. We use qualitative coding to analyze the think-aloud protocols. The protocols were first segmented into segments corresponding to one *activity*. Then a codebook was developed using existing literature on library learning and use, and the analysis of a pilot protocol. The protocols were then coded, while iteratively improving on the codebook as new topics emerge from the data.

2.3.1. Segmentation

First, after the think-aloud protocols had been transcribed, the protocols were segmented into segments corresponding with a an *activity*, a coherent set of actions that achieve one goal. For example, one activity could be writing a piece of code, information seeking activity such as searching or reading, or cognitive activity such as formulating a plan.

2.3.2. Code Book and Code Book Development

Initial codebook was developed using existing literature on library learning and use, and the analysis of a pilot protocol. The initial codebook was then tested by applying it to protocols, and further revised. The initial codebook was then iteratively revised throughout the coding process.

The initial codebook contains the following codes: 1) Activities, 2) Information needs, 3) Mental Mod-

[2]https://blog.logrocket.com/5-alternatives-moment-js-internationalizing-dates/

els, 4) Tasks, and 5) Stages.

Activities refer to activities conducted by the participant. These are identified based on both the transcript and the screen recording.

Information needs refer to the information participant is seeking to acquire. For example, in the quotation below participant seeks information about the return value of an API method:

> "**. . . What does this return? The diff function...**
>
> . . . Maybe it already returns the duration.
>
> Let's see.
>
> [...] . . . Okay, so it already returns a duration"

Mental Models refer to participant's understanding of some aspect of the library or the task and the task context. For example, in the quotation below participant utterances indicate their understanding of what an API method does:

> "Returns the end of date for the given date.
>
> **Okay, so it will also change it to just different time, but the same date.**"

Tasks refer to the task participant is working on. We used existing literature on library use to identify an initial set of tasks we expected to encounter in the data. This list was further refined as more data was analyzed. For example, in the quotation below participant identifies an API method that they believe they can use in their solution:

> "...so now I'm looking if they have like a function that is named in a similar way, because it's like a pretty intuitive way to name a function.
>
> So I search,uh, subtract.
>
> The defined number of seconds from the given date.
>
> So it looks like this is this function."

Stages refer to the stage of the task. During the analysis of the pilot protocol, we identified four stages of a refactoring task. These were: 1) Understanding the goal, where participant forms good enough understanding of the functionality to be refactored to start refactoring it, 2) Solution Design, where participant forms an understanding of how the desired functionality can be achieved using their selected library, 3) Solution Implementation, where participant implements the solution, and 4) Solution evaluation where the participant seeks to evaluate to what extent the implemented solution achieves their goals. For example, the below quotation is from an evaluation stage where the participant prints the value returned by an API call they had implemented to verify that it works as they intended it to:

> "So, I'm just going to paste this here, just to see if it prints falses in the console.
>
> . . . date is false which okay good I consider this a success"

2.3.3. Current Status

Data analysis is currently ongoing. The first round of coding has been conducted using the initial codebook. The codes from the first round are currently being analyzed and consolidated to develop the final codebook.

After the final codebook has been developed, a second round of coding will be conducted using the final codebook.

2.4. Initial Insights

The first round of coding has allowed us to develop some initial insights. In contrast to the COIL model that highlights the importance of online information sources in learning to use libraries (Kelleher & Ichinco, 2019), our data seems to indicate that programmers also learn to understand how artifacts work and how they can be used by trial and error and by observing their behavior in a program. Multiple participants indicated, that as they were implementing code they were trying things out to see how it works or if it would work. One participant even commented on this stating *"I find it easier to try things out rather than think"* as they were testing which way around they should order two API methods.

Participants also indicated, that using artifacts allowed them to test their hypotheses about the artifacts and their use. For example, one participant was trying to figure out the arguments to give to an API method by reading API documentation, and stated: *"ok anyway, I think if I don't import and don't test it I don't get any clue that my understanding is correct from this"*.

Our current hypothesis is, that interaction with the IDE plays an important role in learning to understand how library artifacts work and can be used. It seems to be used for hypothesis verification by testing out hypotheses of artifacts and their use. It may also be used for gaining insight into code behavior when the programmer does not have good enough mental models to be able to mentally simulate code execution.

This would be in contrast with prior literature on API learning (Kelleher & Ichinco, 2019; Sparmann & Schulte, 2023; Kelleher & Brachman, 2023), in which learning is assumed to occur primarily through reading online information sources. This would also indicate avenues for supporting learning to understand libraries through IDE tools.

3. References

Détienne, F. (2002). Software reuse. In *Software design—cognitive aspects* (pp. 43–55). Springer.

Ericsson, K. A., & Simon, H. A. (1980). Verbal reports as data. *Psychological review*, *87*(3), 215.

Fischer, G. (1987). Cognitive view of reuse and redesign. *IEEE Software*, *4*(4), 60.

Hou, D., & Li, L. (2011). Obstacles in using frameworks and apis: An exploratory study of programmers' newsgroup discussions. In *2011 ieee 19th international conference on program comprehension* (pp. 91–100).

Kelleher, C., & Brachman, M. (2023). A sensemaking analysis of api learning using react. *Journal of Computer Languages*, *74*, 101189.

Kelleher, C., & Ichinco, M. (2019). Towards a model of api learning. In *2019 ieee symposium on visual languages and human-centric computing (vl/hcc)* (pp. 163–168).

Krueger, C. W. (1992). Software reuse. *ACM Computing Surveys (CSUR)*, *24*(2), 131–183.

Meng, M., Steinhardt, S. M., & Schubert, A. (2020). Optimizing api documentation: Some guidelines and effects. In *Proceedings of the 38th acm international conference on design of communication* (pp. 1–11).

Panadero, E., Pinedo, L., & Ruiz, J. F. (2025). Unleashing think-aloud data to investigate self-assessment: Quantitative and qualitative approaches. *Learning and Instruction*, *95*, 102031.

Piccioni, M., Furia, C. A., & Meyer, B. (2013). An empirical study of api usability. In *2013 acm/ieee international symposium on empirical software engineering and measurement* (pp. 5–14).

Robillard, M. P., & DeLine, R. (2011). A field study of api learning obstacles. *Empirical Software Engineering*, *16*, 703–732.

Samudio, D. I., & LaToza, T. D. (2022). Barriers in front-end web development. In *2022 ieee symposium on visual languages and human-centric computing (vl/hcc)* (pp. 1–11).

Sparmann, S., & Schulte, C. (2023). Analysing the api learning process through the use of eye tracking. In *Proceedings of the 2023 symposium on eye tracking research and applications* (pp. 1–6).

Taivalsaari, A., Mikkonen, T., & Mäkitalo, N. (2019). Programming the tip of the iceberg: software reuse in the 21st century. In *2019 45th euromicro conference on software engineering and advanced applications (seaa)* (pp. 108–112).

Wang, W., & Godfrey, M. W. (2013). Detecting api usage obstacles: A study of ios and android developer questions. In *2013 10th working conference on mining software repositories (msr)* (pp. 61–64).

Analogy between Interdisciplinarians and Business Analysts in IT

Anna Bobkowska
Faculty of Computer Science
Gdynia Maritime University
a.bobkowska@wi.umg.edu.pl

Abstract

This paper presents a pragmatic approach to interdisciplinary studies. It argues for large diversity of interdisciplinary endeavors which is reflected in diversity of approaches, domains and characteristics of projects, diversity of broad questions and integrative answers, diversity of interdisciplinary problems and solutions, and diversity of possible research approaches to interdisciplinary studies. It presents lessons learned from a case of proposing interdisciplinary approach for very large IT projects for public administration. This case includes review of interdisciplinary approaches and dimensions which facilitate comparison and selection of proper means for a specific problem at hand. Then, it focuses on a profile of interdisciplinarian in analogy to the business analyst working in the area of information technologies. It draws parallels regarding evolution of notions to focus on business analyst, planning interdisciplinary work, conceptual models which can be applied for representing disciplinary knowledge, and specific roles played by both business analysts and interdisciplinarians. Finally, it discusses limitations of this analogy as well as its implications for practice.

1. Introduction

Interdisciplinary studies gain increasing popularity in research and practice. One can notice a large diversity of interdisciplinary endeavors which may vary in domain, type, size, number of stakeholders, implications and other characteristics. Interdisciplinary projects appear in pure science, humanities, social and health sciences and problem-solving in practice. They can be related to private investigations, searching for solutions of real problems where any discipline alone doesn't deliver satisfactory solution, or in research ranging from simple knowledge transfers to formation of entirely new interdisciplinary fields of study, e.g. biomedical engineering. With the increasing social impact of applications of information technologies (IT), one can also notice the increasing need of stakeholders from several disciplines in software projects.

In this diversity of interdisciplinary projects and variety of possible approaches, the question from pragmatic perspective is: what is the essence of interdisciplinarity? The answer can be given by motivation why such projects are necessary, which indicates usually for the need for answers or solutions which are not available in a single discipline. It includes also the need for unified theories which cover a broader scope of phenomena. Another answer can be given based on the fact of involving specialists representing knowledge from several disciplines or on requirements for using knowledge from several disciplines (no matter of motivations). Yet another answer can be given according to the activity which is specific to interdisciplinary studies. And this is integration of knowledge. It can appear in answers, solutions, provisional inputs and outcomes, methods for integration, work of integrators, team collaboration, management and related issues, which overall constitute a field of integrative studies.

There are two goals of this paper. The first goal is to contribute to better understanding of interdisciplinary approaches to IT projects by sharing review of interdisciplinary approaches, dimensions and lessons learned from research on interdisciplinary IT projects for public administration. The second goal is to elaborate on analogy between interdisciplinarians and business analysts in IT.

A variety of approaches has been proposed in the area of interdisciplinary studies. As they were discovered in different context, they focus on different aspects of interdisciplinary research or practice. Human factors are a part of many approaches, but seldom are in the center of research. They can be found in programs of studies aiming at developing interdisciplinarians. Interdisciplinarians can be characterized as individuals who are prepared to participate and/or lead interdisciplinary projects. They should have a broader perspective than one discipline as well as knowledge and skills related to integrative studies. So, when elaborating on analogy between interdisciplinarians and business analysts in IT, we follow the trend of professional development of interdisciplinarians. As a frame of reference regarding business analysis we use IIBA BABOK Guide (International Institute of Business Analysis, 2015). This is a popular handbook for business analysts as it resulted from about ten years of standardization efforts of the community of professionals, followed by professional certification of business analysts. As a representation of the state of the art for interdisciplinary studies, we use results of the review of interdisciplinary approaches presented as realization of the first goal.

The paper is structured as follows. Section 2 is related to the first goal and it describes lessons learned from a case of proposal of interdisciplinary approach for IT projects for public administration. It contains a review of interdisciplinary approaches and indicates for dimensions in order to facilitate their comparison and effective application. Section 3 aims at fulfilling the second goal and it points out selected parallels between interdisciplinarians and business analysts in IT. It discusses also limitations of this analogy, reliability issues as well as implications for practical application. Section 4 draws conclusions.

2. Lessons Learned about Approaches to Interdisciplinary Studies

2.1. Context

These studies were conducted in context of seminars aiming at supporting decision-makers and project managers of very large IT projects for public administration by presenting research results in fields of economics, management and systems engineering. Interdisciplinary approach was the focus and characteristic feature of our contribution.

In the first step (Bobkowska, 2014), we have identified disciplines which are relevant for this kind of projects and proposed a framework for integrating them at high level of abstraction. The disciplines included: Systems Engineering, Business Process Modeling and Re-engineering, User eXperience, Domain Analysis, Project and Program Management, Legal Regulations, Public Administration, Social Communication (Public relations), Social Impact of IT Applications, Integrative studies (IT- Public Administration -Law). As one can see, they are related to IT projects, broad range of issues in public administration and emerging disciplines on the edge of the previous. For each discipline, we have expressed perspective statement and a number of issues which can be addressed to add value to the projects. Regarding the high level integration, we have proposed disciplines which are related to the final products (IT systems) and which are known to decision-makers and project managers. Thus, the framework has utilized first of all the knowledge from Project and Program Management. Other disciplines were connected to extended analysis or design in systems engineering. For example, legal regulation analysis, social analysis, user/citizen analysis extended typical requirements analysis, and they influenced system design extended by User eXperience design for identified groups of users/citizens together with application of public relation results as well as administrative process re-engineering results. In fact, this is the scope of work typical for business analysts extended by impacts from other disciplines.

In the second step (Bobkowska, 2015), we were searching for approaches of integrating knowledge also at lower levels of abstraction. It resulted in a review of available approaches (described briefly in the following sections) together with the dimensions which facilitate comparing them and searching for useful solutions when project managers encounter any interdisciplinary problem to be solved.

2.2. Creativity and Intuitive Integration

Creativity and intuitive approaches are very popular as they are low-cost for entry approaches. The reason to apply them, despite lack of predictability and methodological rigor, can be explained by the fit of these methods to the specifics of integrating knowledge from various disciplines. Intuition and creativity is useful in cases where the problem is unique, and there are: no single correct answer, no precise procedures, no precise verification criteria, and the solution is multidimensional. Further arguments are provided by the results of psychological research on the nonconscious processing of information by the human mind (Lewicki et al., 1992), which show that nonconscious mind has excellent cognitive capabilities. The only problem is that, due to its nature, they elude human control. However, with good understanding the characteristics of nonconscious mind, the process can be directed in some way. Best practices for applying creative and intuitive methods during knowledge integration include:

• gathering together experts from all relevant disciplines and/or all related knowledge units before beginning intuitive information processing (preparation phase);

• enough time to work with a mind unloaded by other problems – only under such conditions the human mind can generate valuable solutions;

• formulate objective criteria or methods for verifying results – not every result of intuitive processing is correct, and sometimes the feeling of finding a solution is not reliable (a solution is found, but it is not the right solution);

• counteraction to errors resulting from social phenomena, such as egocentrism, overconfidence, and the resulting subjective sense of rightness when someone is objectively wrong.

2.3. Boundary Objects

The issue of integrating usability and software engineering techniques (Bakalis, Folmer and Bosch, 2004) arose in the context of terminological inconsistency, methodological misalignment, and practical difficulties in applying techniques from these disciplines together in IT projects. A solution was proposed using the concept of boundary objects. A boundary object is an abstraction of a shared conceptual space, shared knowledge, common products and assumptions used to integrate and coordinate the activities of individuals or teams with different specializations. It seems that the concept of the boundary object is universal and can be useful for integrating knowledge from other disciplines as well. It corresponds to the dimension of knowledge types in interdisciplinary studies.

From the perspective of boundary objects, one might consider whether a boundary object refers to knowledge shared by experts from two related disciplines or whether it is common knowledge for the entire team. In terms of levels of abstraction, the fundamental issue is the required level of detail of knowledge transferred from an expert to others. That is, whether the entire team should acquire the expert knowledge necessary for collaborative problem solving, or whether a certain abstraction is sufficient to make the expert's results understandable to other project participants. Boundary objects can be classified according to the 3P (People-Process-Product) model. In the category of people, boundary objects are represented by team leaders or individuals trained in both disciplines. In the process category, the boundary object appears as integrated process. Integration elements include collaborative decision-making processes, result validation processes, meetings, brainstorming sessions, and discussions of ideas, procedures, and document content. In the product category, boundary objects are made of various specifications and documents, e.g. interface prototypes, use cases and scenarios, specifications, reports, presentations, and functionality and usability review documentation.

2.4. Meta-Models for Integration

The idea of meta-models is well-known in the field of conceptual modeling (Object Management Group, 2017) and used in order to precisely define model elements. The primary applications of meta-models include understanding what models represent as well as development of data repositories and tools to support modeling. More advanced applications include formal comparison of modeling languages, use of meta-models as a language in meta-tools, and exchange between different tools. But the idea of meta-modeling can be used also in a general meaning. An example of such an application is an earlier concept of meta-informatics (Engelbart, 1992), used to define the structure of categories in organizations to improve their operational efficiency. In this extended sense, meta-modeling can be applied to knowledge integration, as it enables:

• distinguishing categories for understanding, e.g. terminology in a specific discipline,

• comparing different approaches and theories,

• integrating techniques from different disciplines at the meta-model level,

• developing new knowledge structures, e.g. in learning organizations.

Meta-modeling has significant potential for integrating knowledge from different disciplines. Creating meta-models can facilitate identification of categories used by experts of different disciplines. Meta-modeling can help in comparing competing theories and selecting proper one. It can help in defining shared units of knowledge. Integrated meta-models can help to create common ground when solving interdisciplinary problems. It corresponds to the dimension of meta-knowledge.

2.5. Paradigm Integration in Multi-Paradigm Modeling

Multi-paradigm modeling (Hardebolle and Boulanger, 2009) addresses the need to overcome technological heterogeneity and reason about the global properties of heterogeneous systems. A good example is the integration of technologies in smartphones, which requires the combination of various technologies: power supply, user interaction, data processing, microwave (Bluetooth), voice acquisition, voice compression and decompression, voice amplification, video compression and decompression, radio, and GPS. This is a formal approach that, at the conceptual level, can also be applied to integration of non-technological domains. Integration mechanisms in multi-paradigm modeling include: model transformations at the model and meta-model level; model compositions from different disciplines; model translations between modeling languages; heterogeneous interactions; multi-view and multi-abstraction mechanisms; component-based approaches and their interconnection; and co-simulation. This approach represents meta-knowledge dimension at the advanced technological level.

2.6. Integration in Interdisciplinary Theories

The problem of knowledge integration is recognized as a significant challenge in all fields of interdisciplinary studies including pure science, social sciences, and humanities (Repko, 2008). The following mechanisms for knowledge integration can be applied:

- informal metaphors or analogies,
- building connections between theories, concepts, and disciplines (bridging),
- identifying conflicts between knowledge units and overcoming them by building common ground,
- extending the meaning of an assumption or term from one discipline to others (extension),
- developing a theory to include additional phenomena by adding variables from additional studies, perspectives, disciplines, etc. (theory expansion),
- redefining terms or assumptions to establish a common meaning (redefinition),
- reorganizing through finding common features, redefinitions, and alignments (organization),
- transforming opposing assumptions or propositions into continuous variables that serve as parameters in particular situations (transformation),
- borrowing and transferring knowledge between disciplines (borrowing and transfer).

The goal of using these mechanisms is to integrate insights from appropriate disciplines to the solution of a given problem and to generate an interdisciplinary understanding.

2.7. Interdisciplinary Teams

From the perspective of increasing performance of cross-functional teams (Parker, 2002), the key issue is building and supporting interdisciplinary teams in order to increase probability of their success. These research results are based on interviews with participants of interdisciplinary projects in various industries. The best practice has been identified, including: clear common goals, shared commitment, joint training, a reward system that recognizes contribution to teamwork, empowering the team to make decisions, and explicitly defined rules of working in a team whose members may come from diverse organizational cultures.

In this approach, precise integration techniques are not in the focus. Integration occurs as the collaborative activity, meaning that knowledge is developed and shared by the team which uses both formal and informal methods. This perspective provides valuable insights into the organizational conditions under which knowledge integration occurs successfully.

2.8. Dimensions

The state of the art regarding interdisciplinary approaches should be analyzed in entire its diversity: diversity of approaches, diversity of projects, diversity of integrative questions and answers, diversity of interdisciplinary problems and solutions, diversity of context of research on interdisciplinary studies, etc. On the other hand we need some categories to compare approaches and select proper interdisciplinary method for a given case. In practice, we must work in multi-dimensional space, which additionally might change in time or in detail or a combination of single aspects forms proper solution. In order to capture categories for analysis, comparison and action in practice, dimensions are a useful concept. They allow to see all the approaches in a big picture. From methodological perspective, they constitute a kind of common ground for the diversity of approaches.

The following dimensions can be distinguished:

• regarding personnel: individual (competencies) and social (team work and culture),

• regarding knowledge: expert knowledge and shared knowledge, as well as the distinction between interface-knowledge and knowledge about categories (meta-knowledge),

• regarding activities: execution and management dimensions,

• sub-dimensions in the execution dimension: creative-intuitive and formal-methodological execution, where methods can be focused on solution delivery or integration alone,

• sub-dimensions in the management dimension, e.g. the selection of appropriate methods and interdisciplinary team management.

2.9. Lessons Learned

To summarize, the following lessons can be learned from this case:

- We need a framework for interdisciplinary studies. It may come from general interdisciplinary methods as well as selected disciplinary methods or structures.
- Interdisciplinary approaches increase complexity of the projects. Apart from typical tasks related to IT projects, one should conduct analysis from perspectives of other disciplines and assess impact of these disciplines on the project. This impact might be of several kinds: a requirement, a task to be performed, a rule for compliance analysis, an artefact to integrate with other artefacts, etc.
- There is a need for representations both at high level of abstraction and detailed levels of abstraction.
- It seems that there are often many possible approaches, and individual experience is one of the factors when selecting framework approach.
- The approaches to interdisciplinary studies come from different backgrounds. Different context of discovery results in focus on different aspects of interdisciplinary studies. In order to facilitate comparison or search for needed methods or artefacts, we have proposed the dimensions and assigned approaches to them with indication of the level of advancement and coverage of issues in a given dimension by a given approach.

3. Interdisciplinarians vs. Business Analysts in IT

3.1. Focus on Interdisciplinarian

Taking into account the diversity of possible interdisciplinary projects and the variety of possible interdisciplinary approaches (together with the need to apply approaches specific for a given discipline in order to reach the goal), it is reasonable to focus on interdisciplinarian who will be able to conduct and manage any type of interdisciplinary project, instead of proposing yet another methodology. Methods, techniques, product templates, rules and fundamentals of interdisciplinary knowledge are also useful, but their role is to provide a kind of toolbox for interdisciplinarians for choosing proper means to reach their goals. Additionally, important aspect of such approach is training of interdisciplinarians' competencies.

The way of thinking about supporting business analyst in IT evolved also from proposing methods, especially methods in the areas of requirements engineering, systems analysis, business process management, later also project management or business acumen. With growing size of software projects, roles of analysts together with their responsibility for products and tasks were defined (IBM RUP, 2006). The notion of the role gave space for specifying skills necessary for a given role and integrating them within methodology. Professional business analysts found out a large number of possible roles in practice and resigned from classifying them. In the IIBA BABOK Guide (International Institute of Business Analysis, 2015), business analyst is defined as “any person who performs business analysis tasks described in the BABOK® Guide, no matter their job title or organizational role.” The content of this guide contains key concepts, knowledge areas which collect tasks typical for business analysis, the list of techniques that might be useful for business analysts as well as specification of underling competencies which “provide a description of the behaviours, characteristics, knowledge, and personal qualities that support the effective practice of business analysis.”

Interdisciplinarians also have different positions or responsibilities in interdisciplinary projects. It would be rather difficult to classify them precisely. Thus, this flexibility in definition of interdisciplinarian seem to be a good choice. Literature on integrative studies contains description of theories, propositions of an integrative process as well as several detailed findings for validation of interdisciplinary studies. They correspond to knowledge areas, tasks and techniques and can be included in the interdisciplinarian’s toolbox. In the handbook on how to become interdisciplinarian (Augsburg, 2006), one can also find the lists of competencies and abilities for which interdisciplinarians are valued in the job market. Surprisingly, many of them are similar to competencies of business analyst, e.g. ability to think conceptually, ability to identify and solve problems, ability to understand other value systems, ability to evaluate alternatives, ability to decide on a course of action, effective written and oral communication skills, effective team work or ethical sensitivity.

3.2. Interdisciplinary Planning and Monitoring

Project planning is traditionally a part of project managers tasks. However, it has been discovered that due to the complexity and diversity of business analysis as well as the need for understanding directly business analysis situation, it is more effective when plans of business analysis are made by business analysts. All these is done in context of clear split of responsibilities between project managers focusing on process and business analyst focusing on products and its validation, as well as efficient collaboration between these roles especially in case of overlapping issues. IIBA BABOK Guide (International Institute of Business Analysis, 2015) describes the knowledge area called “Business Analysis Planning and Monitoring”, which contains the following tasks: plan business analysis approach, plan stakeholder engagement, plan business analysis governance, plan business analysis information management, and identify business analysis performance improvements.

By analogy, interdisciplinarians should also plan and monitor their activities due to complexity and diversity of interdisciplinary projects. They should also collaborate with several stakeholders including project managers. They should be confident about the unique value they bring to the project, i.e. knowledge and skills related to integrative studies. Comparing to business analysts, they have a new challenge related to understanding perspective of each discipline involved and its potential impact on the project together with its limitations. Another challenge of such planning is how to make the best of a given discipline in a given project when integrating the outcomes from all involved disciplines.

Interdisciplnarians can use the best practice captured in tasks of business analysts. They can consider several aspects when planning approach to interdisciplinary projects including preference for level of formalism or number of iterations. They can be more disciplined in rational planning of stakeholder engagement together with adequate ways of collaboration and communication of results. They can also pay special attention to effective strategies and satisfactory collaboration with decision-makers. Another interesting aspect to focus on is interdisciplinary information management which makes a space for integration of best practice from business analysis with specifics of interdisciplinary studies. And finally, business analysts are aware that something might go wrong and identify ways of performance improvements in advance. Having them, they can act effectively even when they encounter threats to effective business analysis realisation. This strategy of risk management can be used also by interdisciplinarians.

3.3. Conceptual Models of Discipline Knowledge

An important achievement of systems analysis (adopted later by business analysis) has been use of conceptual modelling. This shift of paradigm has led to thinking in terms of concepts which represent useful aspects of the reality or system, instead of words which describe them. It allowed for effective understanding of complex systems. It made also foundations for multi-view representation of the same system. The specification of OMG UML (Object Management Group, 2017) delivers a standard modelling language for visualizing, specifying, constructing, and documenting artefacts of systems. Depending on needs, several configurations of methods can use OMG UML for system documentation at different levels of abstraction. However, essential feature of all these configurations is consistent model of the system emerging form several diagrams representing the perspectives of systems' functionality, structure, specific aspects of behaviour or other features. For more advance applications, domain-specific modelling languages can be applied (Kelly S., Tolvanen J-P., 2008).

When considering application of the conceptual modelling in area of interdisciplinary studies, one can notice that theories in disciplines are, in fact, also a kind of conceptual models which represent a perspective of viewing reality. And in similar way, we can steer to one consistent model which has several views represented by different disciplines. For example, in interdisciplinary analysis, one can extend traditional analysis by adding legal perspective, social perspective or user experience perspective. This is consistent with core of meta-informatics concept in its extended application. Regarding modelling languages, some elements of standard modelling languages can be applied as they define universal modelling perspectives such as structure or behaviour, but interdisciplinarian should be flexible for extending them with use of new profiles for theories from other disciplines or even make use of knowledge about defining domain-specific modelling languages. Some concepts, such as goals and scope for systems analysis, and by analogy: values, perspective description and limitations of a given disciplinary approach, might be difficult to capture by modelling, thus a mixture of visual and textual specifications is a good solution. There are also many new challenges, e.g. how to model different kinds of boundary objects. Finally, perfect integration means a consistent multi-view interdisciplinary documentation which fulfils quality criteria of each discipline.

3.4. Interdisciplinarians in Different Roles

Integration of knowledge is essential in interdisciplinary studies. Business analysts work on the edge of issues related to organisation, business processes, system requirements, domain knowledge, related regulations, non-functional requirements and several specific expectations of stakeholders. Although integration of knowledge is not stressed the description of business analysis, business analysts perform integrative tasks in practice.

Thus, the question is what we can learn from business analysts about integration. The most interesting insights are delivered by competencies in the category of interactive skills. This category contains the following competencies: facilitation, leadership and influencing, teamwork, negotiation and conflict resolution, and teaching. There is another category of communication skills with verbal, non-verbal, written communication and listening. And all of them should be interpreted in context of the skill of adaptability for variety of organisations, stakeholders and situations in rapidly changing environments. This means that business analyst should be prepared to play different roles including the roles of leaders, integrators, mediators, team members, experts and facilitators. Regarding the role of integrator, it is good to connect it with the skill of learning. So, the integration made by business analysts is related to learning new knowledge, then transforming it in course of several individual activities as well as team activities, and finally communicating it to stakeholders (teaching) in a meaningful way with use of "the most appropriate visual, verbal, written, and kinesthetic teaching approaches".

The analogy seems quite clear. Interdisciplinarians also should be flexible and adapt to stakeholders who represent different disciplines, projects and situations. They can play similar differentiated roles in projects. They can act according to the same cycle when they work in the role of integrators. They need the same communication and interactive skills as business analysts.

3.5. Limits of the Analogy

The selected analogies described above just indicate for potential use of achievements form business analysis and systems modelling for interdisciplinary studies. In each practical case, one can make their own elaboration, adaptation and creative extension. The strength of such source of knowledge is the best practice from working on complex business analysis for non-trivial cases which was validated and collected by practitioners.

However, although business analysts often work on the edge of different areas, including domain, business and systems, and they must communicate with diversified stakeholders, business analysis does not contain typical interdisciplinary knowledge which can be found in literature of interdisciplinary studies. Thus, to become interdisciplinarian, one should also get acquainted with the findings described in theory of disciplinary studies as well as several integrative studies (e.g. these described in section 2.)

It is worth to mention possible bias of the author, whose background is in software engineering and business analysis with a few trials to integrate it with concepts from other disciplines including usability and psychological perspective, legal regulations, social aspects and creativity. It is a known fact in interdisciplinary studies, that the approaches we know, seem more intuitive for us, we understand and like them, we value them more than others and we can find more connections and applications for them. In general, one can say that our experience has impact on our perception of usefulness of interdisciplinary methods. As a consequence, participants with different disciplinary background might have different preferences regarding methods and approaches. This is the typical phenomena which makes a problem to overcome in interdisciplinary studied. It has been identified and described in details during the studies on boundary objects (see section 2.3). Thus, in order to make interpretation of these findings more objective, we must admit that, while being quite confident that it works, it is likely that it will not be intuitive and easy for everyone and that other approaches might work as well.

3.6. Implications for Practical Applications

Since we keep in mind pragmatic perspective of IT projects, it is time to raise the question about implication of these finding for practical application.

First of all, it is a good news for business analysts who would like to become interdisciplinarians. The results show that they have solid foundations for participation in interdisciplinary projects. The results also suggest direction for further education and training possibilities regarding specific interdisciplinary issues. They show examples of analogy, its limitations and style of searching for more analogies. Real business analysts like challenges. They are used to learning throughout their professional life. Thus, the extension of their competencies towards interdisciplinarians can be treated as adventure which can enhance their business analysis activities as well as show new perspectives of work as interdisciplinarians.

In case of individuals who do not have background in business analysis, they can use selected elements of business analysis in order to learn and train their interdisciplinary skills. Becoming professional business analyst takes quite a lot of time and requires practical experience in this profession, so this might not be the option for every candidate for interdisciplinarian. However, learning selected methods, both from the area of business analysis and interdisciplinary study, can be rewarded quite quickly in improved performance of interdisciplinary studies. Those who continue learning in a comfortable step-by-step manner for a longer time, are on a good way to becoming more-and-more mature interdisciplinarians.

The next challenge is related to supporting education of interdisciplinarians. Would a kind of body of knowledge of interdisciplinary studies help? How it should be structured? How it could take into account different starting points, different levels of advancement and variants of training? None is born interdisciplinarian. Everyone can become interdisciplinarian with systematic education and training. The way of supporting the education of interdisciplinarians in order to make it more effective, efficient and adapted to given cases, remains for further research.

4. Conclusions

The research described in this paper had the following goals:

1/ to contribute to better understanding of interdisciplinary approaches to IT projects by sharing review of interdisciplinary approaches, dimensions and lessons learned from research on interdisciplinary IT projects for public administration,

2/ to elaborate on analogy between interdisciplinarians and business analysts in IT.

Two kinds of interdisciplinary studies have been applied:

- *common ground* – identification of dimensions of approaches to interdisciplinary studies with the purpose of facilitating analysis, comparison and integrated use of different approaches,

- *analogy* between interdisciplinarians and business analysts.

The review of approaches to interdisciplinary studies contains: intuitive integration of knowledge, boundary objects, conceptual meta-modelling, multi-paradigm modelling, integration of knowledge in integrative studies and research on interdisciplinary teams from the organisational perspective. Each of them has a different context of discovery and makes a unique contribution to general knowledge about interdisciplinary approaches.

The dimensions include: individual vs. social aspects, intuitive vs. formal approaches, different kinds of knowledge in interdisciplinary studies (expert-, shared-, interface- and meta-knowledge), execution vs. management activities, and several tasks including integration, auxiliary tasks and tasks directly related to solution delivery. These dimensions allow not only for capturing diversity of approaches in a big picture, but also might be useful for selection of proper means in specific case of interdisciplinary studies.

The case description, approaches and dimensions show complexity and diversity of real interdisciplinary projects. And this was a starting point for the analogy between interdisciplinarians and business analysts. The following issues were discussed: focus on interdisciplinarian who can select proper methods to problem at hand (instead of proposing yet another method); use of the best practice from business analysis planning and monitoring for interdisciplinary planning and monitoring of specific interdisciplinary projects; analogy between conceptual modelling of one consistent system with several diagrams which are related to perspectives of viewing the system and conceptual description of discipline knowledge and integrating them in one consistent documentation of the project, and similarity between interdisciplinarians and business analysts regarding different roles (including the role of integrator) in interdisciplinary projects together with their skills. These are just examples of analogies and useful concepts which can be transferred from business analysis to formation of interdisciplinarians in the similar style.

The following implications result from the research. Business analysts are well prepared for extending their training towards becoming interdisciplinarians. Individuals without background in business analysis can use selected methods of business analysis for development of the competencies in their interdisciplinary education. The question for further research is how to effectively support education and training of interdisciplinarians.

5. References

Augsburg T. (2006) Becoming interdisciplinary. An introduction to interdisciplinary studies, Kendall Hunt Publishing.

Bakalis L. D., Folmer E., Bosch J. (2004) Position Statement, *Proceedings of the Workshop: Identifying Gaps Between HCI, Software Engineering, and Design, and Boundary Objects to Bridge Them* at CHI Conference.

Bobkowska, A. (2014). Zagadnienia w interdyscyplinarnym podejściu do wytwarzania systemów dla administracji publicznej. (Issues in interdisciplinary approach to system development in public administration), Roczniki Kolegium Analiz Ekonomicznych, Nr 33, ISSN 1232-4671.

Bobkowska A. (2015) Przegląd mechanizmów integracji wiedzy w projektach interdyscyplinarnych (Review of integrative mechanisms in interdisciplinary projects), Roczniki Kolegium Analiz Ekonomicznych, Nr 38, ISSN 1232-4671.

IBM (2006) IBM Rational Unified Process Specification, version 7.0.1.

International Institute of Business Analysis (2015) A Guide to Business Analysis Body of Knowledge (BABOK Guide), version 3.0.

Engelbart D. C. (1992) Toward High-Performance Organizations: A Strategic Role for Groupware, *Proceedings of the GroupWare '92 Conference*, Morgan Kaufmann Publishers, (available at www.bootstrap.org).

Hardebolle C., Boulanger F. (2009) Exploring Multi-Paradigm Modeling Techniques, SIMULATION, Vol. 85, Issue 11/12.

Kelly S., Tolvanen J-P. (2008) Domain-Specific Modeling: Enabling Full Code Generation. Wiley-IEEE Computer Society Pr.

Lewicki P., Hill T., Czyzewska M. (1992) Nonconscious acquisition of information. American Psychologist, Vol 47(6), 796-801.

Object Management Group (2017) Unified Modeling Language (UML), version 2.5.1

Parker G. M. (2002) Cross-Functional Teams: Working With Allies, Enemies, and Other Strangers, Jossey-Bass Inc Pub.

Repko A. F. (2008) Interdisciplinary Research. Process and Theory., SAGE Publications.

Language and Mind: A Complex Dynamic Relation

Dušica Filipović Đurđević
Laboratory for Experimental Psychology & Department of Psychology
University of Belgrade – Faculty of Philosophy
dusica.djurdjevic@f.bg.ac.rs

Abstract

This paper describes a view of language as a natural system that evolved by self-organizing to simultaneously fit the goal of successful communication and the constraints of the human system for storing and processing information. I will present relevant research findings to illustrate the immense level of attunement between natural language and human mind. I will also compare natural language and programming language to demonstrate multiple similarities, but to also pinpoint the crucial differences. I will argue that natural languages are tailored to fit communication between two interlocutors that share common experience, using the cues to evoke intended state within the interlocutor, whereas programming languages are tailored to instruct the machines using closed system of keywords and rules. Finally, I will layout proposals for possible future directions in making programming more human-friendly.

1. Introduction

Frequently, our cultural products are revealing of our important biological features. For example, the hight of the step, or a chair is determined by the average hight of the human lower leg; the size of the keyboard keys is determined by the average size of the human fingertip etc. Therefore, by studying the features of the cultural products, we would be able to learn something of the species that created them.

When perceived as a natural system, language is seen as a cultural product that evolved in adapting the communicative goals to the constraints of the human mind (Beckner et al., 2009; Christiansen & Chater, 2008; Gibson et al., 2019). In this process, an open self-organizing system has evolved as a structure that mirrors the human mind. Within psychology, such state is seen as an opportunity to understand cognitive system, i.e. to use the language as a window into human mind. In the earliest days of Experimental Psychology, some researchers were even as extreme as to claim that the only way to analyse higher mental functions if by understanding its products (Wundt, 1900).

2. Language as a complex adaptive system

To illustrate this view, we will demonstrate the sensitivity of language processing on the information load of the language input, the importance of all disposable channels of experiencing the world as humans in language use, as well as the adaptivity potential of the processing system to contextual challenges and the resistance of processing system to such challenges.

2.1. Information based language processing

Cognitive system is highly sensitive to the probabilistic structure of the environment and consequently to language as a highly structured communication system. Natural language processing is not only attuned to the individual probabilities of linguistic events, but also to fine grained relations of language systems nested within other language systems. We will show the interplay of the cognitive system and the complex informational structure of natural language that is built throughout the extensive experience with language and the environment. The rich repertoire of measures developed within the framework of Information Theory will be applied to describe complexities within language, and a repertoire of behavioural measures developed within the framework of Experimental Psychology to describe the human language behavior.

One of the basic Information Theory measures is the Surprisal, which is calculated as the negative logarithm of the probability of the given event to quantify the amount of information conveyed by the event in question. Frequent (i.e. highly probable) events are predictable and hence not highly informative. On the other hand, rare (i.e. improbable) events are unexpected, hence convey information and impose high load on the processing system. For example, when processing isolated words in a visual lexical decision task (where participants read orthographic strings and perform button-press to indicate whether they represent words), word recognition latencies are influenced by the Surprisal derived from word probability, which is in turn based on the surface frequency of the word (how many times the word in question occurs in a linguistic corpus). This effect represents one of the benchmarks in Psycholinguistics and was one of the first findings to demonstrate the sensitivity of human processing to probabilities in language. However, the Surprisal can be applied to demonstrate much more subtle sensitivities of the cognitive system to the complex structure of language. For example, it can be applied to inflected word forms to describe their morpho-syntactic complexity. In Serbian, each adjective can take different inflected forms to denote grammatical gender (masculine, feminine, neuter), grammatical number (singular, plural), and case (nominative, genitive, dative etc.). These grammatical features enable us to unravel the precise syntactic role the given inflected form serves in the sentence ("who is doing what to whom"). However, most of the inflected forms are grammatically ambiguous and can denote multiple syntactic roles. For example, inflected form *lepim* of the adjective *lep* (beautiful) can denote twelve different combinations of gender, number, and case (e.g. masculine plural dative, masculine instrumental singular, feminine instrumental plural, etc.). In our studies, we described each inflected form by the Surprisal that is calculated from the average frequency per syntactic function/meaning, relative to all other inflected forms. By doing so, we obtained the Surprisal that is informed not only by the probability of the surface form of the word, but also by its syntactic potential (the complexity of the roles it can serve in a sentence), in relation to the whole set of its possible inflected form (the morphological system it belongs to). This measure was successful in predicting processing latencies in a lexical decision task with adjectives (Filipović & Kostić, 2003; 2004), and even more successful with nouns (Kostić, 1991; 1995; Kostić et al., 2003). Such findings spoke in favour of the sensitivity of the cognitive system to the probabilities informed by the deeper structure of language.

A demonstration of the even more deep atonement between language and cognition came from the studies demonstrating that human language processing is sensitive not only to the overall morpho-syntactic structure, but also to the relation between the morpho-syntactic complexity of the individual lemma (a full set of the inflected forms of the given stem) and the class to which it belongs. To illustrate, each lemma can be described by the probability distribution of its inflected forms (so called inflected paradigm, e.g. the paradigm of the lemma *sauna* consists of six forms: *sauna*, *saune*, *sauni*, *saunu*, *saunom*, *saunama*). At the same time, it can be described by the probability distribution of the suffixes that are used to create inflected forms within a group of words that are inflected in the same way (so called inflected class, e.g. feminine nouns: *-a*, *-e*, *-i*, *-u*, *-om*, *-ama*). When we compare the two distributions pairwise for different lemmas, we find a range of patterns – for some lemmas the two distributions are almost identical, whereas for others they differ significantly. This divergence can be quantified using the Information Theory measure of Relative Entropy (Kullback-Leibler Divergence), and our studies revealed that it also predicted word recognition latencies – the larger the divergence, the more effortful the processing was (Filipović Đurđević & Gatarić, 2018; Filipović Đurđević & Milin, 2019; Milin et al., 2009). This indicated that human processing system builds expectations based on entrenchment of the linguistic units within the complex hierarchy of relations built from the exposure to language.

A similar Information Theory measure is Entropy, which quantifies the uncertainty within a random variable with multiple discrete outcomes. It is affected both by the number of outcomes (the more outcomes, the higher the uncertainty) and the balance of the outcomes' probabilities (the more balanced the probabilities, the higher the uncertainty). We investigated the phenomenon of polysemy (a form of lexical ambiguity) to demonstrate the sensitivity of cognitive system to both sources of uncertainty. Polysemous words are those that have multiple related senses (e.g. *paper as material*, *wrapping paper*, *scientific paper*, etc.). Psycholinguistic studies revealed that vast majority of words are polysemous (Rodd et al., 2002). Additionally, it is shown that the number of related senses facilitated word recognition time (Filipović Đurđević & Kostić, 2008). Therefore, not only that polysemy is pervasive in language, but it also aids the process of word recognition, thus suggesting that cognitive system uses words as multifunctional units that can be recycled and adapted to the current communicative need. Moreover, word recognition is also facilitated with the balance of probabilities of individual senses. For example, there are words such as *horn*, which can be used equally frequently to denote a part of the animal, and a sound device, whereas some other words, such as *shell,* tend to be dominantly used in one of their senses (animal), and only occasionally in other senses (ammunition). The imbalance of probabilities can be separately quantified using the Information Theory of Redundancy. Our research demonstrated that in addition to the number of senses, recognition latencies of polysemous words are also affected by Redundancy, with more balanced probabilities leading to faster recognition, and that the two sources of uncertainty combine in a single Entropy measure (Filipović Đurđević et al., 2009; Filipović Đurđević & Kostić, 2023; Mišić & Filipović Đurđević, 2022a; 2022b; but see Filipović Đurđević, 2019 for a different effect on homonymy).

If we place the relation between the maximum uncertainty allowed by the given number of polysemous senses (Maximum Entropy calculated as the log number of senses) and the observed Entropy of sense probabilities under the research scrutiny, we observe several important points. Firstly, the observed Entropy generally follows the logarithmic trend of the Maximum Entropy. Secondly, there is significant variation in the observed Entropy within a group of words with the same number of senses. Finally, we note that the observed variation is not unlimited but follows a narrow range (approximately around the levels of 80% of Maximum Entropy; Filipović Đurđević & Kostić, 2017). Therefore, it appears that there exists a "sweet spot" of uncertainty in language, and we suggest that it represents a compromise between informativity of the communication system and its learnability by humans. To strengthen such a claim, we conducted a study in which we ran a simulation based on error-driven learning of polysemous words represented via their co-occurrence patterns in language (distributional semantics). We demonstrated that measures derived from association weights that emerged in the simulation (and are related to polysemy indices) were successful in predicting word recognition latencies (Filipović Đurđević & Kostić, 2023). By doing so, we demonstrated that complex linguistic relations can be captured via simple learning rules.

2.2. Embodied language processing

In addition to relying on the information structure of the input, human cognitive system uses multiple additional channels of experience with the environment. Here, we will focus on sensorimotor experience, emotions and motivation.

The Embodied cognition framework posits that the systems for storing and manipulating information (memory) use the same resources as the systems involved with interacting with the world around us, namely perception (sensation) and action. According to this view, activating a representation in memory is heavily linked to reenactment of the same processes that were involved in gaining the experiences leading to the representation in question (e.g. thinking about a cat reenacts the sound it makes, the colour, the shape, the tenderness of its fur, the hand movements involved in handling the cat etc.). Moreover, it asserts that such knowledge is also involved in language processing of the word *cat*. These claims were tested by collecting sensorimotor norms, i.e. by providing speakers with a word and asking them to rate the level in which they see, smell, taste, hear, etc. the object denoted by the word. Based on the collected ratings, a single measure of perceptual richness (but also sensorimotor richness) can be derived. The research shows that words that denote objects with higher perceptual richness are recognised faster and memorised more accurately (Božić & Filipović Đurđević, 2025; Connell & Lynott, 2012; Filipović Đurđević & Živanović, 2011; Filipović Đurđević et al., 2016; Lynott et al., 2019; Popović Stijačić & Filipović Đurđević, 2015; 2018). Moreover, this applies not only to the typical speakers, but also to the speakers with Mild Cognitive Decline and First Episode Psychosis (Filipović Đurđević, et al., 2024; 2025).

In addition to sensorimotor experiences, human language processing heavily relies on emotional and motivational experience with the objects denoted by the words. Multiple dimensions of such experience have been proposed, but the most frequent ones are Emotional Valence, which captures the difference between unpleasant and pleasant items (e.g. *war* vs. *flower*), and Arousal, which captures the difference between objects that do not provoke our reactions as opposed to objects that invite us to act, or motivate us (e.g. *sunset* vs. *slide*). Although precise nature of the effect is still debated, the evidence speaks clearly in favour of the relevance of emotional/motivational dimensions on language processing and cognition (Gorišek, et al., 2024; Kousta et al., 2009; 2011).

2.3. Adaptive language processing

Finally, we will demonstrate how resistant language processing is to different challenges posed by the environment, and how quickly the processing system adapts to keep the communication successful.

For example, if we look carefully into the relation between Emotional Valence and Arousal, we will find a U-shaped curve indicating that words that elicit negative emotions and words denoting positive emotions are highly arousing, whereas neutral words leave us in an unaroused, relaxed state (Kuperman et al., 2014; Warriner et al., 2013; 2017). However, the arousing potential of words seems to be dependent on the wider environmental context. For example, it was documented that the signature U-shaped curve, as documented before the onset of COVID-19, started flattening during the pandemics, and became almost linear during the course of time. Although the Emotional Valence ratings did not change, their arousing potential changed in such a way to denote that negative words elicited higher arousal, whereas the arousing potential of positive words decreased, thus turning the U-shaped curve into the linear relation (Popović Stijačić et al., 2023), This finding revealed how the interaction between linguistic stimuli and cognitive processing was changed by the wider environmental context without noticeable effects on language communication.

Human language processing is also adaptable to local context, as elicited by the task conditions. This was observed in a study of visual word recognition during switching of the alphabets. The demonstration was carried out in Serbian, a language that relies on two writing systems – Cyrillic and Roman, with Serbian speakers being fluent in both alphabets and adjusted to frequent alphabet switching across discourse (e.g. reading a book in Cyrillic and turning to computer to check email in Roman alphabet). Code-switching is facilitated by language control, which is typically studied in bilingual language switching. In our study, we wanted to demonstrate that it also applied to within-language alphabet switching. To do so, we presented participants with two experimental blocks with visual lexical decision task. In the first block, all the stimuli (including consent form, instructions, practice trials and the main trials) were presented in a single alphabet, thus leading the participants to adapt to a single alphabet. However, in the second block, stimuli were randomly presented either in Cyrillic or Roman, and the task was to denote if a letter string was a word in either alphabet (or a meaningless string – pseudoword). The task was particularly hard because some of the strings could be pronounced in both alphabets but had lexical meaning only in one. Such design enabled us to detect two types of language control. The effects of sustained control were visible due to the initial adaptation to the single language block, whereas transient language control was elicited by random alphabet switching in the second block. Therefore, on some trials, our participants needed to resolve dual conflicts – the change of the alphabet as compared to the initial single-alphabet block (global switch) and the change of the alphabet as compared to the alphabet of the previous trial in a mixed-alphabet block (local switch). In addition to confirming that both sustained and transient language control were applied, the study also revealed an exciting pattern of adaptation. The adaptation is indicated by a decrease in processing latencies as observed during experiment. In the mixed-alphabet block, we found evidence of two simultaneous adaptation processes. Although the participants were generally becoming faster during the course of the experiment due to practice effects, this acceleration was dramatically more pronounced for globally switched Cyrillic and locally switched Roman alphabet. This indicates that two separate streams of adaptation were at place for different alphabets governed by two different control processes. The effects of sustained control were being dynamically adaptive for Cyrillic, whereas the effects of transient control were being dynamically adaptive for Roman alphabet. In addition to illustrating the flexibility of human processing system, this finding also reveals its amazing resistance to highly challenging conditions.

Above all, the uttering of the native language is subjectively effortless to typical speakers, straightforward to such an extent that it took thousands of years of civilisation to face the complexity of the language processing. For example, in early years chess playing was considered to be a hard problem for the machine, whereas early intuition was that teaching a machine to use natural language would be the easy task. These early intuitions were driven by our introspection about problems that are hard for the human mind (but not the machine, as will be demonstrated by the end of the twentieth century). Human language usage is not only unchallenging, but it also allows humans to play with language (e.g. in jokes that rely on linguistic twists, puns, etc.), or even speaking backwards, as documented in some case studies (Preković et al., 2016).

3. The relation of programming language and natural language

Now we turn to programming language and try to investigate how it relates to natural language. We have demonstrated that human language processing is relying on multiple systems of representations, but we should also keep in mind that it also relies on multiple modalities for communication: we use auditory channel when speaking, but also visual for gesturing and the facial expressions accompanying the speech. Multimodality is crucial for successful transmission in a noisy environment which frequently accompanies communication. This process is also critically dependent on the information structure of language, and the high level of redundancy.

We will start the analyses of programming language by placing it under the scrutiny of the proposed linguistic universals, or design features that were applied in attesting communication systems (Hocket, 1958; 1959; 1960; 1963). For purposes of illustration, we will focus only on a subsample of those features, such as semanticity (language symbols have meanings, i.e. refer to something), arbitrariness (symbols do not resemble the referents), discreteness (language units are discrete – their referents may indicate a continuum of change, but the change in the symbols is discrete, e.g. morning vs. dawn), duality of patterning (units without meaning combine into meaningful units), productivity (lower-level units combine to create novel meanings), and displacement (language can refer to events occurring at different time and place compared to the time and place of the speech act). These criteria have been successfully applied to demonstrate that various communication systems used by some animal species cannot be considered a language. For example, the famous dance of the bees (tail wiggling frequency denotes the distance, and the direction points to the location) fails the test of arbitrariness, discreteness, and duality of patterning. Similarly, the voicing of vervets (one sound denotes predator from the ground, different sound denotes predator from the sky) fails the test of duality of patterning, productivity and displacement, etc. The question is if we could attest whether programming language could be considered as a language following the linguistic universals criterion. In doing so, we quickly notice that programming language in fact does follow all the listed design features, and even some that are included later (Chomsky, 2010; Hauzer et al., 2002). For example, recursion is a language feature that denotes the existence of a rule that can be applied to the results of the application of the same rule (e.g. *It is the course taken by the student that had a supervisor that published the work that demonstrated that…*), and programming language tics this box, as well.

However, although programming language passes the language universals test, we will argue that there are crucial differences that separate it from the natural language. Some of the differences would become obvious if we tried to instruct somebody using the precise instructions, based on the premise that they would make no inferences, but only perform what they are explicitly instructed (as beautifully illustrated in "Exact instructions challenge" https://www.youtube.com/watch?v=cDA3_5982h8).

One of the fundamental differences between programming language and natural language is related to the description of the language system itself. Programming languages are easily described via the closed set of keywords and rules that apply to them. In fact, this is a typical way of introducing programming languages in the textbooks and manuals. However, when we try to apply the same principles to the natural language, we soon face difficulties. This is highly visible in grammar books, as they typically list a set of rules, followed by dozens of pages of the exceptions to the rule. A famous linguist used the phrase "All grammars leak" to describe the state of attempt to describe the natural language using the closed set of rules (Sapir, 1921, p. 38). Therefore, the natural language appears to be an open system, eluding the closed-set descriptions, unlike programming language.

One important feature of the programming language (and even crucial) is that the statements need to be either true or false. For a program to be functional, the compiler must be presented with either of the two values, with no middle ground. However, when it comes to natural language, we find the abundance of the middle ground cases. For example, in one of our studies we asked participants to rate the acceptability of the presented statements (the task was to estimate if they would expect to encounter such a statement in their language). Although we found that for some statements the speakers were clear in categorising them either as acceptable or unacceptable, there was a number of statement categories for which they were not as decisive, thus rating the statements as 50% acceptable (or anywhere in the range between 25% and 75%; Diesing et al., 2009; Kolaković et al., 2022; Zec & Filipović Đurđević, 2017). Human language processing seems to operate despite this vagueness, whereas a compiler would report an error.

When it comes to number of units, the natural language and the programming language differ dramatically. It is estimated that an average twenty-year-old student uses approximately 42 000 of lemmas, 4 200 of multiword expressions, and 11 100 of word families (Brysbaert et al., 2016). At the same time, programming languages use 20-100 keywords (https://github.com/e3b0c442/keywords). Moreover, they are less combinatorial, thus producing flatter probability distributions, i.e. higher entropy (Febres & Jaffe, 2015; Febres et al., 2015). This indicates that programming languages use less symbols to convey more information, i.e. are less redundant.

Finally, we would like to draw attention to the intolerance of programming languages towards ambiguity. Programming languages use unambiguous symbols which clearly map onto the intended function. At the same time, as illustrated in section 2.1, ambiguous mappings are pervasive in natural languages and seem to reflect an important design feature rather than an accident. Human language processing can even benefit from the ambiguities in certain tasks or quickly adapt when they pose a challenge. Such advantage, nor such adaptation occur for programming languages.

4. Conclusions and future directions

Based on the presented evidence we can conclude that language processing and representation is influenced by the information structure of natural language system (information load, entropy, relative entropy, etc.), sensorimotor bodily experiences, but also affective and motivation. We also demonstrated that processing of natural language is highly adaptive, both to local and global context, and that language units (e.g. words) are used as multifunctional tools. Finally, the natural language is structured to learnable and adaptively used by human cognitive system. Therefore, in order to understand the structure and function of the natural language system, we must take into account and explore the processes that underlie its representation, processing, and production (as also suggested by Bybee (2010) in the famous sand dunes analogy – to understand the sand dunes, we must study not only their shape, but also the forces that create them).

We demonstrated that human languages are represented and processed via multiple channels in a manner that is highly adaptable to the local and global communicational context. They are also rich in redundancy which, in addition to the adaptive nature of human cognitive processing also contributes to resilience to noise. It is rich in ambiguous mappings but structured to be learnable by humans. On the other hand, programming languages are less flexible (e.g. either true or false), described by finite set of rules. Although they use fewer symbols, they also apply less combinatorics thus leading to overall higher entropy levels (less redundancy). They are intolerant to ambiguities and structured to dictate.

Although natural and programming languages share multiple features (semanticity, arbitrariness, discreteness, duality of patterning, productivity, displacement, recursiveness, etc.), the most relevant for the processability of programming languages in human mind is what lies outside of the intersection. The natural language serves the role of communication by using the cues to evoke intended state within the interlocutor. The key prerequisite for such a process to be successful, is a shared experience between the interlocutors. On the other hand, programming language serves the role of communication using the instructions to evoke the intended behaviour in the interlocutor who shares no experience with the instructor. Equating the two processes would imply that obtaining the product from a vending machine could be labelled as “chatting with the vending machine”.

The key root to the crucial differences between the natural and the programming languages should be attributed to different goals of the two language systems and the roles they serve. Human-to-human communication is based on shared experience using open but learnable coding system, whereas human-to-machine interaction is based on the instructions using a closed coding system of keywords and rules.

The conundrum of how to make the programming languages more accessible to humans will occupy cognitive and computer scientists alike. Following the conclusions laid out in this analysis, we identify two possible paths to finding the solution. One strategy would be based on constructing the adaptive machines equipped with more human-like experiences. Including the Large Language Models in the process of code writing and code reviewing would be the first obvious option. After all, Large Language Models encapsulate all the human experience as coded in natural language and reveal the power of the natural language as the model of the world (i.e. the aspect of the world relevant for humans). The other strategy would be to take the human constraints into consideration when building the programming languages. The inspiration for this approach can be harvested from the success story of adapting command panels to human attentional capacities during the twentieth century. For example, in the early days of flight industry, accidents were more frequent and typically caused by human error. The detailed analyses would reveal that the crucial information was not provided on time or was disguised by the abundance of other signals leading to human system processing overload. Careful experimental studies helped to pinpoint the precise limitation of the cognitive system and to design the control panels that would present the information in a way that would be beneficial. The future will tell if the same strategy can be applied to programming languages as well.

5. Acknowledgements

I owe a debt of gratitude to my collaborators, with whom I have shared the journey of exploring language as a natural system and its processing in human mind. I am also grateful to the organizers of PPIG2025 for their kind invitation and for the opportunity to present my keynote talk and to share my insights on language processing with the wonderful PPIG community.

6. References

Baayen, R. H., Milin, P., Filipović Đurđević, D., Hendrix, P., & Marelli, M. (2011). An amorphous model for morphological processing in visual comprehension based on naive discriminative learning. *Psychological Review, 118*(3), 438-481. http://psycnet.apa.org/journals/rev/118/3/438

Beckner, C., Ellis, N. C., Blythe, R., Holland, J., Bybee, J., Ke, J., Christiansen, M. H., Larsen-Freeman, D., Croft, W., Schoenemann, T., & Five Graces Group. (2009). Language is a complex adaptive system: Position paper. *Language Learning, 59*(Supplement 1), 1–26. https://doi.org/10.1111/j.1467-9922.2009.00533.x

Božić, S., & Filipović Đurđević, D. (2025, June 3). The neglected role of lexical ambiguity in embodied cognition models. https://doi.org/10.31219/osf.io/5gdae_v1

Bybee, J. (2010). *Language, usage and cognition*. Cambridge: Cambridge University Press

Chomsky N. (2010). Some simple evo devo theses: how true might they be for language?. In R.K. Larson, V. Deprez, H. Yamakido (Eds.) *The Evolution of Human Language* (pp. 45–62). Cambridge University Press. https://doi.org/10.1017/CBO9780511817755.003

Christiansen, M. H., & Chater, N. (2008). Language as shaped by the brain. *The Behavioral and Brain Sciences, 31*(5), 489–558. https://doi.org/10.1017/S0140525X08004998

Connell, L., & Lynott, D. (2012). Strength of perceptual experience predicts word processing performance better than concreteness or imageability. *Cognition, 125*, 452-465.

Diesing, M., Filipović Đurđević, D. & Zec, D. (2009). Clitic placement in Serbian: Corpus and experimental evidence. In S. Featherston and S. Winkler (Eds.) *The Fruits of Empirical Linguistics, Volume 2: Product* (pp. 59-73). Berlin, New York: Mouton de Gruyter.

Febres, G., & Jaffé, K. (2016), Calculating entropy at different scales among diverse communication systems. *Complexity, 21*, 330-353. https://doi.org/10.1002/cplx.21746

Febres, G., Jaffé, K., & Gershenson, C. (2015), Complexity measurement of natural and artificial languages. *Complexity, 20*, 25-48. https://doi.org/10.1002/cplx.21529

Filipović Đurđević, D. (2019). Balance of meaning probabilities in processing of Serbian homonymy. *Primenjena psiihologija, 12*(3), 283-304. DOI: https://doi.org/10.19090/pp.2019.3.283-304

Filipović Đurđević, D., Đurđević, Đ. i Kostić, A. (2009). Vector based semantic analysis reveals absence of competition among related senses. *Psihologija, 42*(1), 95-106. https://doi.org/10.2298/PSI0901095F

Filipović Đurđević, D. & Feldman, L. B. (2024). Do readers exert language control when switching alphabets within a language?. *Journal of memory and language, 139*. https://doi.org/10.1016/j.jml.2024.104546

Filipović Đurđević, D. & Gatarić, I. (2018). Simultaneous effects of inflectional paradigms and classes in processing of Serbian verbs. *Psihologija, 51*(3), 259–288. https://doi.org/10.2298/PSI170811015F

Filipović, D. i Kostić, A. (2003). Kognitivna obrada prideva. *Psihologija, 36*(3), 353-378.

Filipović, D. i Kostić, A. (2004). Kognitivni status roda prideva u srpskom jeziku. *Psihološka istraživanja , 14*, 125-168.

Filipović Đurđević, D., Đurđević, Đ. i Kostić, A. (2009). Vector based semantic analysis reveals absence of competition among related senses. *Psihologija, 42*(1), 95-106. https://doi.org/10.2298/PSI0901095F

Filipović Đurđević, D. i Kostić, A. (2008). The effect of polysemy on processing of Serbian nouns. *Psihologija, 41*(1), 69-86.

Filipović Đurđević, D., & Kostić, A. (2017). Number, Relative Frequency, Entropy, Redundancy, Familiarity, and Concreteness of Word Senses: Ratings for 150 Serbian Polysemous Nouns. In S. Halupka-Rešetar and S. Martínez-Ferreiro (Eds.) *Studies in Language and Mind 2* (pp.13-77). RS, Novi Sad: Filozofski fakultet u Novom Sadu. http://digitalna.ff.uns.ac.rs/sadrzaj/2017/978-86-6065-446-7

Filipović Đurđević, D., & Kostić, A. (2023). We probably sense sense probabilities. *Language, Cognition and Neuroscience, 38*(4), 471-498. https://doi.org/10.1080/23273798.2021.1909083

Filipović Đurđević, D., & Milin, P. (2019). Information and Learning in Processing Adjective Inflection. *Cortex, 116*, 209-227. https://doi.org/10.1016/j.cortex.2018.07.020

Filipović Đurđević, D., Milin, P., & Feldman, L. B. (2013). Bi-alphabetism: A window on phonological processing. *Psihologija, 46*(4), 419-436.

Filipović Đurđević, D., Popović Stijačić, M., & Karapandžić, J. (2016). A quest for sources of perceptual richness: Several candidates. In S. Halupka-Rešetar and S. Martínez-Ferreiro (Eds.) *Studies in Language and Mind* (pp. 187-238). RS, Novi Sad: Filozofski fakultet u Novom Sadu. http://digitalna.ff.uns.ac.rs/sadrzaj/2016/978-86-6065-359-0

Filipović Đurđević, D., Sekulić Sović, M., Erdeljac, V., Mimica, N., Ostojić, D., Kalinić, D., Vukojević, J., & Savić, A. (2024). Sensorimotor lexical norms as a window into changed cognitive functions. *International Word Processing Conference, Book of Abstracts.* July 4-6, 2024, Faculty of Philosophy, University of Belgrade, 36.

Filipović Đurđević, D., Sekulić Sović, M., Erdeljac, V., Mimica, N., Ostojić, D., Kalinić, D., Vukojević, J., & Savić, A. (2025). Sensory ratings and sensitivity to perceptual variables: a novel approach to evaluating semantic memory in Mild Cognitive Impairment. Psyhiatria Danubina, 37(2), 180-193. https://doi.org/10.24869/psyd.2025.180

Gibson, E., Futrell, R., Piantadosi, S. P., Dautriche, I., Mahowald, K., Bergen, L., & Levy, R. (2019). How efficiency shapes human language. *Trends in Cognitive Sciences, 23*(12), 1087. https://doi.org/10.1016/j.tics.2019.09.005

Gorišek, L., Filipović Đurđević, D., & Damnjanović, K. (2024, December 5). Foreign Language Effect on the Judgment of Bullshit: Interplay of Language, Emotions and Meaning. https://doi.org/10.31234/osf.io/2quzc

Hauser, M. D., Chomsky, N., & Fitch, W. T. (2002). The faculty of language: What is it, who has it, and how did it evolve? *Science, 298*(5598), 1569–1579. https://doi.org/10.1126/science.298.5598.1569

Hockett, C. F. (1958). *A course in modern linguistics*. New York: Macmillan.

Hockett, C. F. (1959). *Animal 'languages' and human language*. Human Biology, 31, 32–39.

Hockett, C. F. (1960). *The origin of speech*. Scientific American, 203, 88–111.

Hockett, C.F. (1963). The Problem of Universals in Language. In Joseph H. Greenberg (Ed.), *Universals of Language*, (pp. 1-22). The MIT Press.

https://github.com/e3b0c442/keywords (accessed September 8, 2025)

https://www.youtube.com/watch?v=cDA3_5982h8 (accessed September 8, 2025)

Kostić, A. (1991). Informational approach to processing inflected morphology: Standard data reconsidered. *Psychological Research. 53*(1): 62-70.

Kostić, A. (1995). Informational load constraints on processing inflected morphology. In L. B. Feldman Ed., *Morphological Aspects of Language Processing*. New Jersey. Lawrence Erlbaum, Inc., Publishers.

Kostić, A., Marković, T. i Baucal, A. (2003). Inflectional morphology and word meaning: orthogonal or co-implicative cognitive domains? U H. Baayen & R. Schreuder (Eds.): *Morphological Structure in Language Processing* (pp. 1-45). Mouton de Gruyter

Kolaković, Z., Jurkiewicz-Rohrbacher, E., Hansen, B., Filipović Đurđević, D. & Fritz, N. (2022). *Clitics in the wild: Empirical studies on the microvariation of the pronominal, reflexive and verbal clitics in Bosnian, Croatian and Serbian.* (Open Slavic Linguistics 7). Berlin: Language Science Press. DOI: 10.5281/zenodo.5792972, ISBN: 978-3-98554-032-7 (Print Hardcover), e-ISBN 978-3-96110-336-2, 461 S

Kousta, S.-T., Vigliocco, G., Vinson, D. P., Andrews, M., and Del Campo, E. (2011). The representation of abstract words: why emotion matters. *Journal of Experimental Psychology: General, 140*, 14–34. https://doi.org/10.1037/a0021446

Kousta, S.-T., Vinson, D. P., and Vigliocco, G. (2009). Emotion words, regardless of polarity, have a processing advantage over neutral words. *Cognition, 112*, 473–481. https://doi.org/10.1016/j.cognition.2009.06.007

Kuperman, V., Estes, Z., Brysbaert, M., & Warriner, A. B. (2014). Emotion and language: valence and arousal affect word recognition. *Journal of Experimental Psychology: General, 143*(3), 1065–1081. https://doi.org/10.1037/a0035669

Warriner, A.B., Kuperman, V. & Brysbaert, M. (2013). Norms of valence, arousal, and dominance for 13,915 English lemmas. *Behavior Research Methods, 45*, 1191–1207.

Lynott, D., Connell, L., Brysbaert, M., Brand, J., & Carney, J. (2019). The Lancaster Sensorimotor Norms: multidimensional measures of perceptual and action strength for 40,000 English words. *Behavior Research Methods, 52*(3). https://doi.org/10.3758/s13428-019-01316-z

Milin, P., Filipović Đurđević, D., & Moscoso del Prado Martìn, F. (2009). The simultaneous effects of inflectional paradigms and classes on lexical recognition: Evidence from Serbian. *Journal of Memory and Language, 60*(1), 50-64. https://doi.org/10.1016/j.jml.2008.08.007

Mišić, K., & Filipović Đurđević, D. (2022a). The influence of experimental context on lexical ambiguity effects in Serbian. *Teme, XLVI*(2), 577-596. https://doi.org/10.22190/TEME210418030M

Mišić, K., Filipović Đurđević, D. (2022b). Redesigning the Exploration of Semantic Dynamics – SSD Account in Light of Regression Design. *Quarterly Journal of Experimental Psychology, 75*(6), 1155–1170. https://doi.org/10.1177/17470218211048386PDF

Popović Stijačić, M., & Filipović Đurđević, D. (2015). Uspešnost reprodukcije u zavisnosti od broja čula kojima je moguće iskusiti pojam. *Primenjena psihologija, 8*(3), 335-352. https://doi.org/10.19090/pp.2015.3.335-352

Popović Stijačić, M. & Filipović Đurđević, D. (2018). Perceptual Strength Norms For 2,100 Serbian Nouns. *International Meeting of the Psychonomic Society Abstract Book*, Amsterdam – Netherlands, 10-12 May 2018, 81.

Popović Stijačić, M., & Filipović Đurđević, D. (2022). Perceptual richness and its role in free and cued recall. *Primenjena Psihologija, 15*, 255-381. https://doi.org/10.19090/pp.v15i3.2400

Popović Stijačić, M., Mišić, K., & Filipović Đurđević, D. (2023). Flattening the curve: COVID-19 induced a decrease in arousal for positive and an increase in arousal for negative words. *Applied Psycholinguistics*, 1-21. https://doi.org/10.1017/S0142716423000425

Prekovic*, S., Filipović Đurđević*, D., Csifcsak, G., Šveljo, O., Stojković, O., Janković, M., Koprivšek, K., Covill, L.E., Lučić, M., Van den Broeck, T., Helsen, C., Ceroni, F., Claessens, F., & Newbury, D. (2016). Multidisciplinary investigation links backward-speech trait and working memory through genetic mutation. *Scientific Reports, 6*, 20369; doi: 10.1038/srep20369. (* denotes equal contribution) http://www.nature.com/articles/srep20369

Rodd, J. M., Gaskell, M. G., & Marslen-Wilson, W. D. (2002). Making sense of semantic ambiguity: Semantic competition in lexical access. *Journal of Memory and Language, 46*, 245–266. https://doi.org/10.1006/jmla.2001.2810

Sapir, E. (1921). *Language: an introduction to the study of speech*. University of California Libraries.

Warriner, A. B., Shore, D. I., Schmidt, L. A., Imbault, C. L., & Kuperman, V. (2017). Sliding into happiness: A new tool for measuring affective responses to words. *Canadian Journal of Experimental Psychology = Revue Canadienne de Psychologie Experimentale, 71*(1), 71. https://doi.org/10.1037/cep0000112

Wundt, W. (1900). *Völkerpsychologie, Volume 1*.

Zec, D., & Filipović Đurđević, D. (2017). The Role of Prosody in Clitic Placement. In V. Gribanova and S. Shih (Eds.) *The Morphosyntax-Phonology Connection: Locality and Directionality at the Interface* (pp. 171-196). Oxford, UK: Oxford University Press.

Živanović, J. & Filipović Đurđević, D. (2011). On advantage of seeing TEXT and hearing SPEECH. *Psihologija, 44*(1), 61-70.

Purpose, Transparency and Challenge: Self-taught programming in adolescence

Jan Dittrich
University of Siegen
jan.dittrich@uni-siegen.de

Felienne Hermans
Vrije Universiteit Amsterdam
f.f.j.hermans@vu.nl

Abstract
Many programmers will agree that programming is a unique activity, which, although it has been compared to writing, gardening and painting, is unlike other things humans do. But if it is unlike other activities, how do people learn to be a programmer and what affected them in this process?

We start at the beginning of people's programming careers, when they first start to program and interview 10 developers about their experiences. Based on the interviews, we suggest three core themes that affected the programmer's early learning experiences: Purpose of activity, transparency of technology and meaningful challenges.

1. Introduction
A large part of programmers in the workforce today learned programming in a *'self-taught'* way, having started to program at a young age before the internet was widespread. Learning programming not in school, but with help of instructional media is still common: The 2024 StackOverflow survey allowed to report several ways of learning programming. Over 82% learn from "other online resources" while the percentage of people having learned programming in school is about as high as the use of physical media like books.[1] The use of physical media is even higher for programmers between 35-44(59%) and between 45-54(68%), indicating that older programmers might be even more likely to be self-taught.

We are interested to learn more about the phenomenon of self-teaching as it seems to be an unexplored topic in programming education. We wonder: How did childhood experiences of programming come to be? What made it possible for teenagers to start their professional development at such a young age? And how do the views of self-taught programmers shape their theories and practices today in their careers in technology? That is what this paper sets out to explore by conducting semi-structured interviews with 10 self-taught programmers who learned programming in their teenage years.

Using thematic analysis, we find three main themes in our interview data. Participants seem to have been driven by a clear **Purpose** in programming; each had a self-selected goal in mind that they wanted to reach; often making a game, sometimes making a website. This purpose was not only technical but also embedded in a community; peers or parents enabled the love for programming, in material ways, with access to computers or computer books and magazines, or immaterial by providing encouragement and support. Many participants also mention being bored, having nothing else to do, which might have contributed to the need of finding a purpose.

A second theme which enabled programming in childhoods is **Transparency**: in the eighties, programming was much more easy to observe than it is now; BASIC games shipped with their source, and in the nineties, web pages' markup was often easy to read and to understand. We recognize that in today's digital world, the feeling that programming is a thing you can learn, that the codes that they show will be one day understandable to you, might seem much further away than 3 or 4 decades ago, despite an abundance of learning materials.

The final theme is the **Challenge** of doing something difficult. Participants recognized that programming would be challenging, like a puzzle or like running a marathon, and they specifically sought out that challenge. Our participants saw that trying, and then succeeding in something hard, was an interesting

[1] https://survey.stackoverflow.co/2024/developer-profile#2-learning-to-code

reward in itself. Important part of this challenge was that is was self-imposed, sought out and not forced by someone else.

2. Background and Related Work

A seminal work on the role of computers and programming in childhood is Sherry Turkle's *The Second Self* (Turkle, 2005), discussing computers and programming as a tool as well as a way to think about questions of life, death and what it means to think. Seymour Papert's *Mindstorms* (Papert, 1980) was published five years prior and details how and why children should learn to program using the language LOGO which allows to draw by code. A similar focus on drawing was discussed by Goldberg by using smalltalk (Goldberg & Kay, 1977). This tradition of using smalltalk in programming education continues until today with smalltalk-inspired languages like Scratch. Notable about the culture surrounding these tools and their makers is a focus on child centered education, outside of traditional classroom teaching. Papert writes in his book "Mindstorms": "I believe that working with differentials [a gear train used in cars] did more for my mathematical development than anything I was taught in elementary school." Teachers thus only have a small role to play in his educational vision. As such, our participants learning outside of school fits the viewpoints of the day.

We are, however, less interested in the tools used for learning, these have been studied extensively in its earlier days (Solomon et al., 2020; Battista & Clements, 1986), see (Lockwood & Mooney, 2018) for a comprehensive overview of integration of Computational Thinking into K-12 education. In addition to programming in schools, we have also explored how programming in taught in out of school clubs in prior work (Aivaloglou & Hermans, 2019).

In this work, however, we want to explore learning of programming entirely outside of educational contexts involving teachers. We aim to find out more about what enabled learning as children for people who are adults today, how context, practices and values enabled their learning, and how that learning shaped their ideas about programming.

The fact that all our participants are use he/him pronouns is indicative of the time period that we study. Particularly relevant for the timeframe in which our participants learned programming is also the history of early home computers and their user communities. Here, the masculinization of computing is also discussed, for example in the marketing of the popular C64 home computer (Juul, 2024) and the shift towards gaming becoming more excluding towards girls and women (Kirkpatrick, 2017). Factors that lead to more participation of women in computer science programs was later researched by Fisher and Margolis (Margolis & Fisher, 2001).

3. Methods

To explore the experiences of childhood programming, we interviewed 10 people that taught themselves programming, which we recruited through (then) Twitter. We conducted semi-structured, online interviews with all participants. We subsequently analyzed their answers using thematic analysis (Braun & Clarke, 2019), to explore our overarching research theme of self-directed programming education in teenage years.

3.1. Interview Protocol

In each interview, we asked these questions:

1. Can you retell how you learned programming?
2. What is being self-taught?
3. Do you consider to still be a self-teacher, if you learn new things now?
4. Is there a difference between self-taught people and not self-taught people?
5. What is the effect on you now?

ID	Age	Occupation	Pronouns	Residence	Origin
P1	-	-	He/Him	-	-
P2	-	-	He/Him	-	-
P3	50-59	Freelance programmer	He/Him	Belgium	Belgium
P4	30-39	Freelance programmer	He/Him	Germany	Argentina
P5	50-59	Freelance programmer	He/Him	UK	UK
P6	30-39	Researcher	He/Him	UK	UK
P7	40-49	IT consultant/software engineer	He/Him	Netherlands	Netherlands
P8	40-49	Software Developer	He/Him	Netherlands	Netherlands
P9	30-37	Dev-Ops/System Engineer	He/Him	Netherlands	Netherlands
P10	20-29	Software developer	He/Him	Czech republic	Czech republic

Table 1 – Overview of personal details of the 10 participants

6. What is needed/required for self-teaching?

With Questions 1 to 3, we specifically explored the devices and tools that our participants used, whether their direct community (parents, family members, friends) provided help, or acted as role models, and how other media sources like books, magazines or video tutorials contributed to learning. Question 4 sought to understand how participants see *self-taughtness*, and how they imagined others to see it. With Question 5, we aimed to explore how their childhood experiences shaped their current thinking in their profession (which are often in IT or in adjacent occupations). It also sought to encourage participants to reflect on their experiences as related to peers with a different pathway into computing. Questions 6 was deliberately broad and open to allow referring to all people, media, free time and personal traits.

3.2. Data Analysis

All 10 interviews were automatically transcribed by the video conferencing software ZOOM, which we used to conduct the interviews. We also recorded audio, and used the audio where needed to correct and extend the automatic transcripts.[2]

We analyzed the improved transcripts by using thematic analysis (Braun & Clarke, 2019), identifying themes for short sections of each interview which we then discussed. Each of the two authors initially coded five interviews alone (P6 to P10), after which we compared notes and grouped the themes for those first five interviews into categories. From there we individually coded the remaining five interviews, but in the same room, so we could continuously communicate about the found themes. Through iterative refinement of the categories for the remaining interviews, we identified core themes that could be found across several participants.

3.3. Participants

Our participants all use he/him pronouns, come from different countries, but mainly from the EU, and hold a variety of jobs related to programming. Table 1 gives an overview of the participants and their backgrounds.

4. Results

The goal of this paper is to understand what constitutes being self-taught, and what enabled this between roughly 1980 and 1995. From the interviews we conducted, we constructed three large themes: Purpose, Transparency and Challenge.

Note that additions to quotes between round brackets are context added by the authors. In some cases we have also shortened quotes, indicated by . . . , and removed words like ‘ehm’. Our analysis resulted three main themes that we elaborate upon in the next sections.

[2]which was very often needed, the term ‘self-taught’ does not seem to be included in many datasets the ZOOM speech to text algorithm is trained on, and the non-native English of both interviewees and interviewers often tripped up the tool, too

4.1. Purpose

Participants all spoke about finding a *purpose* in programming, a thing that they felt both drawn to from within, and enabled to do by outside factors.

4.1.1. A thing to build: Games and Websites

Having something concrete in mind to (re)create contributed to participants having a purpose, and for most of our participants, what they wanted to create were games.

Rather than serving a singular simple aim that could be summarized of "*having fun while playing a game*", games also served as motivation for programming and as a topic to bond over with peers.

Our participants pointed out that using the computer for games was obvious, and further elaborated: "*This was what these little home computers were for, nobody did anything serious with them*" (P5). Similarly, P4 pointed out that the activities on the computer were "*playing games or doing programming*". The choice between these activities was even seen to be baked in the hardware: "*I mean, unless you shove a cartridge in there with some cool Konami game, you immediately get a prompt for programming or loading stuff from disk or tape[...]*" (P7)

Thus, it is not surprising that programming was often approached through wanting to create games: "*...of course you have to try to create games.*" (P3). P4 initially wondered "*How do people make these games?*" and P6 asked his father "*How do I make a computer game?*" continuing that "*... his answer was basically to give me a book from his bookshelf on BASIC.*". Two participants, P6 and P10 also mentioned the customization of games.

Games were not just a means for interaction with the computer alone. P8 says his MSX microcomputer club "*was more a games exchange.*" and P7 told that one purpose for exchanging with peers was to get new games: "*Basically, you bought a box of floppy disks and then you went to a friend that you copied whatever new stuff he had lying around.*". Thus, games also serve as means to build social ties.

Not all participants were interested in building games though. Younger participants like P2 also mentioned making websites, but in all cases the participants had a thing in mind they wanted to build.

4.1.2. Boredom

However, just the goals of making a game or a website might not have been enough to start programming. Games and websites might have kept children busy *playing* the games or *visiting* the websites rather than creating them.

Participants frequently described that interacting with computers was a welcome way to spend free time, and that at one point, all levels of a given game would have been completed.

P4 told us: "*So at night I didn't have anything to do, because I lived also far away from my friends*" and "*There was nothing more to do; either I was doing playing games or doing programming.*"

These feelings are very much in line with the experience of the first and second author of this paper. P9 contributed to this topic as well: "*I mean back ... we didn't have an Internet or anything, so there wasn't a whole lot to do.*" P10 says his self teaching was enabled by "*Various kinds of discoveries . . . And a lot of free time*".

P6 also described that, as a young teen, in those days, there weren't so many things you could do: "*You know with that age, what activities were available back then? If you're home at, let's say like 10 o'clock at night, and you're not out with your friends, then what have you got? Maybe like TV or reading a book. Or playing games with my brother.*"

P9 stressed that, because the computer was so limited, making games was needed: "*And mostly, I pretty quickly got bored with what the machine could do . . . So I pretty quickly figured out that there had to be a way to make machine do more ... so I took it from there and I started in a QBasic*" and added "*I want to emphasize the boredom part . . . That's the main motivator for almost anybody*".

Programming was the only way to create new games or programs when all the games one owned were played, and there was no distraction from Netflix or TikTok.

It is interesting to think about how learning this way is difficult in today's world where kids never reach 'the end of content'. P9 explicitly reflected on this topic: "*That which is increasingly absent in this modern world is boredom*".

4.1.3. Social Permission

A final part of purpose is that people around you allow, or even encourage you to do an activity: Participants talked about being part of a community, that both helped them to get started and to keep going. In other words, they experienced what we call 'social permission' to pursue their interests.

P1 said that, in addition to access to book and materials, access to "*people that will actually answer questions that you run into*" is most important for self-directed learning. He further mentioned that it is not only learning from others, it is also sharing the love for programming, stating he thinks that "*the availability of people actually excited for this work*" is what gets kids going. P3 confirmed this, says that "*friends at school were the main source of information.*"

Magazines played a role in creating small communities of peers. P5 remarked "*And we did read magazines. A lot of the magazines were actually really brilliant.*" P8 also mentioned magazines: "*MSX magazines that also had listings, which my neighbor also liked, so he typed in the listing, and I tried- I solved the bugs in those programs.*" Here, the media provides the code that he and his friend then collaborated on. P9 told that he started to learn alone, but "*As time went on, I got to know some other kids who were a couple years older, who were more experienced with programming and they basically taught me some other stuff.*" and P7 remarked that "*... there was classmate who's dad did some LOGO... And maybe we also had a machine running LOGO in school.*"

There were also mentions of working alone at least for some time: P9 said he did not have support, contrary to other people: "*I noticed some friends of mine had parents who would guide them ... That's what I missed. I basically had to rely on my own curiosity.*" P4 also described programming alone for a long time, and longing to find a group to learn together with: "*I could not find in my class somebody that that I could ask... because I was the one who knew about programming.*", eventually finding peers in his university career: "*Finally... I found the place where where I was among people at my level, so that I could improve.*" P6 told us: "*I don't think I knew anyone else that was doing it in primary school*", but that he found peers with similar interest in secondary school.

Most interaction with peers are about programming and computers, but descriptions about working together on code are rare. The direct interaction with code itself is mostly a solitary activity.

4.1.4. Parents

It was not just peers who made our participants feel that programming was a good activity to pick up. Parental support has played large role in enabling programming enthusiasm. In almost all interviews parents, most often fathers, play a large role in shaping the interest of participants in interacting with computers.

There were repeating patterns of how children got access to computers and how their parents interacted with their kids. First of all, there was direct support received from parents in the form of buying the computer needed for programming, which at the time were quite rare. Sometimes the computer was brought into the house for different reasons, for example for a parent's jobs (P6, P7, P9, P10), but often they were bought specifically for children to use or even to program on (P1, P3, P5). P5 expresses that the computers were often bought for children directly: "*I can't think of anyone whose computer was their dad's or their mom's. It was a child's thing.*" In addition to the computer itself, parents also provided educational materials like books or magazines.

Parents were usually computer users themselves. They did not teach or encourage programming directly, but they often had positive feelings about technology and computers. The situation seems to have been positioned at participants at a sweet spot between not being forced to do programming by overexcited parents, but also not being discouraged by parents who were very critical about computer use.

For example, P4 described his father working for the government overseeing "*digitizing of processes, and he may have seen the benefits of that ... But I think he saw this [digitization] coming and that is was important.*" P1 told about modifying an existing English language program to work in Dutch saying "*my dad was very concerned as well as proud.*" P2 stated: "*My dad though, he was more interested in technology [than P2's mom], he is an electrician*", P10 describes his father similarly: "*His work was technical, but he was not dealing with software, he was a hardware technician*".

P6 describes "*at one point I went to my dad and I said: "How do I make a computer game?" and his his answer was basically to give me a book from his bookshelf on BASIC.*" He goes on to describe how his father had bought the book for himself but never really explored it in depth.

4.2. Transparency

Many participants in our study indicated they started programming with a wish to recreate something they knew. Older participants (P6, P4, P3) mention playing games and wanting to build those, slightly younger participants like P2 mentioned making websites.

It seems to have been important to have access to the code behind running programs, and to change that code and learn from it, in two ways. Firstly, seeing a completed game or website clearly showed what learning to program could bring, helping to keep enthusiasm going ("It must be possible to do this"). Secondly, having access to the source code made it possible to get started by adapting programs rather than building new ones from scratch, a sort of Use-Modify-Create model *avant la lettre* (Franklin et al., 2020).

For most of our participants, 'succeeding at programming' did not mean reaching a mathematical definition of a solved problem—something like correctly reversing a list or finding a certain substring in a piece of text—but being able to successfully figure out a way of replicating parts of another program, like a realistically bouncing ball (P1) or rounded corners of resizable boxes on websites (P2). After that success, some participants would tinker with the solution to make it more uniquely theirs. This is a recognizable experience for the second author, who while in high school had together with a friend decided their goal was to program a rotating cube in BASIC. The goals was self-selected, and there was no clear ceiling, when the rotation was done, we added the challenge to make it interactive, to add texture and so forth. It was not finished, in our minds, when we graduated high school.

4.3. Challenge

The final large theme that we suggest based on the interviews is enjoying a challenge. Of course, this relates also to purpose, as doing a challenging thing is much more likely to give a sustained purpose than something that can be completed quickly and without effort.

4.3.1. Programming as puzzle solving

There can be many different reasons to be excited about learning programming: It can be driven by a concrete goal, like making a game or a website, but we also found that many of our participants were, at least in part, driven by learning something complex because it was intellectually rewarding.

P1 makes this distinction very concrete: "*I don't think I wanted to create as much as I wanted to understand what was going on. So it was this great, great puzzle*".

The reward was in the programming itself and not other factors: "*Even though it was challenging, it was a lot of fun, figuring things out on your own, when you succeed*"(P2). The accompanying frustration was seen as enjoyable, P2 continues: "*Most people could probably run a marathon just by being stubborn enough. But most people don't do that, because they know it's a lot of pain. They kinda need to enjoy that pain, I suppose*". P9 expresses a strong focus on solving problems: "*You really want to understand why there is a problem ... It's almost like a sort of Pavlovian reaction, like: computer gives problem, I need to fix it*".

Consequently, getting and using a permanent result was not relevant. P5 describes that he and his friends wrote games "*all the time*" while also saying that "*they were rubbish.*". That the games worked meant that the puzzle was solved: "*You play it for five minutes and then you... then the next night you do another one*" (P5). P4 similarly describes that "*It [building games] was just for me, it was much- interesting to make the games than to play the games.*". P10 mentions that he liked programming for its own sake, and not for external reasons: "*being able to program as a kid wasn't something that that gave me my social credit, which I think has changed in recent years*".

4.3.2. Exploration

Participants mentioned learning through exploration. Often, our participants find that this is not just the way they learned, but a requirement for being a programmer. P1 explained "*I do consider to [figuring things out] be a prerequisite for being a good programmer. Because if you can figure things out, you can program.*"

P3 stated that what drove his interest is "*Interesting things and explore those*", which meant "*Lots of experimentation and not following rules but experimenting*". This is tied to the feeling of hacking or breaking things, which can be attractive. P4: "*...we will basically try to break Windows [The Operating System], putting websites and folders in the desktop. ... So that's basically like like mashing stuff until it worked.*"

Learning by exploration is related to tinkerability and we doubted, whether we would put it together in one category. Ultimately, we decided to describe them as two separate aspects: Tinkerable artifacts are compatible with most school systems; even if you would have a teacher-led form of teaching, a teacher could show students, step by step, how to recreate games or websites and then have kids pick a program to replicate, adapt and change it.

Exploration, however is not so much about what you do as an activity, but about how you do it: From your own motivation, by finding just the right amount of information and then going back to work. This form of learning aligns more with Papert's way of constructionist teaching in which kids build something that has meaning to them by figuring out how to do that. This model of learning is not compatible with explanation by a teacher. After all, Papert said that "*by explaining something [to someone] you take away the ability for a child to discover it*" (Papert, 1980).

4.3.3. Obsession

Some of our participants named being 'obsessed' with the computer and not doing any other things, which fits modern day stereotypes about people interested in programming (Lewis, Anderson, & Yasuhara, 2016). P10 said: "*I was completely obsessed with programming, so I didn't need a reason to continue doing it. As opposed to anything else, it was just natural.*" P5 told: "*unless we were doing the exam coursework, we were writing games. All the time. All the time.*"

This focus can lead to neglecting other activities, as P8 described: "*My whole school effort went down, and I was only at the computer, when I was 12 or 13, I think.*" He also mentioned how special it feels to be programming: "*. . . and that's the thing I love about the computer, because you dive into it and you get emerged in the technology and you discover things. That's . . . yes, it's special*".

There are also related, less extreme attachments to programming. P4 stated that being interested is the only way for him to learn "*I cannot stay focused if I am not interested . . . A tight feedback loop and a problem is the only way I can stay seated and not move.*"

4.3.4. Solitary Activity

Despite the fact that some encouragement in the form of special permission seems to have been needed, this does not mean that the act of programming itself was always done in pairs of groups; the contrary seems to have been true: programming was often done alone, but talked about with friends or parents, or about showing them running programs, or talking about ideas for projects.

P6 for example mentioned not having people to program with: "*. . . I don't think I knew anyone else that was doing it in primary school. . . Only when I got to secondary school in that second year, so I would have been be 12 or 13.*"

Some participants even explicitly mention that programming with other people was not desirable for them. P10 says that "*the programming interest that . . . I had was very much a solitary activity, which it didn't have to be, but for me it just made the most sense.*". P3 states that "*You learn a lot from working in a team, I know. But for me it was never that interesting.*"

Several participants mentioned that they believe that there are certain types of people drawn to programming, even though these were different traits. P9 names being an introvert: "*I'm not the most outgoing type . . . so I spend a lot of time with the computer*", while P8 mentions creativity: "*I think I'm a creative person and I learned programming by- because I'm creative.*"

5. Discussion

5.1. Purpose

There were several sources of motivation and purpose that our participants drew from: The immediate motivation was usually to create games, and, for a younger participant, creating websites. These motivations were, however, enabled by several factors, including the social permission and support by parents and peers as well as having time to explore the use of computers.

Games were frequently referred to by our participants and also described as being as one of the two things that 80s microcomputers could be used for, the other activity being programming. These activities were not separate: Programming could create games, and enthusiasm for games could motivate to do more programming. Gaming was the factor that eased socialization around computing topics, and aside of games itself, young people also exchanged magazines which often included code listings to create games.

The connection of programming and computer games has a long history: Even when computers were still expensive expert tools at universities and government institutions, programmers created games for them. Later, the association of microcomputers with games was also the result of marketing these as both toys and educational tools to boys (Juul, 2024) Other creative uses might be less prominent, but also have a long history, for example using them as creative tool in computer generated art, for making music or for creating textile patterns.

The focus on programming and games as well as the social legitimization of these activities was only possible under the assumption that it was okay for kids to be bored, to have free time and to spend that time at computers: Boredom was mentioned as a major factor. This makes sense, as a lot of activities around learning programming need time and do not yield immediate results: Searching for a mistake in code can easily take hours and taking this time would be less likely if the life of our participants would have been more directed at either learning for school or spend in extracurricular activities away from their computer.

One thing to note is that purpose, contrary to the other two large themes, is not as strongly connected to programming. Teenagers do pick up music, reading, drawing or other activities with similar fervor. However, childhood programming forms a more direct pathway into an occupation, and thus young programmers are more likely to still work in the field of programming. As such they are bringing their purpose into their adult lives at a larger proportion then to people that used to like drawing or music as a kid. This might help us understand some of the culture of programming including a focus on doing hard things better (Hermans & Schlesinger, 2024).

5.2. Transparency

Today, the connection between games and code is not obvious: Games do not come with their source code and while creating games is still a possibility, it is not a common activity to do at ones own computer let alone smartphone or tablet. Often, such activities are actively prevented by the technology: Running custom code on your game console or on your iPhone is difficult and an exception. Websites today do still provide inspectable code, but since it is mostly machine generated rather than directly written by programmers, it is almost impossible to find out how the website works by inspecting the code.

This, however, was different for our participants: The explorability of how a game, or any other computer program works, was much stronger in the 80s and 90s: Programs often came as source code that could be read by users and later, website's source code could be inspected. Program code for games was shared in magazines and one participant mentioned that the creation of games was more fun than actually playing it (P5).

It is notable that participants learned from examples rather than mere trial-and-error: Engagement with coding could also be replicating code from a magazine or trying to replicate an example. Knowing that it was created with code that they had access to, had a twofold effect. Firstly, it meant that one could learn how another programmer archived a certain effect (the bouncing ball, the rounded corners of boxes...), and secondly, it was certain that somehow, with enough effort, one could replicate the effect.

The examples used were not designed to be used for education: That code of games or websites could be read and replicated was part of the medium itself. This does not exclude use structured learning materials, but made learning by self-selected examples possible. In particular, wanting to replicate something or getting some code running can provide a motivating challenge.

5.3. Challenge

There are many potential ways to engage with code, one of them being the challenge: Being strongly motivated by overcoming a difficult problem that, in the context of our research was usually self-imposed. It might feel very different, though, if a computer program that "should work" just does not: The computer can become an antagonist (Turkle, 2005, 102) and the artifacts seems to impose the challenge.

What participants programmed were usually copies or derivatives of already existing programs. The models from which to copy can be directly provided by listings in magazines and books: You could manually copy a game by typing the source code. However, examples can also be taken from other computing experiences, for example interesting looks of websites (P2) or existing games (P8: "*You saw in a game, and now you could do it yourself.*"). This is similar to honing one's drawing skills by creating drawings of fan-favorite characters. To work as an example to be copied or extended, programmers must assume that they actually are able to do this: A simple text adventure might be plausibly created by a single person, but a somewhat up-to-date-looking 3D action game is known to be the work of many people with different specializations and thus not a plausible thing to copy – though it might still serve as motivation to be, at one day, part of such a production.

At a first glance, Challenge might feel odds with Transparency, as the former focus on hard things, while the latter makes things more accessible. We however believe they strengthen each other. The transparency of code made it possible for budding programmers to see what they were up against, which looked challenging, but also doable.

Our analysis confirms several known phenomena: The supportive parents (Margolis & Fisher, 2001, 94), in our research particularly the fathers, providing both the hardware and parts of the legitimization of the hobby of programming, the solitary and often obsessive interaction with machines over longer times (Turkle, 2005, 191) and the motivation to find out "how it works" by reproducing it (Turkle, 2005, 74). All of these are male-associated stereotypes. Furthermore, all our participants used he/him pronouns, none she/her or they/them, with a majority coming from Europe and being typically between 30-50 years old, many of them fitting well in the cohort to whom (or their parents) microcomputers were marketed as an educational toy for boys. (Juul, 2024)

Thus, the question needs to be posed of what we think we can learn from our participants for the future of education and design of tools to be more inclusive. On one hand, we want to understand the past experience of being self-taught better. The idea of the self-taught programmer is a powerful concept that still does influence both education and legitimization of roles until today, so we hope to make sense of the current state of education and tools by our qualitative study. On the other hand, we think we take aspects from the learning experiences of our participants and find our more about both, what makes programming interesting and motivating while wanting to make it interesting and motivating. By interrogating these aspects, we would like to see how the powerful trope of the "self-taught-programmer" might help to understand what can be motivating about learning programming for a broader audience, too. This necessarily involves not to replicate the past, but take what we learned as a starting point for re-interpretation.

6. Implications for programming education in schools

As we mentioned in Section 4, children in our study were enabled in their learning by *social permission*, parents encouraged the learning of programming by providing books and computers (particularly since computers were an investment back in the day). However, there was a balance, as parents were not actively pushing programming, since they were not programmers and did not know programming themselves. Furthermore, contexts were created in which kids were interested, giving them a purpose.

How can our results inform educational practices and tools for modern programming education, especially in the context of schools rather than private bedrooms?

6.1. Purpose: Concrete Goals

What we see in these interviews is that once these kids had in mind a thing they wanted to create, and something, both in the online or offline world that they wanted to be part of giving them a purpose. With that purpose, their motivation remained extremely high even though they also describe getting stuck and feeling frustrated.

This has implications for programming education, since lessons typically do not contain a lot of *purpose creation*, or at least not in the first stages. Specifically in programming in a textual language, extensive instruction in the intricacies of a programming language syntax are needed to even create simple programs. Some students never reach a good enough understanding of programming details to create programs, a concept known as the *syntax barrier* (Denny, Luxton-Reilly, Tempero, & Hendrickx, 2011; Gilsing, Pelay, & Hermans, 2022).

Because of the complexity of languages, programs in education start small with the canonical ‘Hello World’ program. It takes a long time before learners exposed to examples of what would be possible with programming in the medium to long run. There are exceptions of course: Systems like SonicPi or Processing are focused on different goals (music and art) and as such do take learners along in the path of possibilities. However, they still require complex syntax that distracts. Teaching methods such Use-Modify-Create model (Franklin et al., 2020) aim to start with a concrete goal which children adapt, but as the programs must be understandable for learners, they remain simple in many cases.

While not all kids might be (so strongly) motivated by purpose and goals, one could imagine that more children would be if programming education placed more emphasis would be placed on what programming is *for*.

6.2. Transparency: Modern tools lack it

In the interviews, we find that it was easy to inspect and change the code that creates the software. The programming languages were relatively simple and systems allowed for easy inspection.

This accordance could be called ‘tinkerability’: the option to look at something (in our interviews: a game or a website), and to look inside the code. This transparency motivates by demonstrating and recreation of the artifact is possible; the purpose is building something like the explored artifact, the puzzle is figuring out how the current code works and how to change it.

Sadly, in the modern digital world it is much harder to explore common technologies when compared to the 80s and 90s. Websites are no longer text with some added markup which is easy to read. Today, much code is hard to read and machine-generated by other tools. This prevents users from understanding the source code but for the companies creating the websites it enables particular programming workflows, smaller file sizes and defense against blocking of advertisements. Games no longer ship with their source code, game consoles prevent users (without deep technical knowledge or expensive tools) to run their own games.

Despite efforts to merge them, such as the work of Bret Victor[3], current programming environments almost always separate written code from the running program in most cases, while our results stress that that division limits the possibility to explore and learn in the broadest sense.

Our findings point to the need for simpler languages for education, with less and not more features, which make the connection between code and execution model more, and not less transparent. Such a move might be contrary to the assumption that visual programming tools such as Scratch are the best first tool for children to start in their programming journey. Scratch's interface does not enable or encourage 'inspection', you cannot click a block and ask what it does. Scratch is not only opaque, but also supports complex language features—also termed *high ceiling* (Resnick et al., 2005)—and nowadays of course also features AI options such as text-to-speech via Amazon Polly [4]. These features might increase purpose and challenge, but decrease transparency even more, as they lean on increased abstraction of the machine.

While browsers do still support features to inspect code, today's web pages are in most cases not understandable by inspecting their source: Much of the markup will be generated server-side, without showing how it is generated. Executable JavaScript code will also often be generated by other tools (like TypeScript) and then, in addition, be minified, with whitespace and variable names stripped from the code to save bandwidth. While the code formally is accessible, it is barely possible to read, even for experts, since how the code was originally written and how it appears to the user are so different.

In other words, the tools that people are creating for programming education and the tools that professionals use for programming both no longer support the transparency that we find is needed for self-teaching. This is a hard problem to address, a teacher could of course create an old-style web 1.0 website, which would be achievable to analyze for learners, but it is unlikely that this would motivate learners, since they are used to more complex websites. The first author, for example, did not learn programming as a child, since they only knew programming in QBasic, which in the 90s already looked old.

6.3. Challenge: Programming tasks as puzzles

Of the three main themes, challenge might be the hardest to integrate into current educational practices and tools. In each group of learners, some will crave hard problems to sink their teeth into (often associated with status, see (Hermans & Schlesinger, 2024)). Other students however might feel insecure, or not interested in the topic, and as such night not be excited by challenge. These are much more likely students with lower prior knowledge and lower self-efficacy, i.e. girls and kids from lower social economic families. If we want to entice them to like programming, challenge is unlikely the way.

It would be possible to integrate challenge into formal education. Challenge can be compatible with many education systems, because in schools lessons are often structured around solving problems with different levels of difficulty. However, these challenges are in most cases given by teachers rather than self-imposed and it remains an open question whether similar excitement and motivation can be created in that fashion. We also encounter the population issue: a potential problem of focusing eduction on challenges can be that great difficulty and overcoming it is emphasized, thus potentially only focusing on the most advanced and competitive learners.

One fruitful related concept here can be that of a *puzzle*. Puzzles are often created by another person, like a teacher giving tasks in school. Puzzles also do not need to be very difficult to be enjoyed. Think of Sudoku or crossword puzzles: Solving them is not the heroic work of lone geniuses but can happen at the kitchen table and be a social activity ("Do you know a word with 5 letters that means...?").

Using a puzzle-model for giving programming tasks has limits: Unlike the challenges of our participants, the puzzles in education are not self-imposed, and thus less connected to activities that learners find

[3]https://worrydream.com/LearnableProgramming/
[4]https://aws.amazon.com/blogs/publicsector/scratch-and-aws-educate-build-new-activity-for-annual-hour-of-code/

personally exciting. A related problem is that it is easy for experienced programmers to come up with programming puzzles that are exclusively abstract and math-like, while learners might be interesting in other ways of using code, for example to create visual patterns or sounds.

In summary, while it might be possible to motivate some learners with challenge, it will not be easy to replicate the success of self-driven motivation in a classroom for all learners, since what one child finds interesting and challenging, might be confusing and frustrating to another.

7. Concluding Remarks

Our goal for this paper was to find out what being self-taught was like for people who were children in the 1980 and early 1990s. We did this based on interviews and thematic analysis. Our goal was to find out what shaped their experiences and enabled them being self-taught as well as what we could learn from these experiences to shape today's tools and programming education.

Based on the interview data, we constructed three main themes: Purpose, that is having a motivation for programming; Transparency, the possibility to inspect how other programs work and to tinker with them; and challenge, a motivated engagement with difficulties.

We discussed how these findings connected to existing research and how the insights could be helpful for helping kids to learn programming today, including those who do not self-teach but get in contact with programming first in the classroom. One interpretation of the results might be to double down on existing ideas of self-guided discovery without guidance which would most likely benefit mostly boys already motivated about learning programming. However we want to emphasize that the basic principles of Purpose, Transparency and Challenge can be also interpreted in a less gendered way: Purpose does not need to be utilitarian or focused on the technology for its own sake. Transparency also means a focus on a powerful yet small and thus understandable feature set and focus on community exchange of code and how-tos rather than only results. Challenge, while on the first glance heroic and conflictual can also be understood as puzzle-like: An activity that follows rules, has an initial state and a solution and which can also be done collaboratively.

Our works shows that we can gain insights from the past experiences of programmers today, while, instead of merely repeating what worked for them in the past, build more inclusive tools and pedagogies based on these insights.

8. References

Aivaloglou, E., & Hermans, F. (2019). How is programming taught in code clubs? Exploring the experiences and gender perceptions of code club teachers. In *Proceedings of the 19th Koli Calling International Conference on Computing Education Research* (pp. 1–10).

Battista, M. T., & Clements, D. H. (1986, January). The effects of Logo and CAI problem-solving environments on problem-solving abilities and mathematics achievement. *Computers in Human Behavior*, *2*(3), 183–193. Retrieved 2019-06-07, from `http://www.sciencedirect.com/science/article/pii/0747563286900026` doi: 10.1016/0747-5632(86)90002-6

Braun, V., & Clarke, V. (2019, August). Reflecting on reflexive thematic analysis. *Qualitative Research in Sport, Exercise and Health*, *11*(4), 589–597. Retrieved 2021-04-02, from `https://doi.org/10.1080/2159676X.2019.1628806` (Publisher: Routledge _eprint: https://doi.org/10.1080/2159676X.2019.1628806) doi: 10.1080/2159676X.2019.1628806

Denny, P., Luxton-Reilly, A., Tempero, E., & Hendrickx, J. (2011, June). Understanding the syntax barrier for novices. In (pp. 208–212). ACM. Retrieved 2019-09-07, from `http://dl.acm.org/citation.cfm?id=1999747.1999807` doi: 10.1145/1999747.1999807

Franklin, D., Coenraad, M., Palmer, J., Eatinger, D., Zipp, A., Anaya, M., ... Weintrop, D. (2020, August). An Analysis of Use-Modify-Create Pedagogical Approach's Success in Balancing Structure and Student Agency. In *Proceedings of the 2020 ACM Conference on International Computing Education Research* (pp. 14–24). New York, NY, USA: Association for Computing Machinery. Retrieved 2025-02-26, from https://dl.acm.org/doi/10.1145/3372782.3406256 doi: 10.1145/3372782.3406256

Gilsing, M., Pelay, J., & Hermans, F. (2022, December). Design, implementation and evaluation of the Hedy programming language. *Journal of Computer Languages*, *73*(101158), 1–17. Retrieved 2023-02-14, from http://www.scopus.com/inward/record.url?scp=85140296845&partnerID=8YFLogxK doi: 10.1016/j.cola.2022.101158

Goldberg, A., & Kay, A. (1977). Methods for teaching the programming language Smalltalk. *Report No. SSL*, 77–2.

Hermans, F., & Schlesinger, A. (2024, October). A Case for Feminism in Programming Language Design. In *Proceedings of the Onward! '24.* Pasadena. (To appear) doi: 10.1145/3689492.3689809

Juul, J. (2024). *Too much fun: the five lives of the Commodore 64 computer*. Cambridge, Massachusetts: The MIT Press.

Kirkpatrick, G. (2017, May). How gaming became sexist: a study of UK gaming magazines 1981–1995. *Media, Culture & Society*, *39*(4), 453–468. Retrieved 2025-04-08, from https://doi.org/10.1177/0163443716646177 (Publisher: SAGE Publications Ltd) doi: 10.1177/0163443716646177

Lewis, C. M., Anderson, R. E., & Yasuhara, K. (2016, August). "I Don't Code All Day": Fitting in Computer Science When the Stereotypes Don't Fit. In *Proceedings of the 2016 ACM Conference on International Computing Education Research* (pp. 23–32). New York, NY, USA: Association for Computing Machinery. Retrieved 2025-05-20, from https://dl.acm.org/doi/10.1145/2960310.2960332 doi: 10.1145/2960310.2960332

Lockwood, J., & Mooney, A. (2018, February). Computational Thinking in Secondary Education: Where does it fit? A systematic literary review. *International Journal of Computer Science Education in Schools*, *2*(1), 41–60. Retrieved 2025-05-27, from https://www.ijcses.org/index.php/ijcses/article/view/26 (Number: 1) doi: 10.21585/ijcses.v2i1.26

Margolis, J., & Fisher, A. (2001). *Unlocking the Clubhouse: Women in Computing* (First Edition ed.). Cambridge, Mass: Mit Pr.

Papert, S. (1980). *Mindstorms: Children, Computers, and Powerful Ideas*. New York, NY, USA: Basic Books, Inc.

Resnick, M., Myers, B., Nakakoji, K., Shneiderman, B., Pausch, R., & Eisenberg, M. (2005, January). Design Principles for Tools to Support Creative Thinking. *Report of Workshop on Creativity Support Tools*, *20*.

Solomon, C., Harvey, B., Kahn, K., Lieberman, H., Miller, M. L., Minsky, M., ... Silverman, B. (2020, June). History of Logo. *Proc. ACM Program. Lang.*, *4*(HOPL), 79:1–79:66. Retrieved 2024-09-22, from https://dl.acm.org/doi/10.1145/3386329 doi: 10.1145/3386329

Turkle, S. (2005). *The Second Self: Computers and the Human Spirit.* The MIT Press. Retrieved 2024-04-22, from https://direct.mit.edu/books/book/2327/The-Second-SelfComputers-and-the-Human-Spirit doi: 10.7551/mitpress/6115.001.0001

LUCY-learn: Visualising Image Classifiers for Education in Machine Learning

Isabel Millar
Computer Laboratory
University of Cambridge
im539@cam.ac.uk

Alan F. Blackwell
Computer Laboratory
University of Cambridge
afb21@cam.ac.uk

Abstract
We present LUCY-learn, an application designed for use in classrooms, that allows students to experiment with the application of different image classifiers to satellite images in the vicinity of the students' school or neighbourhood. The Land Use Classification sYstem (LUCY) illustrates fundamental probabilistic principles of machine learning by visualising the probability distributions of classifier features and output layers. Learning is scaffolded through visualisations ranging from a hue-based pixel classifier to comparison of local image features acquired by a simple convolutional neural network. The result has been evaluated in classroom trials, with successful learning outcomes.

1. Introduction
The constant advance of machine learning (ML) technology means we must begin to integrate machine learning education into K-12 computing education. Developing students' awareness of not only what ML applications can do but how they do it – crucially including their limitations – is critical to a comprehensive understanding of computer science. Despite government endorsement of the use of generative AI in education (Department of Education, 2025), there is no mention yet of ML in the English computer science curriculum (Department of Education, 2014). This is not due to a lack of interest or ability; Evangelista et al found that high school students have a desire to learn more about ML and proposed a workshop that students of this level are capable of understanding, presenting no reason *not* to teach it (Evangelista, Blesio, & Benatti, 2018).

This paper follows previous contributions at PPIG, including a presentation at PPIG 2024 of applications developed by Josephine Rey and Gemma Penson (Rey et al., 2024), one evaluating an interactive visualisation of Causal Bayesian Networks in a UK classroom context and the other extending the Scratch language with PPL functionality to investigate how classroom use of interactive statistical modeling tools in South Africa might support curriculum priorities in that country. In this paper we build on that work with LUCY-learn – the Land Use Classification sYstem – an interactive image classification for iPad, allowing students to visualise and compare different classes of image classifier, applied (as in Rey's work) to satellite images.

2. Previous approaches to ML education
To teach technical details of ML systems, many tools aim to increase transparency of algorithms and training processes, often allowing students to train models themselves. Jiang et al studied students' understanding of text classification models by creating StoryQ, a tool to allow students to investigate and create models, exposing the impact of individual feature weights in the classification of a sentence (Jiang et al., 2022). Focusing specifically on k-means clustering, 'SmileyCluster' introduces students to ML concepts by allowing them to group face glyphs together (Wan, Zhou, Ye, Mortensen, & Bai, 2020). 'Next world adventure' was created to help students conceptually understand n-gram models, allowing students to choose their own sentences (Vahedian Movahed & Martin, 2025). An image recognition tool for high school students was developed by Mariescu-Istodor and Jormanainen, aiming to explain how machine learning applications work, making classifications based on aspect ratio and fullness (Mariescu-Istodor & Jormanainen, 2019).

Tedre et al integrated ML concepts into the computational thinking agenda, highlighting the differences in testing, correctness and transparency of algorithms (Tedre, Denning, & Toivonen, 2021). The range of teaching methods in ML education, extends from workshops and discussion to more hands-on approaches that help engage students and make abstract concepts more tangible (Lee & Kwon, 2024). Broll et al aim to make complex ML principles available to students without previous mathematics knowledge, gradually exposing detail of the training and inference processes (Broll, Grover, & Babb, 2022). Similarly, Winn's *Model-Based Machine Learning* focuses on connecting the abstract mathematics behind ML systems to real-world scenarios where models are used (Winn, 2023). These strategies can be related to constructivist theories of learning that students learn most effectively when participating actively (Triantafyllou, 2022) so that new knowledge is built on previous understanding. Building an understanding of classification by starting with the most simple architecture possible can help to eliminate these misconceptions.

3. The LUCY-learn system

LUCY-learn provides four separate image classifiers, each including controls and output visualisations. Classifiers are displayed in order from simplest to most complex, providing a natural progression to help students build knowledge. Two classifiers are simply pixel based, the first classifying hues by ranges within a distribution, and the second a Bayesian multinomial classifier (Jurafsky & Martin, 2025). The third and fourth classifiers apply convolutional neural networks to demonstrate use of local image features.

In order to support classroom evaluations designed for schools in Oxford and Cambridge, a selection of satellite images from these areas were captured using the MapBox API, chosen to cover a range of different land uses. The project is written in Python and Swift, and the CNNs are implemented using timm. Fine tuning is completed using PyTorch. Models were converted to a format suitable for mobile deployment using CoreMLTools.

3.1. Simple pixel classifier

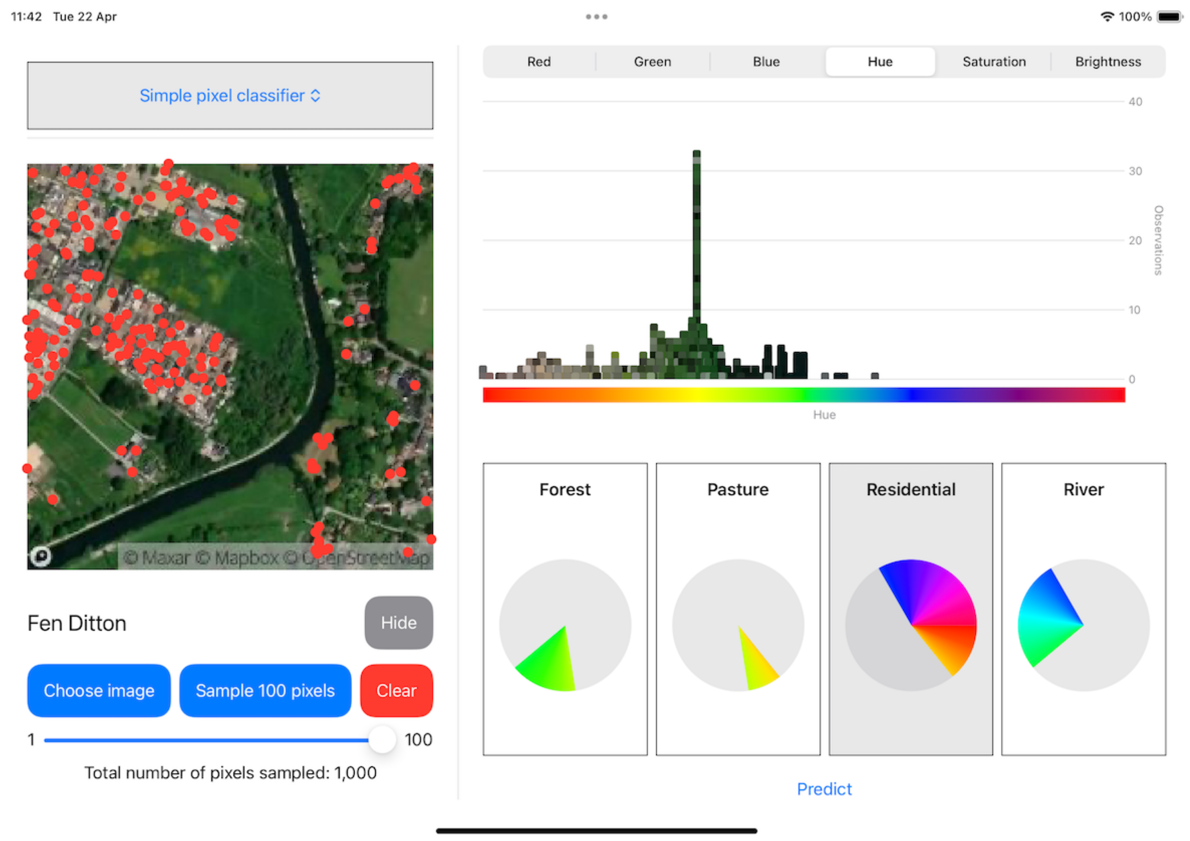


Figure 1 – Hue-based pixel classifier, showing pixels that have been predicted as 'Residential'

The simple pixel classifier tests the simple hypothesis that a hue range may be linked to land use, for example greenish hues may be forest. In many cases this simple assumption is incorrect, which stimulates a discussion about what can be done to improve accuracy. The layout of the dashboard encourages students to think critically about why results are produced.

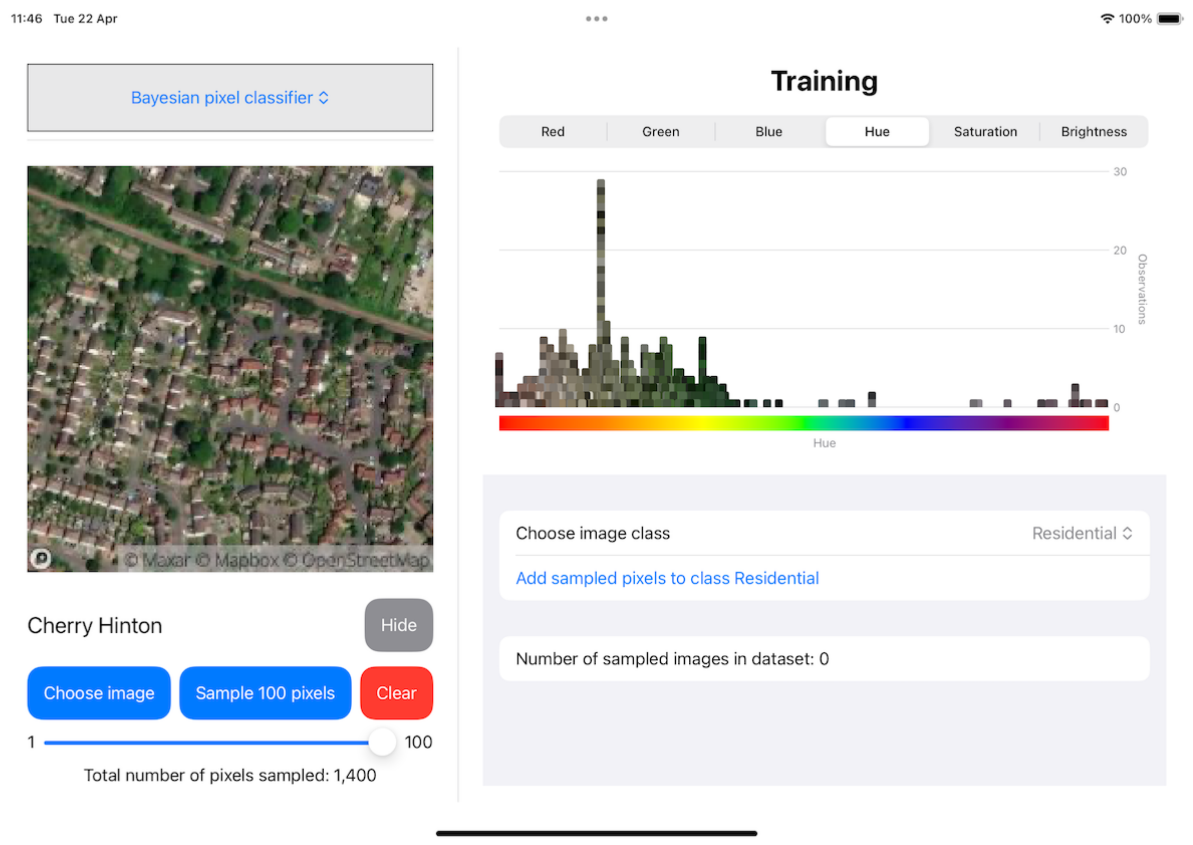
For this introductory level, only four classes are used: Forest, Pasture, Residential, and River. They appear distinctive in satellite images, and represent a large proportion of the land use in Oxford and Cambridge. The classifier makes a judgement based on the hue values of a set of pixels sampled from an image. A method to train this classifier is not provided; instead, the hue ranges are hard-coded into the class, and a prediction function is implemented to calculate the probability of each class.

This adapts the approach of Rey et al (Rey et al., 2024), in which hue values sampled from a satellite image are visualised as a histogram and pie chart. Our version includes an option to hide the satellite image, encourage students to make their own judgement from the distribution of hues. Trying (and potentially failing) to guess what the image looks like with only limited information helps them understand why more features are needed for better classifications. The controls allow students to select a satellite image of their choice and sample from one to one hundred pixels.

Sampled pixels update the histogram and pie chart (figure 1) in real time. The histogram allows the distribution to be sorted by red, green, blue, hue, saturation or brightness. The UI displays each of the hues as a sector of the colour wheel, showing students which hues are classed as which land use. These can be selected to highlight which pixels in the image are in this range of hues (Figure 1).

The incorrect classifications made by this simple classifier exposes the problems of using local colour as the only feature for classification. For example, when provided with an image that students would label as ‘River’, the classifier predicts ‘Forest’ because the proportion of ‘blue’ pixels in the image is lower than the ‘green’ pixels of the banks.

3.2. Bayesian classifier



Cherry Hinton

Total number of pixels sampled: 1,400

Choose image class: Residential

Add sampled pixels to class Residential

Number of sampled images in dataset: 0

Figure 2 – The training interface for the Bayesian classifier.

The second classifier is a Bayesian multinomial classifier (Jurafsky & Martin, 2025). This provides an introduction to both Bayesian probability and training and inference processes. By allowing students to build a dataset and train a classifier the concepts of prior and posterior probability are illustrated, as classifications for an image update based on previously seen data.

As this classifier also uses pixel values, the UI is very similar to the simple pixel classifier, with the same controls to allow the user to sample pixels and choose images, and the same hue distribution histogram (figure 2). Buttons are added to the main visualisation panel to label a set of pixels based on the image's class, add this to the training dataset, and train the classifier. Once the classifier has been trained, students can sample pixels and make predictions.

The small volume of data used to train this model allows students to perform data collection and labelling themselves. While this does not produce a very accurate classifier, it is sufficient to help students to understand training, inference and Bayesian probability. The relatively poor accuracy helped facilitate a discussion about the amount of data needed to train models, and the effects of a biased or incomplete dataset.

3.3. CNN land use classifier

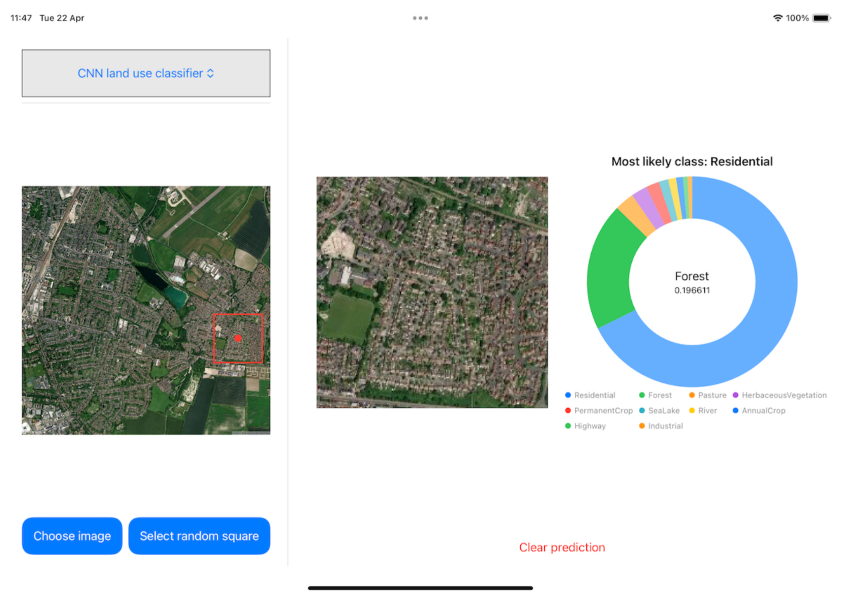


Figure 3 – CNN land-use classifier, showing part of a satellite image that has been selected from the larger area on the left with the 'Forest' section selected in the ring chart.

The CNN classifier employs a model pre-trained from EuroSat images as the basis for land use classification [1]. A visualisation of convolution features was also included in the app, and was used to support additional Year 12 curriculum content on the use of matrices in image processing (not discussed further here). The land use classifier dashboard shows magnified patches extracted from a larger satellite image. In the same style as the simple pixel classifier, a ring chart displays the probabilities of each land use class (figure 3). This reinforces that classifiers make classifications based on probabilities.

3.4. CNN school classifier

The final classifier is a CNN that we trained to classify image regions as either 'school' or 'not school', with a novel visualisation to help students assess why certain images may be seen as more or less 'school-like' (figure 5). We created a training dataset including images of all schools in Oxford and Cambridge, using coordinates from `https://get-information-schools.service.gov.uk/`, and images collected using the Mapbox API. A contrast set of not-school images from these regions was randomly generated. Once a model with suitable accuracy was obtained, it was converted to a CoreML

[1] `https://huggingface.co/cm93/resnet18-eurosat`

model and integrated into the app in the same way as the previous CNN.

We created a visualisation of ‘schoolness’ in the same style as the hue distribution histogram in the pixel classifiers, providing continuity throughout the app and helping students learn by building on their previous understanding of the pixel classifier. The schoolness scale on the x-axis corresponds to the output probability of the classifier. Values are sorted into ten bins, and the x-axis shows a thumbnail example of an image classified in each bin. This allows students to see trends in how the image changes to be more school-like, identifying features that might have more impact on schoolness. Students can then test other images with similar features to see if they are classified similarly.

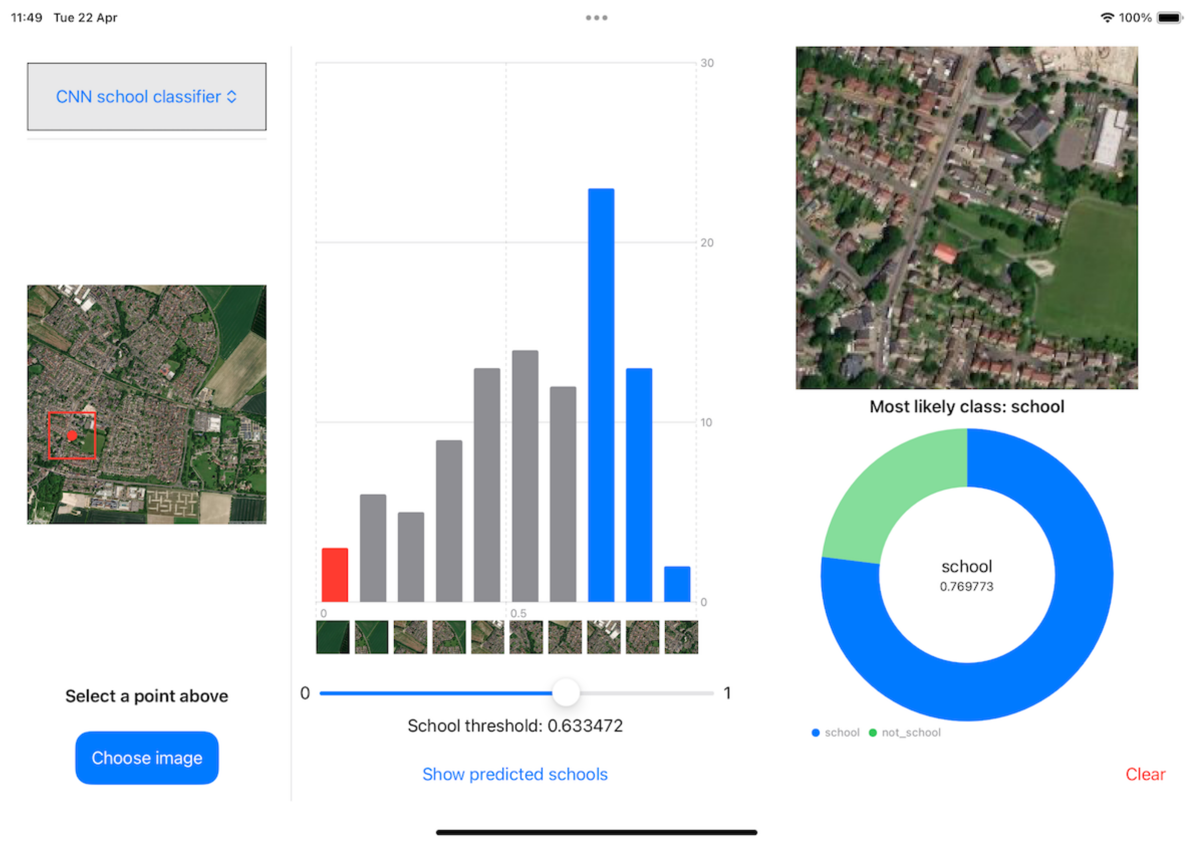


Figure 4 – School classifier prediction with histogram showing bins for ’schoolness’

The visualisation includes a slider to change the schoolness threshold. This slider was used in lessons to help facilitate discussions about accuracy, and what might constitute a suitable value for different applications. This prompted conversation about precision and recall, for example we might want a classifier that never produces a false positive prediction, or in contrast we might want to be sure all schools have been selected and can tolerate selecting some extra images as well.

4. Learning Evaluation

Machine learning is not currently included in the English national curriculum, meaning there was no basis for comparison to current teaching. The goal was to develop students’ understanding of image classification and the probability concepts underlying classification methods. School trials were conducted by the first author in two classes – one year 10 computer science class and one year 12 further mathematics class. The variation in age and ability allowed us to focus on evaluating slightly different parts of LUCY-learn with each group, using lesson plans following a similar structure, but with more mathematical detail (of logarithms and matrices) for the more advanced class. The study was approved by the Ethics Committee of the Cambridge Department of Computer Science and Technology.

Students completed pre-lesson and post-lesson questionnaires, using a long-answer open question to

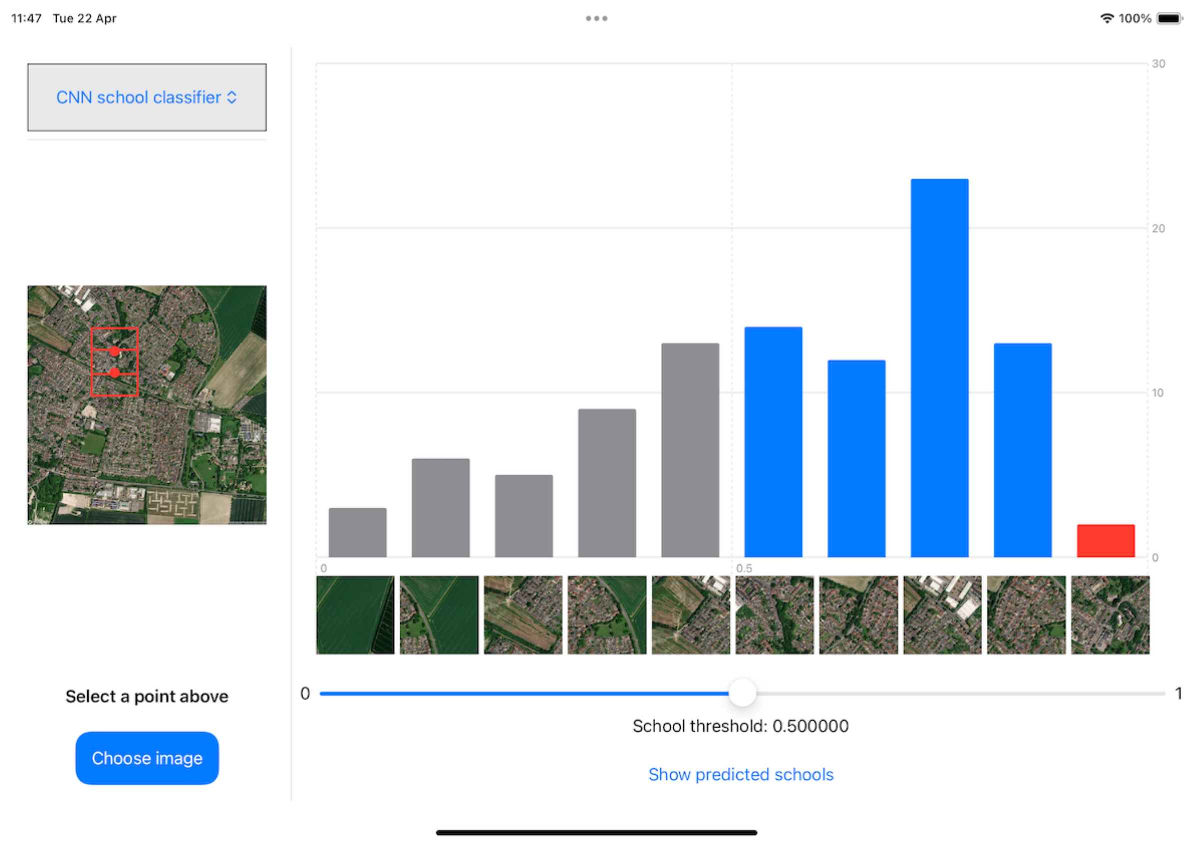


Figure 5 – The novel histogram visualisation of 'schoolness'.

enable qualitative evaluation of the development of understanding. Students also completed a Likert-scale evaluation, to report their assessment of LUCY-learn by comparison to any previous learning about image classifiers or ML.

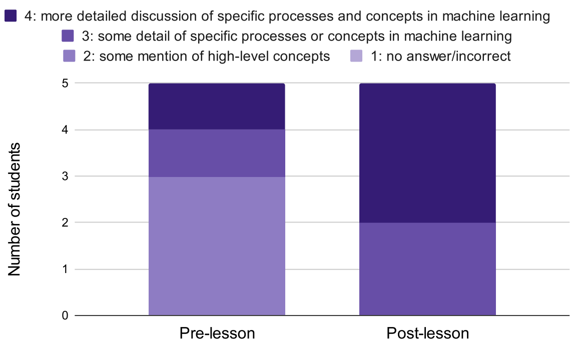
4.1. Year 12 lesson

The first trial was with a group of seven Year 12 students in a Further Mathematics class at Oxford High School (OHS). Students were allocated one school iPad between two, allowing them to collaborate on tasks. Students picked up details of the two pixel-wise classifiers quickly and were able to correctly use the app to train the Bayesian multinomial classifier, understanding why predictions were being made and the impact of the dataset in the prediction process. When discussing performance of the classifiers, students were able to identify issues with accuracy and suggest solutions for these problems. Overall, students were able to successfully use LUCY-learn to pick up concepts and put them into practice.

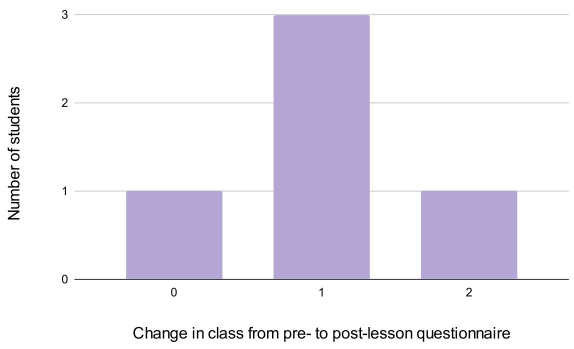
Students answered the following question before and after the lesson:

> Websites often use CAPTCHA tests – for example 'select all squares with traffic lights in' – to check if a user is human. Self-driving cars can use image classifiers to identify hazards on the road and therefore make decisions about actions to take. How can the answers collected from CAPTCHA tests help improve self-driving cars' ability to recognise objects on the road?

Students' answers were separated into four classes (figure 6): no answer/incorrect (1), some mention of high-level concepts (2), some detail of specific processes or concepts in machine learning (3), and more detailed discussion of specific processes and concepts in machine learning (4). The development of students' understanding through the lesson is evident in the figure. Answers to the pre-test question stated that CAPTCHAs collect labels from humans and can be used to train self-driving cars, but with



(a) Classes



(b) Change in class

Figure 6 – Evaluation of pre-lesson and post-lesson responses for year 12 class.

no detail of what this training entails. Some students mentioned that cars may make a prediction based on an image being "similar" to one it has already seen, but did not explain what constitutes "similar" images.

In the post-lesson questionnaire, all students mentioned probability or the inner workings of CNNs, as well as how we may use the output of a CNN to provide a confidence rating and make a decision of class based on this. Discussion of probabilistic qualities of classifiers, such as the fact that the and of confidence indicates understanding of the important distinction between data-driven systems and the rule-driven systems addressed in the current computer science curriculum (Caspersen, 2023).

4.2. Year 10 lesson

To evaluate the educational benefits provided by LUCY-learn over a range of ages and abilities, a second trial was completed with 17 mixed-ability Year 10 students in a computer science class at St Bede's School in Cambridge. The school does not have iPads, so the lesson was taught by projecting a demonstration, then handing the iPad around for students to try themselves. Students had previously completed a topic on data representation including images, so the lesson placed a greater focus on the simple pixel and Bayesian classifiers to tie into the year 10 (UK GCSE) curriculum. The lesson went well and students went from being able to identify barely any applications of machine learning to making intelligent analyses about why and how different classifiers make classifications.

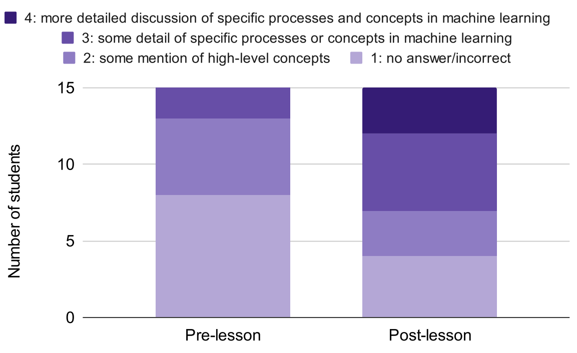
The evaluation question for this lesson was simplified to suit year 10 students' age and experience:

> Some smart cameras like ring doorbells can detect objects in the photos and videos they take, for example a package being delivered or a person at your door. This is called image classification. How does a computer figure out what object is in a picture?

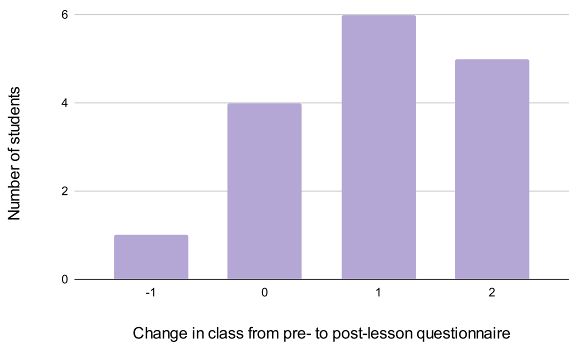
In the pre-lesson questionnaire, only seven students attempted an answer, in contrast to the eleven attempts in the post-lesson questionnaire.

Most students were able to understand the role of colours in the classification of an image in the post-lesson questionnaire, with many making reference to proportions of each colour. Some commented only on different colours suggesting different classes – while others integrated more detail about probability and accuracy of classifiers.

Higher-achieving students attempted explanations in their pre-lesson questionnaires, spanning a range of "based off other similar images on the internet" (P13) to the general comment of "machine learning" (P6). Most of these students' answers were much more developed in the post-lesson questionnaire, referencing probabilities and in some cases more complex architectures. P14 started with an understanding of needing a database of images to check against, to an explanation of how a probability value for each class can be constructed in a pixel-level classifier and CNNs can extract other features in images.



(a) Classes



(b) Change in class

Figure 7 – Evaluation of pre-lesson and post-lesson responses for year 10 class

Answers were grouped into the same classes as for the year 12 lesson (figure 7). Overall, eleven of the fifteen student questionnaires showed a development in understanding, with their post-lesson answer being classed higher than their pre-lesson one. These answers made comments about the use of colour in classification, as well as the need for labelled data, large datasets, and classifications' reliance on probability. The results show a clear development in understanding.

Category	LUCY-learn		Other ML	
	Mean	STD	Mean	STD
Understanding	4.6	0.52	4	0.96
Ease of use	4.47	0.52	5	0.52
Application	3.87	0.52	3.83	0
Interest	4.1	0.67	3.75	0.5

Table 1 – Summary of the results from Year 12 students' Likert-scale evaluations.

Likert scale responses showed that LUCY-learn was generally well received by year 12 students, with no negative values. Table 1 shows a summary of the mean and standard deviation for each category of questions. Students' self-evaluation of their progression was that LUCY-learn helped them to develop their own understanding, but they felt less confident to apply their knowledge to other situations or to teach other students. For year 10 students, only one student had learnt about machine learning before, so we do not report scale comparisons. Students in the year 10 class also reported that they might struggle to apply what they had learned.

5. Conclusion

We have described an app that builds on the PPIG 2024 paper by Rey et al (Rey et al., 2024). Rey's work presented an extension to Scratch, in which a Scratch background based on satellite images could be used to explore basic principles of Bayesian probability. The LUCY-learn app focuses specifically on satellite images as an example of machine learning classifiers. Where Rey's work was designed for deployment in schools in South Africa, informed by consultation with educators in that country, LUCY-learn focuses on extending the UK computer science and mathematics curriculum with fundamental principles of machine learning.

LUCY-learn provides a progressive model in which students are able to experiment with more sophisticated approaches to image classification, from simple hue categories to convolutional deep networks. The implementation of these models includes an opportunity for students to experiment with training their own classifier, evaluation of how pretrained classifiers generalise to new contexts, and critical assessment of the accuracy of specialised deep learning models via a novel visualisation of class likelihoods for sub-images.

Classroom evaluation confirmed that students at different school levels and with a range of abilities were able to learn core principles of machine learning, beyond the current UK curriculum. This understanding was evaluated in relation to scenarios recognisable to students, although there is still room for development of LUCY-learn to allow students to test their understanding in different situations. One example of this would be to integrate another way to build their own classifier in the app, perhaps with more detail than the existing Bayesian classifier training, potentially exposing students to some code and allowing them to test their understanding. The single lesson that we taught did not yet give students confidence that they could apply this knowledge, but this could be addressed by integrating more practical activities into the app.

6. References

Broll, B., Grover, S., & Babb, D. (2022). Beyond black-boxing: Building intuitions of complex machine learning ideas through interactives and levels of abstraction. In *Proceedings of the 2022 acm conference on international computing education research - volume 2* (p. 21–23). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/3501709.3544273 doi: 10.1145/3501709.3544273

Caspersen, M. E. (2023). Principles of programming education. In (2nd ed., chap. 18). Bloomsbury Publishing.

Department of Education. (2014). The national curriculum in england: Key stages 3 and 4 framework document [Computer software manual]. Retrieved from https://www.gov.uk/government/publications/national-curriculum-in-england-secondary-curriculum

Department of Education. (2025). *Generative artificial intelligence (ai) in education.* Retrieved from https://www.gov.uk/government/publications/generative-artificial-intelligence-in-education

Evangelista, I., Blesio, G., & Benatti, E. (2018). Why are we not teaching machine learning at high school? a proposal. In *2018 world engineering education forum - global engineering deans council (weef-gedc)* (p. 1-6). doi: 10.1109/WEEF-GEDC.2018.8629750

Jiang, S., Nocera, A., Tatar, C., Yoder, M. M., Chao, J., Wiedemann, K., … Rosé, C. P. (2022). An empirical analysis of high school students' practices of modelling with unstructured data. *British Journal of Educational Technology*, *53*(5), 1114-1133. Retrieved from https://bera-journals.onlinelibrary.wiley.com/doi/abs/10.1111/bjet.13253 doi: https://doi.org/10.1111/bjet.13253

Jurafsky, D., & Martin, J. H. (2025). *Speech and language processing: An introduction to natural language processing, computational linguistics, and speech recognition with language models* (3rd ed.). Retrieved from https://web.stanford.edu/~jurafsky/slp3/ (Online manuscript released January 12, 2025)

Lee, S. J., & Kwon, K. (2024). A systematic review of ai education in k-12 classrooms from 2018 to 2023: Topics, strategies, and learning outcomes. *Computers and Education: Artificial Intelligence*, *6*, 100211. Retrieved from https://www.sciencedirect.com/science/article/pii/S2666920X24000122 doi: https://doi.org/10.1016/j.caeai.2024.100211

Mariescu-Istodor, R., & Jormanainen, I. (2019). Machine learning for high school students. In *Proceedings of the 19th koli calling international conference on computing education research.* New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/3364510.3364520 doi: 10.1145/3364510.3364520

Rey, J., Penson, G., Blackwell, A. F., Ge, H., Li, X., & Arnold, H. (2024). Educational tools for probabilistic machine learning curriculum in schools. In *Proceedings of the 35th annual workshop of the psychology of programming interest group (ppig).*

Tedre, M., Denning, P., & Toivonen, T. (2021). Ct 2.0. In *Proceedings of the 21st koli calling international conference on computing education research.* New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/3488042.3488053 doi: 10.1145/3488042.3488053

Triantafyllou, S. A. (2022). Constructivist learning environments. In *Proceedings of the 5th international conference on advanced research in teaching and education.*

Vahedian Movahed, S., & Martin, F. (2025). From pre-conceptions to theories: How middle school student ideas about predictive text evolve after interaction with a new software tool. In *Proceedings of the extended abstracts of the chi conference on human factors in computing systems.* New York, NY, USA: Association for Computing Machinery. Retrieved from `https://doi.org/10.1145/3706599.3719789` doi: 10.1145/3706599.3719789

Wan, X., Zhou, X., Ye, Z., Mortensen, C. K., & Bai, Z. (2020). Smileycluster: supporting accessible machine learning in k-12 scientific discovery. In *Proceedings of the interaction design and children conference* (p. 23–35). New York, NY, USA: Association for Computing Machinery. Retrieved from `https://doi.org/10.1145/3392063.3394440` doi: 10.1145/3392063.3394440

Winn, J. (2023). *Model-based machine learning*. CRC Press. Retrieved from `https://books.google.co.uk/books?id=y_jYEAAAQBAJ`

Vibe coding: programming through conversation with artificial intelligence

Advait Sarkar
University of Cambridge and
University College London
United Kingdom
advait.sarkar@cl.cam.ac.uk

Ian Drosos
Microsoft Research
United Kingdom
t-iandrosos@microsoft.com

Category	**Associated questions**
Goals	What kinds of applications are vibe coded? What degree of success is expected? To what extent is exploration a motivation? (Section 3.1)
Intentions	How are implementation ideas initially formed and refined? Is intent pre-defined or shaped iteratively through interaction with AI? (Section 3.2)
Workflow	What are the stages of the vibe coding workflow? How is programmer effort distributed? How are various tools and resources combined? (Section 3.3)
Prompting	How do programmers express their intent to the system? What setup and prompting strategies are used? What is the granularity of instructions? Are prompts single- or multi-objective? What input modes are employed? (Section 3.4)
Debugging	What methods are used to identify and resolve problems in AI-generated code? (Section 3.5)
Challenges	What technical, conceptual, and workflow-related obstacles arise in vibe coding beyond debugging? What strategies are used to overcome them? (Section 3.6)
Expertise	What forms of knowledge are deployed in vibe coding? When do practitioners transition from instructing AI to direct manual work? (Section 3.7)
Trust	How do users build confidence in AI outputs? What are the risks of overreliance? (Section 3.8)
Definition and Performance	How is vibe coding understood relative to other AI-assisted programming practices? How do performative contexts such as video streaming shape the practice and its perception? (Section 3.9)

Table 1 – Major categories of findings from our study of the practice of vibe coding. Findings were derived from in-depth qualitative framework analysis of approximately 8.5 hours of video of extended think-aloud vibe coding sessions.

Abstract

We examine “vibe coding”: an emerging programming paradigm where developers primarily write code by interacting with code-generating large language models rather than writing code directly. We present the first empirical study of vibe coding. We analysed over 8 hours of curated video capturing extended vibe coding sessions with rich think-aloud reflections. Using framework analysis, we investigated programmers’ goals, workflows, prompting techniques, debugging approaches, and challenges encountered.

We find that vibe coding follows iterative goal satisfaction cycles where developers alternate between prompting AI, evaluating generated code through rapid scanning and application testing, and manual editing. Prompts in vibe coding blend vague, high-level directives with detailed technical specifications. Debugging remains a hybrid process combining AI assistance with manual practices.

Critically, vibe coding does not eliminate the need for programming expertise but rather redistributes it toward context management, rapid code evaluation, and decisions about when to transition between AI-driven and manual manipulation of code. Trust in AI tools during vibe coding is dynamic and contextual, developed through iterative verification rather than blanket acceptance. Vibe coding is an evolution of AI-assisted programming that represents an early manifestation of “material disengagement”, wherein practitioners orchestrate code production and manipulation, mediated through AI, while maintaining selective and strategic oversight.

1. Introduction and Background

On February 2, 2025, the influential computer scientist Andrej Karpathy posted to Twitter/X describing the idea of “vibe coding” (Karpathy, 2025): *“There’s a new kind of coding I call "vibe coding", where* *you [...] forget that the code even exists. [...] I barely even touch the keyboard. [...] I "Accept All" always, I don't read the diffs anymore. When I get error messages I just copy paste them in with no comment, usually that fixes it. The code grows beyond my usual comprehension, [...] but it's not really coding - I just see stuff, say stuff, run stuff, and copy paste stuff, and it mostly works."*

This description of a new style of programming, that shall henceforth be referred to as the Karpathy canon, has had significant influence. Beyond sparking discourse, it has spurred the adoption of the term "vibe coding" to describe an idealised style of programming involving the use of "agentic"[1] software development tools such as Cursor, GitHub Copilot Agent mode, Windsurf, Bolt, and others. From the excerpt above, it would seem that the core ideal of vibe coding is disengagement from directly working with code (authoring, editing, reading, etc.) and instead trusting the agentic tool to perform these operations reliably enough for practical use, based on natural language (preferably dictated) descriptions of the vibe coder's intent.

The concept of an idealised philosophy of programming being seeded by an influential figure is something of a trope in programming practice. Prior examples include literate programming, seeded by Knuth (1992), or egoless programming, seeded by Weinberg (1971). Perhaps the most novel aspect of the Karpathy canon is its brevity. While literate and egoless programming are each accompanied by a book-length treatise, the Karpathy canon is a 185-word tweetise.

What often occurs following such philosophical seeding is the collective mobilisation of the community of programmers to make sense of, adapt, and adopt (or resist) this proposed way of working. For some, the Karpathy canon merely provides a convenient label for a practice they were already engaged in. For others, the term is an introduction to a new way of working, but with innumerable unanswered questions that need to be negotiated through personal practice on a case-by-case basis. Is it still vibe coding if I manually edit the code from time to time? Is it still vibe coding if I use the keyboard instead of dictation? Is it vibe coding if I use a non-agentic tool? How do I vibe code effectively? What is the spirit of vibe coding, and how does it differ from its letter?

The manner by which a community, through reflexive practice and discourse, comes to hash out these questions and elaborate the concept of vibe coding is a sociological question, which we will not attempt to address in this paper. We raise this to emphasise that, despite nucleating around the Karpathy canon, vibe coding is an *emerging* phenomenon and its properties are being *continuously negotiated* as of this writing.

Thus, to isolate vibe coding as an object of study, the only criterion we adopt that connects all the disparate activities that programmers engage in while vibe coding or attempting to vibe code is that the programmer self-describes their activity as vibe coding. This may result in false negatives; we might exclude programmers engaging in an activity that to an external observer would have qualified as vibe coding, but which the programmer themselves does not explicitly identify as vibe coding. But with an awareness and appreciation of the breadth of experimentation and negotiation required for a programming community to align on the boundaries of a concept such as vibe coding, we hesitate to characterise any self-identified vibe coding session as a "false positive", as that would entail us, as researchers, applying a norm for what constitutes vibe coding where in fact none exists.

As an emerging programming practice and sociological phenomenon, vibe coding is intrinsically worth exploring as it prompts us yet again to revisit longstanding questions in human factors of programming research, such as the role of expertise in programming, programmer agency and control, and so on, much as it was worth characterising the novel nature of programming with earlier generations of code-

[1]Unfortunately, there is no consensus on the definition of the term "agentic" or "agent" in this context. It is primarily a commercial term rather than a technical term. We invoke this term to make an explicit connection with that commercial discourse, and do not make ontological claims about "agents" as a clearly distinct category of AI tools. For our purposes, the property of interest is that tools described as "agentic" can typically take a greater range of programmatic actions on the user's content (such as reading code, making direct edits across multiple files, and executing terminal commands) than previous implementations of AI assistance within software development environments.

generating language model tools such as GitHub Copilot (Sarkar et al., 2022a; Lee et al., 2024; Barke et al., 2023; Vaithilingam et al., 2022).

Moreover, it can be argued that programming practice is the canary in the coal mine, so to speak, for how generative AI affects practice in knowledge work more broadly. Programmers themselves are at the frontiers of developing generative AI tools for knowledge workers, and the unique genesis of modern programming practice, forged as it was from an identity crisis in the nature of programming as craft, science, or engineering (Ensmenger, 2012) has arguably endowed the everyday practice of programming with an unusually high degree of reflexivity. Arguably, programmers are in the privileged position of being among the first wave of knowledge workers to identify new opportunities for applying generative AI to their practice, and also empowered to build these frontier tools for themselves. In essence, where programming goes, so follows the rest of knowledge work.

If the Karpathy canon is taken at face value, vibe coding is our first glimpse at a knowledge workflow subject to material disengagement. By this we mean that the *material substrate* of programming, i.e. code, is no longer worked by the programmer. The programmer deliberately disengages from working in the material substrate to instead orchestrate its production and editing via an AI producer-mediator. It does not take any great leap of the imagination to suppose that a similar disposition or workflow of material disengagement could be applied to all knowledge work tasks, whether writing reports, or emails, or producing spreadsheets, or presentations. Studying how and to what extent material disengagement affects the practice of programming, in its highly natural and externally valid manifestation of vibe coding, gives us a valuable opportunity to anticipate potentially forthcoming changes in the structure of knowledge work.

Thus motivated, **we contribute the first empirical analysis of vibe coding** as an emerging programming phenomenon, through a framework analysis of curated think-aloud videos of programmers vibe coding sourced from YouTube and Twitch (Section 2). Our analysis provides a picture of how goals and intentions are formed during vibe coding, the vibe coding workflow, the nature of prompting, debugging practices, challenges that arise during vibe coding, what kinds of programmer expertise are deployed and how, and the nature of trust in AI during vibe coding (Section 3). We discuss how vibe coding appears to represent an evolution of previous generations of AI-assisted coding, the vibe "gestalt", the tradeoffs of material disengagement, and how vibe coders attempt to influence societal attitudes to the use of AI (Section 4).

Our analysis is preliminary; we study only a small set of videos in detail, and the landscape of tools and practices associated with vibe coding is rapidly evolving. Our findings should therefore be understood as a temporally contingent snapshot of how vibe coding appears to manifest at this specific early juncture. Such a vignette serves both as a foundation for future work on understanding the vibe coding phenomenon, and as a reference point for comparison as the practice evolves further.

2. Method

2.1. Data sources

Our approach to studying vibe coding through think aloud videos on YouTube and Twitch draws inspiration from prior work. Barik et al. (2015) demonstrated that online community investigations can "yield insights into qualitative research topics, with results comparable to and sometimes surpassing traditional qualitative research techniques".

Analysis of online sources such as Hacker News, YouTube, Reddit, and other online fora has shed light on aspects of programming practice such as programming as play (Barik, 2017), the introduction of the lambda function into Excel (Sarkar et al., 2022b), and the first generation of code-generating large language models (Sarkar et al., 2022a). Just as Hacker News serves as "an important venue for software developers to exchange ideas as part of a broader cultural ecosystem" (Barik et al. (2015), referring to Wu et al. (2014)), YouTube and Twitch function similarly for the vibe coding phenomenon. These

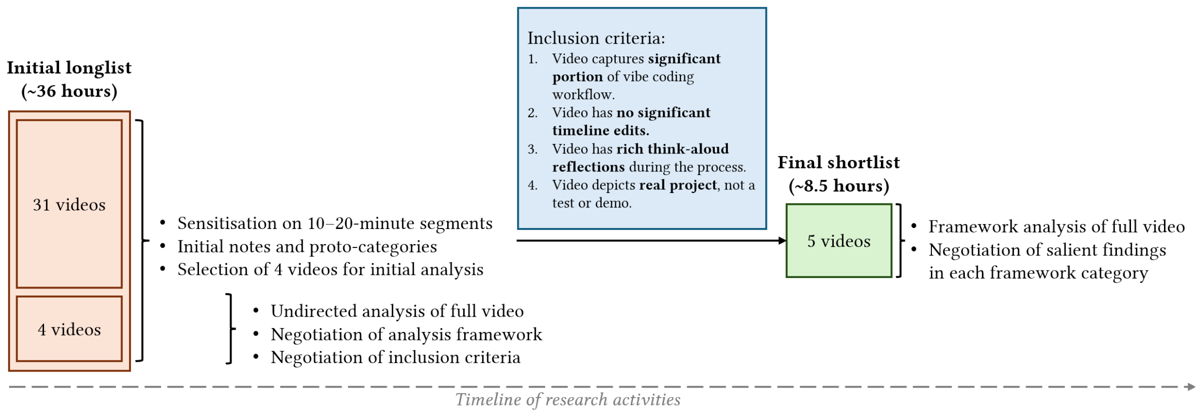

Initial longlist (~36 hours)
31 videos
4 videos
• Sensitisation on 10–20-minute segments
• Initial notes and proto-categories
• Selection of 4 videos for initial analysis
• Undirected analysis of full video
• Negotiation of analysis framework
• Negotiation of inclusion criteria
Inclusion criteria:
1. Video captures significant portion of vibe coding workflow.
2. Video has no significant timeline edits.
3. Video has rich think-aloud reflections during the process.
4. Video depicts real project, not a test or demo.
Final shortlist (~8.5 hours)
5 videos
• Framework analysis of full video
• Negotiation of salient findings in each framework category
Timeline of research activities

Figure 1 – Overview of research process. An initial longlist of 35 videos was analysed in two phases, resulting in a framework for qualitative analysis and a set of inclusion criteria. Applying these criteria, 5 videos (approximately 8.5 hours) were analysed in detail in their entirety according to the framework.

platforms are social environments where programming practices are formed and enacted, making them valuable contexts for studying emerging programming practices.

While analysing public discourse regarding vibe coding could help understand how it is *perceived* within programming communities, at this preliminary stage we prioritised studying the *activity* of vibe coding, on the basis that an understanding of how vibe coding is practiced is a necessary prerequisite to understanding any associated discourse. However, with the foundational understanding that this paper provides, an analysis of the discourse would be suitable for future work.

2.2. Video analysis method

Given that vibe coding is an emerging phenomenon, we sought the flexibility to document isolated events, sequential event chains, concurrent events, variations in the duration of phenomena of interest, and temporally disconnected but conceptually related events (e.g., when a programmer initiates an activity, transitions to another task, and subsequently resumes the original work). Thus we chose not to segment videos into predefined episodes or fixed time intervals (e.g., 20-second segments, as exemplified in Srinivasa Ragavan et al. (2021)). Moreover, we required a method that permitted flexible analysis of both verbal think-aloud information and observed actions within the videos.

Consequently, we adopted framework analysis (Ritchie and Spencer, 2002; Goldsmith, 2021), a form of directed content analysis wherein researchers first familiarise themselves with the data, develop an analytical framework, and subsequently index data according to this framework.

Unlike codes in a qualitative codebook, framework categories do not represent concrete phenomena but rather constitute conceptual domains into which observations can be classified. This approach thus requires less initial commitment to an ontology of findings. Rather, the indexing process serves to reorganise the data (they are used merely to collate data *about* a concept of interest, rather than data *in support of* a phenomenal theme) so that the directed research questions implied by the framework categories can then be addressed through various interpretive methods. In our case, because the size of recorded observations in each category was relatively small (around 2000 words of combined researcher notes per category), two researchers were able to collaboratively negotiate the most salient findings from each category.

2.3. Video selection

We selected an initial longlist of videos by searching the YouTube and Twitch platforms using platform native search using the search terms “vibe coding” and “vibe coding session”. We considered but refrained from including a wider array of search terms (e.g., including tool names such as "Cursor") because as mentioned in Section 1, vibe coding is an emerging practice that is being continuously defined and negotiated and our study is partly an exercise in understanding that process; we thus adopt the methodological principle of ordinary language philosophy (Laugier, 2013) of attending to the term as it is used and enacted in practice, rather than attempting to impose any *a priori* interpretations of vibe coding, as including other keywords would have done.

Searches were conducted during the period 17 March 2025 to 02 April 2025. Prior to inclusion, videos were lightly inspected to exclude any false positive videos that clearly contained no developer reflections about the concept of vibe coding (e.g., advertisements, slideshows with text-to-speech voiceovers, ambient music videos, etc.) and extremely short videos (less than 3 minutes). This resulted in an initial list of 35 videos with a combined duration of 35 hours, 46 minutes and 21 seconds.

Next, two researchers independently viewed 10-20 minute sections of each video to sensitise and familiarise themselves with the content, making initial notes about potential themes and quality criteria. Through negotiation, four videos were selected for the two researchers to view in their entirety to perform a preliminary qualitative analysis.

Next, the two researchers met to discuss their analyses of the four videos with the objectives of a) determining the quality criteria for videos to be included in the final analysis, and b) determining the categories of the framework analysis.

We determined the following four criteria for inclusion. First, we prioritise videos that capture a *significant portion* of the vibe coding workflow, ideally from end to end (starting with a blank project and ending when the project is complete), so that the researchers can understand the full context of the project and any behaviours displayed during vibe coding. Second, we prioritise videos without significant timeline edits, which video creators often make to shorten videos and improve audience engagement, but which can compromise the value of the video as a reflection of the true vibe coding process. Third, we prioritise videos with rich think-aloud reflections from the developer *during* the coding session, aiming to exclude those with post-hoc voiceovers or those with significant pauses and silences where coding activity is occurring but without spoken reflection. Fourth, we prioritise videos in which the project being worked on has a sufficient level of realism, external validity, or seriousness, as opposed to videos of toy projects made purely for demonstration.

The researchers revisited the longlist and annotated each video according to these quality criteria, negotiating with each other if any uncertainties arose. Out of the initial list of 35, only 6 videos met all four criteria unambiguously. In the intervening time, one of these had been withdrawn from the Twitch streaming platform. Consequently, our final list for deeper framework analysis consisted of 5 videos with a combined duration of 8 hours, 27 minutes, and 19 seconds. Of these, two videos, YT22a and YT22b, were two parts of the same vibe coding project by the same developer – in our results we therefore combine findings from both under the single ID YT22.

The full list of videos considered, including the four videos selected for the framework development and the five videos analysed in detail are given in Appendix A.

2.4. Framework analysis

For determining the categories of the framework analysis, the researchers independently compiled notes while viewing videos from the entire longlist on potential categories and research questions to address, resulting in a combined list of 22 proto-categories at various levels of granularity. After negotiation, three categories (on programmer goals, the formation of intention, and the deployment of expertise) were further elaborated to accommodate nuances that researchers had observed at this initial stage. Two proto-categories were merged into a broader category on workflows and tools, and one proto-category was removed, namely, how programmers and non-programmers differ in their approach to vibe coding, because our dataset did not contain any vibe coding sessions featuring a user with no programming expertise.

Our final framework consisted of 9 top-level categories organised into 20 subcategories. The 9 top-level framework categories were: goals, intentions, workflow, prompting, debugging, challenges, expertise, trust, and definition and performance. An overview of these categories and some of the questions associated with each is given in Table 1. The full framework, including subcategories and elaborations of specific questions, is given in Appendix B.

Finally, two researchers divided the dataset into two parts of similar duration, and independently applied the framework to qualitatively analyse one part each. In a slight deviation from the standard method of framework analysis where units of the analysed data are directly attached ("indexed") to framework categories, in this case the researchers made qualitative notes and observations, including excerpts from the think aloud transcripts of the videos, under each category as appropriate. This deviation is necessary to accommodate the difficulties with unitising videos to properly account for emerging phenomena (as mentioned earlier) and also to account for the fact that a phenomenon may be observed by the researcher as manifesting through an irreducible combination of onscreen actions and think-aloud reflection (which may not occur simultaneously). An overview of the research process is given in Figure 1. The results of this analysis are presented in Section 3.

3. Results

3.1. Goals

3.1.1. What kinds of applications are built using vibe coding?

Our examples spanned varying domains and complexity levels. In the simplest case (YT15), a creator developed an animated diagram explaining a coding agent's workflow. This project focused primarily on visual design elements, including arrow positioning, labelling, animation, and aesthetic styling with specific colour themes. In TW1, the creator created and successfully deployed various member-exclusive features to an existing gaming website. The YT21 creator built a code-base chat application that integrated with GitHub repositories, featuring user authentication, persistent chat histories across sessions, and URL-based navigation to specific conversations. The project applied retrieval augmented generation to answer questions about a user's codebase. In YT22, a developer created a web application to help international students with job applications, incorporating resume tailoring based on job descriptions, user account management, payment processing via Stripe, and file upload functionality.

3.1.2. Do programmers expect complete or partial success with vibe coding, and to what extent is vibe coding exploratory?

Users demonstrated a range of expectations when engaging in vibe coding, from full success to partial achievement to exploratory (Kery and Myers, 2017) use, sometimes with these expectations evolving throughout their process.

In YT15, the user begins with an expectation of full success, seeking to create a diagram because they don't feel they have the skills to make this diagram themselves. The user ultimately adjusts to accepting partial success, expressing satisfaction with reaching "80% of the way,". They also show an emergent exploratory element, as when the project is completed, they express curiosity about the generated code ("how did we do this?") and express the intent to read the generated code to learn from it.

TW1 also expected full success for vibe coding new features to add to their deployed website. Rather than applying vibe coding in an exploratory manner, their approach combined vibe coding to generate code for each feature with manual fixes and programming to accomplish a set of tasks they had collected on a to-do list.

In YT21, the programmer demonstrates strong expectations of full success. They note that they had already made rapid initial progress before recording, reaching "80 or 90% of the way to a working version." Their stated intention to incorporate the application into a larger project and eventually productise it for public use all signalled expectations of complete or near-complete success. While the project being "greenfield" (i.e., the codebase was built from scratch) allowed "more of an opportunity to explore," this exploration occurs within the framework of building a finished application rather than just experimenting with the technology.

YT22 shows a measured expectation of partial success, specifically aiming for a Minimum Viable Product (MVP). The creator discusses monetisation plans, indicating serious intentions beyond exploration. However, they explicitly state that the outcome doesn't need to be "production ready".

3.2. Intentions

3.2.1. How is the initial intention formed for a vibe coding project?

In most cases, creators formulated objectives *before* engaging with AI tools, though the specificity and scope of these prior intentions varied. The creator in YT15 approached the session with the goal to generate a diagram explaining a code agent's workflow, having already conceptualised the diagram's structure and components. The programmer in YT21 began with a defined intention to enhance an existing application by adding support for persistent conversations across multiple sessions, having already decomposed this goal into manageable steps. We observed detailed pre-planning in YT22, where the developer had established not only the core functionality but also the technical stack, authentication method, and monetisation strategy before starting to code.

3.2.2. How and why is intention refined over the course of the vibe coding session?

The iterative, conversational nature of vibe coding supports the refinement of intention through progressive discovery and refinement. Across all videos, the creator's ideas expanded beyond their initial vision as refinements were driven by factors such as dissatisfaction with the initial output and emerging needs or ideas during the process of evaluating the output. In the vibe coding workflow, users must simultaneously form intentions for the artefact they are developing and form intentions for how to communicate those goals effectively. The inevitable mismatch between the user's internal intent and the AI "interpretation" of it becomes a significant driver for refining not only the project's direction but also the user's prompting strategies and their understanding of AI capabilities.

AI contributes to shaping final by filling implementation gaps and suggesting new possibilities. In YT15, the intent clearly evolved through multiple iterations, starting with an initial diagram layout, developing the placement of arrows, then adding styling and animation. The direction of the project was significantly shaped by the back-and-forth with the system as the generated code variously: succeeded in matching the creator's intent directly, thus allowing progress; or succeeded indirectly such as by interpreting vague instructions, thus sparking ideas; or failed, thus necessitating debugging or a goal "pivot" (Drosos et al., 2024).

In one instance for TW1, the AI generated code had an error, but on inspection they realised that there was a bug in the previous code that was revealed by the new code. In another instance, the code generated did not match their original intent, but gave them ideas for how it might be useful for implementing future alternatives.

In YT22, we see a negotiated process where the creator actively evaluated AI suggestions, removing some AI-generated features, like the completion rate box, accepting others, while identifying missing functionalities such as a login button. Similarly, in YT21 testing the application revealed unexpected needs, such as discovering the necessity of a stop button during testing, which arose directly from experiencing the application's behaviour and identifying a usability issue that was not part of the initial plan.

AI tools are also consulted outside of the vibe coding process for guidance and iteration on intent, such as soliciting feature ideas (TW1). Other traditional sources of intention refinement, such as viewing prior art and alternative commercial products as design references, were also observed (YT21).

The blurring of prototyping and production. Vibe coding seems to significantly blur the lines between prototyping and the initial stages of production. Rather than sketch out ideas in text, diagrams, and other traditional low-fidelity media, it is possible to generate functional code very early in the ideation process. The speed at which a basic, runnable application is established suggests that vibe coding might collapse the boundaries between these traditional development phases. This raises several questions, including to what extent the initial "vibe" may set the trajectory for the entire project in a more profound way than traditional prototyping.

Context momentum. Within vibe coding sessions, the history of interactions and the outputs previously generated significantly influence the subsequent direction of code generation. We term this phenomenon *context momentum*. This creates a form of path dependence, where early prompts and the outcomes they yield can steer the project onto a particular trajectory that may become challenging to diverge from later in the session (which is reminiscent of the cognitive dimension of premature commitment (Green, 1989), albeit here it is difficult to characterise this as a property of a notation). Context momentum is the cumulative effect of the model's "interpretations" and the programmer's responses building upon one another to shape the evolving intention for the program.

One example comes from YT18,[2] a project focused on building an exchange rate application, where the user initially requested historical exchange rate data. The model interpreted this request by providing a date picker to retrieve data for a single date. Even though the user's underlying intention was for a feature that would retrieve a range of dates (which was not articulated explicitly in the prompt), the user found the response satisfactory and proceeded. The model's interpretation thus became the established path. When the user later requested a feature to query a date range (this time explicitly) in a different part of the application, the newly generated code, likely influenced by the prior single-date implementation, still fetched data only for a single date. The user believed this was a downstream consequence of the earlier implementation choice, explicitly linking the current difficulty to the prior model interpretation and its resulting code. Thus, while programmers may choose to accept edits that do not entirely align with their intent, but which can still satisfy their requirements, to maintain speed of development, this may result in greater downstream challenges in communicating intent and steering AI outcomes.

Context momentum may also have positive, exploratory outcomes. For instance, YT15's decision to add animations was an impromptu goal that appeared to arise on the basis of the possibilities presented by the existing structure in the generated code. The possibility of the developer formulating such a goal thus appeared to be contingent on a certain set of arbitrary but satisfactory implementation decisions made by the model, and a different set of satisfactory decisions may have led to different improvised goals.

3.3. Workflow

3.3.1. What are the stages of the vibe coding workflow?

The vibe coding workflow consistently begins with an initial goal-setting and prompting phase. In YT15, this involved the creator giving a detailed spoken prompt to Cursor via a transcription tool to generate the initial diagram, establishing a diagram explaining the code agent's workflow as the first goal. Similarly, the creator in YT21 started with a clear goal: to modify his existing 'chat with a code base' application to support multiple chat conversations per repository, which they subsequently decomposed into smaller sub-goals. In YT22, the creator sketched a flow diagram to visualise the tool's functionality and applied Code Guide,[3] a tool that generated a project plan and documentation from his app requirements.

Following initial generation, developers consistently engage in evaluation and iterative refinement cycles. In YT15, after quickly previewing the diagram, the creator notes that it "meets requirements" in terms of basic functionality but "is not very pretty," prompting refinement of specific elements

[2]This video was not part of our final framework analysis, but this example is given here for its clarity.

[3]Available at: `https://www.codeguide.dev/`. Last accessed: 19 September 2025.

(e.g.,"arrows are better, but could be even better"). Throughout TW1, the creator reviewed the model's code changes, and upon discovering any issues prompted the model about what was incorrect and re-generated code. YT22 similarly shows the developer's continuous review of the AI-generated code with manual editing to fix errors, remove unwanted features and adjust functionality. This evaluation phase frequently involved test running the application, such as frequently switching to the browser to test the implemented changes and verify if they met expectations (YT21).

Thus, vibe coding is characterised by *iterative goal satisfaction* (Drosos et al., 2024) cycles, where developers (1) Formulate a goal or sub-goal, (2) Prompt the model to generate code to achieve that goal, (3) Review the generated code, (4) Accept or reject the changes, (5) Test the changes in the application, (6) Identify any bugs or areas for improvement, (7) Refine the prompt or transition to manual debugging and editing. Steps (2)-(7) are repeated until the sub-goal is satisfied or pivoted, at which point the programmer returns to step (1). The entire process is repeated until the programmer decides not to formulate any further goals.

3.3.2. Portfolio of tools and technologies

An ecosystem of interconnected tools is used with AI-integrated development environments at the centre. In YT15, Cursor serves as the primary vibe coding platform, providing an interface for prompting, code generation, and editing, allowing users to switch between models such as o3-mini and Claude 3.5 Sonnet. The creator also uses a transcription tool to convert voice instructions into prompts for Cursor. A web browser is used for previewing and evaluating the generated HTML and JavaScript code.

YT21 demonstrates a similar pattern. The programmer uses Cursor for code generation and editing, switching to a web browser to visualise and test the web application. This programmer actively used browser developer tools (inspector, console, network tab) to identify error messages and check API calls. Additionally, the terminal is used to monitor server-side processes and error messages.

The YT22 creator begins with Excalidraw for initial planning and conceptualization, creating flow diagrams of the intended tool. This is followed by using Code Guide, a meta-level tool that generates project plans and documentation intended to reduce AI hallucinations by pre-processing prompts to introduce structure and detail. Only then does the creator move to Cursor for code generation in agent mode, intentionally configured with specific models like OpenAI's GPT-4o. The terminal is used for project setup and running development servers, while the web browser serves for testing, verification, and accessing documentation.

3.4. Prompting

3.4.1. How do creators set up a vibe coding session for effective prompting?

To help their prompts be more effective, creators modify the system instructions for the code-generating language model, compare different IDEs to evaluate the fitness-for-purpose of their AI integration, compare different code-generating models for their speed, quality, and cost-per-token, and pre-process prompts to reduce hallucinations.

The creator in TW1 takes a formalised approach to prompt customisation, starting with an existing template called *.cursorrules* sourced from GitHub. They incorporate this into their own *Claude.md* file, which serves as system instructions for the code-generating model. They personalise these instructions by specifying the programming languages, frameworks, and libraries used in their codebase.

YT21 reveals more of the implicit preparation required for effective prompting during vibe coding. The programmer selected Cursor specifically for its AI integration after evaluating alternatives like VS Code and Replit. The programmer had already mentally mapped their goal of making the chat interface "more like ChatGPT" and broken it down into smaller steps, starting with "creating an API for a new conversation." Their familiarity with the codebase and pre-selection of Claude 3.5 Sonnet as their model represent forms of preparation, even if not explicitly framed as setup work.

In YT22, the creator uses Code Guide to input requirements and generate a project plan and documentation, which they review and refine by updating version information. This preparation is strategically intended to help the model hallucinate less. They actively switch between Claude 3.7 Sonnet and OpenAI GPT-4o based on considerations like output speed and token costs (explicit experimentation and comparisons of performance were also observed in YT15 and YT21).

Vibe coding does not appear to be dependent upon, or associated with, any particular level of prior development of the codebase. Developers in our dataset variously started vibe coding entirely from scratch starting with an empty directory (YT15), or from a boilerplate starter template (YT22), or *in medias res* for continuing to develop a pre-existing codebase (TW1, YT21).

3.4.2. What explicit prompting strategies are used during vibe coding?

We observed several explicit prompting strategies, such as management of the context, referring to specific code elements by name, structuring instructions into lists, outcomes, and constraints, pasting in examples and error messages, referencing previous output, pre-processing or generating prompts through external tools, deliberately limiting the scope of prompts, and including external documentation as resources.

We observed deliberate management of the AI's working context. When shifting focus to animating arrows, YT15 opened a new "composer" window (essentially, a new chat thread within Cursor) to create a fresh context. Similarly, Before starting new phases, YT22 closed all tabs to "clear the context from the AI".

The YT21 developer frequently provides context through code snippets and refers to specific code elements by name, using names of functions, variables, and APIs to direct the model with precision and granularity. They also structure complex instructions using numbered lists and explicitly state desired outcomes and constraints. They also provide concrete examples, e.g., pasting an example of the expected JSON response from an API to guide implementation. This developer also explicitly referenced the model's previous output in new prompts to provide corrective feedback. Uniquely in our dataset, this developer employed inline prompts for localised changes directly within the code editor (i.e., the smaller-scoped, autocompletion-style of interaction typical of earlier generations of coding assistants) for targeted modifications.

As mentioned, the developer in YT22 began with documentation and implementation plans generated by Code Guide as context for prompting Cursor. TW1 also demonstrates a meta-prompting approach by using one model to generate a prompt for the coding AI agent. The developer in YT22 also adapts community knowledge, using a prompt structure found on Twitter but tailoring it to their specific project context. They explicitly practice scope limitation, focusing on one "phase" of the implementation plan at a time to reduce hallucinations. In another form of scope limitation, they also introduce constraints to prompts such as "don't integrate stripe yet. Just make a design with dummy data", perhaps anticipating and preempting the model's tendency to fill in unstated gaps, or tendency to set an overambitious goal that the model is likely to fail to achieve in a single step.

The developer in YT22 demonstrates further fine-grained guidance methods, particularly by providing external documentation as context, explicitly copying the URL of the official Tanstack Form documentation with instructions to "use docs," employing the model's ability to index external resources. When encountering runtime issues, they supplement prompts with error messages, directly pasting browser errors and asking for fixes.

3.4.3. What is the granularity of prompts and iterations? Do users address a single objective at a time with prompts, or issue multi-objective prompts?

The granularity of prompts in vibe coding spans a spectrum from high-level directives to extremely detailed, low-level instructions, spanning from single-objective prompts, to complex, mixed-objective prompts where different unrelated requirements are expressed simultaneously and at different levels of detail.

In YT15, the developer begins with detailed spoken instruction to establish initial requirements for a diagram, including modules and data flow specifications. When results don't meet expectations, they employ iterative refinement with precise feedback. When trying to prompt the model to produce an animation, they provided a detailed step-by-step description specifying the exact sequence of visual elements they expected to see.

The creator in YT15 uses both precise and imprecise language to convey aesthetics. When requesting arrow repositioning, the creator gives specific directions like "not to use diagonal ones" and that "arrowheads should be pointing to the middle of the boxes." However, the creator also employs imprecise descriptors such as expanding space "significantly," relying on the model's interpretation rather than providing exact measurements. The creator also issues a high-level prompt mentioning a desired theme and colour palette without specifying details, leaving these to the interpretation of the model. Similarly, the programmer in YT22 uses a mix of specific and ambiguous instructions for styling, such as "move the date below and use a very small font" and "don't use bold font, narrow the space between conversations." These prompts combine specific actions with somewhat subjective descriptors like "very small" and "narrow," leaving some interpretation to the model while maintaining directional control.

YT21 shows a similar pattern but with explicit reflection on the strategy. The programmer initially uses very brief, high-level prompts such as simply requesting to "create an API to create conversation." When these broad prompts fail to yield the desired results, the programmer acknowledges, "I should have been more clear here," and shifts to specific, low-level prompts that target particular functions, variables, and styling elements. The programmer also employs extremely fine-grained iterations for localised changes, such as using inline prompts to modify specific parts of functions. The programmer's statement that they "start by asking very dumb questions. And if it gets confused, [they] start providing more detail" reveals an intentional approach to granularity adjustment.

The developer in TW1 attempts to keep prompting succinct yet detailed, but ultimately finds longer prompts to be better so as not to give the model "a lot of leeway to go off the rails." This developer strikes a balance between high-level directives (e.g, "Increase the exp for VIP members to 1.3x on the back-end") while providing specific implementation details when necessary (e.g., which files to edit, and specific callouts about variables or code types within these broader requests).

YT22 intentionally modifies high-level prompts to focus on smaller chunks ("just do phase one") to reduce hallucinations, revealing a mental model of how prompt granularity affects AI performance. Their debugging prompts were consistently single-objective (pasting specific errors with requests to fix), and even feature requests focused on particular elements like "Make a page with all resumes that run through the optimization" rather than requesting multiple features simultaneously. YT22 also reveals situations where extremely fine-grained prompting may be deemed inefficient; the creator often makes manual edits rather than prompting for minor adjustments.

3.4.4. What is the role of different input modes such as voice, text, and images, in vibe coding prompting?

Despite the emphasis on voice input in the Karpathy canon, typed text is the primary mode of prompting, but we do observe instances where transcribed speech and images are used as well. Typed text itself may also contain a mix of elements, such as instructions, copied and pasted error messages, and links to documentation (as previously described).

In YT15, the developer primarily uses voice prompting via a transcription tool, though they note the ingrained habit of typing prompts, and consciously try to stay in voice modality, citing the desire to adhere to the Karpathy canon. Despite this, they still switched to text input at times. They attempt to use a screenshot to better communicate the desired changes, though this was unsuccessful due to limitations of the model they had selected at the time (we did observe successful use of screenshots in other videos in our extended corpus).

In contrast, YT21 and YT22 dominantly use typed text. YT21's creator relied virtually exclusively on text-based prompts within Cursor's composer window and inline chat feature, with no evidence of voice or image inputs. Similarly, in YT22, text is the primary mode of input with the creator typing new prompts and modifying existing ones through typing.

Code reuse strategies appear consistently across the examined videos, naturally through textual means. In YT21, the creator frequently copied and pasted entire function bodies into the composer window to provide context for their natural language instructions. Similarly, the creator would copy error messages from the terminal or browser console into the prompt window. The inclusion of expected output formats also appears as a code reuse strategy in YT21, where the creator provided an example of the expected JSON response to guide the model.

Notably absent from these observations are examples of creators directly borrowing code from online sources like Stack Overflow or GitHub to guide the model, though the practice of sharing documentation URLs in YT22 suggests that external resources do play a role.

3.5. Debugging

3.5.1. What debugging strategies are applied during vibe coding?

Across the observed vibe coding sessions, developers employ a blend of traditional debugging techniques and AI-assisted approaches. The debugging process typically begins with visual inspection of the code and test running the application. This initial assessment is followed by more detailed debugging strategies depending on the nature of the issues encountered.

AI "hallucinations" and failures to fully adhere to prompt instructions can create errors. In YT21, the model sometimes generated code with non-existent properties, requiring the developer to identify these issues through error messages and code inspection. The developer in TW1 expressed frustration when generated code did not adhere to the system instructions they defined, noting that it exported a function when it didn't need to. In YT22, the model produced documentation referencing Next.js 14 and JavaScript files when the creator was using Next.js 15 with TypeScript, requiring manual updates.

In YT15, the creator primarily relied on identifying visual discrepancies between the desired outcome and the generated output. For example, they noticed issues such as unsatisfactory arrows in diagrams, a canvas that was too small, and overlapping arrow labels. This approach is particularly relevant for graphical or UI-focused projects, where the correctness of the code is largely judged by its visual output rather than its internal logic.

More technically sophisticated debugging approaches appear in YT21 and YT22, where the programmers heavily utilised browser developer tools. In YT21, the programmer examined error messages in the browser's console and network tabs to pinpoint issues and verify API calls. Similarly, in YT22, when encountering a blank screen, the creator immediately opened the browser's developer console, identifying a "trpc error". Terminal analysis was also observed in YT21, with the programmer monitoring terminal output alongside browser tools.

A distinctive feature of vibe coding is the use of the AI itself as a debugging tool, or perhaps more accurately, to bypass the manual debugging process entirely. In YT22, the creator copied error messages directly from the browser console and pasted them into the Cursor chat prefixed with prompts like "Please fix it" or "Refer the docs to fix this error." Similarly, in TW1, the creator asked the AI to "find and debug the issue and fix it."

Creators frequently employ iterative refinement through targeted prompting. In YT15, the creator provided detailed instructions for repositioning arrows, fixing overlapping labels, and adjusting animations. This strategy of using natural language prompts to guide debugging is also observed in YT22, where instead of directly modifying code, the creator issued prompts such as "I don't need the function of pasting the URL of job description. Please use the text approach only." TW1 employed a strategy of resubmitting a prompt after iteration on the system instructions to guide the model to produce results in line with their preferences.

Creators still engage in manual code review and editing. In YT21, the programmer frequently scanned and read the AI-generated code to assess alignment with expectations and identify potential issues before testing. Similarly, in YT22, the creator frequently examines the generated code, scrolling through files and looking at diffs to understand the logic of the AI output. In TW1, the creator inspected the AI-generated code to check if it was viable, and in some cases if the output was "close", they accepted the code with the intention to fix it manually.

Users often exhibited mixed manual-automated debugging strategies involving hypothesis formation and strategic prompting. In YT21, when errors occurred, the programmer formulated hypotheses about the cause based on error messages and observed behaviour, then verified these hypotheses through browser or code inspection. This was followed by specific and targeted prompts containing instructions on how to fix identified issues, such as prompting the model to "pick the repo_ID from the URL".

Model switching is occasionally used as a debugging strategy. For instance in YT15 the creator switched from the o3-mini model to Claude 3.5 sonnet when trying to animate arrows. This suggests that creators develop mental models of the relative capabilities and proficiencies of different language models.

Users sometimes acknowledge and deliberately ignore issues, as in YT15, where the creator explicitly chose not to address a compiler warning ("do not use empty rule sets").

The special nature of code reading in vibe coding. A striking aspect of how programmers engage with AI-generated code in vibe coding, particularly during debugging, is the speed and high-level nature of their code inspection. Rather than line-by-line code review, we observed programmers rapidly moving from point to point, performing what could be described as impressionistic scanning rather than a linear read. This allows for a rapid, almost immediate assessment of whether the generated code "looks all right", "meets requirements", or is "exactly what I asked for" (YT15).

This speed is facilitated by several key techniques. Programmers pay close attention to the visual code diffs presented by the IDE (e.g., Cursor), which highlight additions in green and deletions in red. They can quickly assess the volume of changes by looking at the size of the red and green highlights and make a decision to accept changes without too much careful scrutiny if the overall shape and size of the changes seem acceptable, or reject them if not. In one instance, the programmer accepted a large number of diffs within two seconds because they could immediately tell it was what they wanted (YT21).

Programmers are drawn to specific function calls and the code comments written by the LLM (e.g., YT21). They look for key indicators of task success, such making sure that the right API is being called or the right identifiers are being used (YT21), inspecting lines of code in more detail if something catches their eye during this rapid scan. Especially with structured code like HTML or React components, programmers can quickly assess the overall structure.

Experienced programmers can look at AI-generated code and intuitively sense if it's using the right level of abstraction. Observing the AI writing low-level HTML/JS/CSS primitives for diagrams, the programmer immediately recognised the pattern and felt like they were "reinventing the wheel", speculating that there "should be some libraries that do these things" (YT15).

The ability to see beyond the immediate AI suggestion and understand its implications across the project is a critical use of expertise. Expertise allows programmers to cross-reference AI-generated code with other parts of the codebase or external documentation. They might check a specific file to ensure the AI used the correct API path (YT21) or cross-reference parameter names used by the AI against functions in different files (YT21, YT15).

The expert ability to mentally visualise the finished product *through* the code without needing to test run the application also supports the rapid scanning of AI generated code for verification. For example, YT21 repeatedly glanced at HTML to ensure generated code appeared in the right section, immediately spotting where the data is going to be displayed or where the title is going to be displayed simply by looking at the code structure.

Moreover, the process of rapid scanning serves for more than simply allowing programmers to react to potential errors in the current round of AI-generated edits. In reading AI-generated code, programmers also exhibit a *proactive stance*. Programmers consciously prepare themselves for reviewing subsequent cycles of AI output. By familiarising themselves with the overall structure of generated code as it is generated, noting key identifiers, components, and implementation decisions made by the model, they prepare themselves to assess *future* outputs and diffs, maintaining their “glanceability” as the codebase grows. They might reference API documentation or component structures to know what the model *should* produce in certain areas (YT21, YT22).

A related proactive behaviour is the programmer’s ability to hypothesise about situations such as needing to make changes to an additional related file, even though Cursor hasn’t suggested it directly (YT21). This requires an understanding of the codebase’s interconnectedness and dependencies that are not direct semantic dependencies, but rather contained in the programmer’s mental schema, and therefore invisible to the code-generating model.

3.6. Challenges

3.6.1. What challenges besides code bugs do programmers encounter while vibe coding?

Practitioners face several challenges when engaging in vibe coding beyond traditional code bugs. These challenges span technical, conceptual, and workflow dimensions.

One challenge is communicating visual and abstract ideas. In YT15, the creator struggled with articulating what they wanted to do with the arrows, encountering the fundamental difficulty of trying to talk about diagrams using words. Complex conceptual goals can also be difficult to articulate as prompts, as seen in TW1 where the developer faced difficulties forming the prompt to describe a matchmaking feature, asking “how is [AI] going to give me what I want if I don’t even know?”

Understanding AI capabilities and limitations is another challenge. In YT15, the creator expressed uncertainty about whether a specific AI model (Claude 3.5 Sonnet) could create an animation as desired and later switched to a different model (o3 mini) after an unsatisfactory experience. In TW1, the creator felt that some models were better at certain workflows, indicating that matching the right AI capability to the right task requires experience and knowledge with the models. TW1 also stated that “LLMs will get you most of the way there, but they don’t get you all the way there”, perceiving universally inherent limitations in AI capabilities. YT22’s need to include specific documentation for Tanstack Form to ensure proper integration also demonstrated that the model’s “general knowledge” isn’t always sufficient for specialised frameworks.

Tool-specific limitations also present challenges. In YT15, the creator didn’t know how to revert changes made by Claude and struggled to navigate Cursor’s interface to find a checkpoint. In YT21, the programmer faced a situation where Cursor displayed generated code changes as a chat message (rather than edits in the file), making integration difficult.

Deciding when to rely on AI versus manual intervention (discussed further in Section 3.7.3) introduces new metacognitive load (Tankelevitch et al., 2024). In YT21, the programmer frequently reviewed and sometimes rejected AI-generated code, stating that reviewing changes helped them remain in control. In YT22, the creator frequently switched between prompting and making manual edits without clear criteria for choosing one approach over the other. The developer in TW1 observed that for small adjustments, they felt faster working manually, but for more substantial edits, they preferred using AI assistance.

3.6.2. How are these challenges addressed?

Iterative refinement of prompts is almost inevitably the programmers’ first recourse when encountering communication challenges with the model. In YT15, when the creator struggled to articulate visual intent for arrow placement, they iterated through multiple prompts with increasingly detailed spoken instructions. Similarly, in YT21, the programmer recognised when initial prompts were too vague and provided more detailed and specific prompts in subsequent attempts, often including numbered lists of requirements and referencing existing code snippets. When the model "did not listen to the system instructions", TW1 refined the system instructions to emphasise acceptable behaviour; a higher-order strategy than refining individual prompts.

At one point, the programmer in TW1 adopted an unusual prompting strategy where the developer responded to an unsatisfactory result by prompting "Bro, you are losing aura with this. Come on." before pasting the original prompt and resubmitting. This explicit "vibification" produced more code than the previous turn, though the programmer in TW1 still expressed dissatisfaction with the code quality. This suggests that some vibe coders experiment with social or emotional appeals, albeit perhaps tongue-in-cheek, in prompts. We also observed similar styles of prompting in other videos in our broader corpus. The results of such appeals, as might be expected, are mixed.

Developers strategically switch between different AI models based on their perceived strengths. In YT15, the creator actively switched between OpenAI's o3-mini and Anthropic's Claude 3.5 Sonnet models based on their qualitative assessment and the perceived strengths of each for different aspects of the task, using one for animation and another for initial code generation. Similarly, in TW1, the developer discussed where they used models from the Claude family to generate code for a web page, but switched to OpenAI's GPT 4.5 to rewrite the text because they believed its text quality was an improvement over Claude.

3.7. Expertise

3.7.1. When and in what ways do experts deploy their expertise in vibe coding?

Expertise is consistently deployed throughout the entire vibe coding process, though in ways that differ from traditional programming.

In the initial phases of projects, experts apply their knowledge to select appropriate tools and models. In YT15, the creator demonstrates expertise when evaluating different AI models based on their capabilities for diagram creation. Similarly, in YT21, the programmer makes a deliberate choice to use Cursor over the Visual Studio Code or Replit IDEs based on an expert assessment of their affordances.

During code generation and review, expertise manifests in rapid evaluation and error detection. In YT15 the creator applies their expertise to rapidly assess HTML code visually. Similarly, in YT21, the programmer consistently reviewed the code for key elements and potential issues, making immediate judgments about the correctness and suitability of the suggested code (as described in Section 3.5.1), demonstrating a high level of familiarity with programming languages and coding patterns.

Expert knowledge also guides problem identification and solution development. In YT22, upon encountering a blank screen, the creator quickly inspects the browser console and identifies a 'trpc error,' demonstrating expertise in reading and interpreting error messages. In YT21, the programmer displays expert knowledge of debugging techniques by immediately recognising database issues from error messages, effectively using browser developer tools to diagnose problems, forming hypotheses about root causes, and inserting print statements to trace code execution. YT21 also demonstrates understanding of API endpoints, data structures, and how different parts of the application communicate, allowing them to identify issues with API calls. Similarly, in YT22, the creator identifies that an application is trying to connect to PostgreSQL instead of Supabase and knows exactly how to manually edit environment files to correct the configuration. In YT22, the creator evaluates AI-generated code against expected standards by checking generated React components against documentation, demonstrating and applying their knowledge of UI libraries.

Expertise guides quality control and feature alignment. In YT22, the creator recognizes and decides to remove unwanted AI-generated features that, while not buggy or intrusive, don't align with the intended design. In TW1, the creator demonstrates understanding of when generated code might incur technical debt, which "lowers developer velocity", and addresses these issues proactively.

Finally, experts also deploy their knowledge to determine when to transition between AI guidance and manual editing (discussed in more detail shortly in Section 3.7.3).

Thus we observe that expertise is not replaced but rather redirected: from writing code directly to evaluating, guiding, and refining AI-generated solutions. The expert assumes more the role of director, reviewer, and editor than a line-by-line author, but their technical knowledge remains essential throughout the entire development process.

3.7.2. What kinds of expertise are used in vibe coding?

As previously mentioned, all the videos we studied featured programmers clearly in possession of expertise in software development. YT21 leverages a range of expertise including specific technical knowledge of databases and programming languages, but equally important are their debugging skills, understanding of API calls, and software architecture concepts. Similarly, YT22 displays technical knowledge of their stack (Next.js, TypeScript, etc.). They make informed decisions about which libraries to use based on project needs and can identify when AI suggestions align with or deviate from requirements.

Beyond "traditional" programming expertise (code quality and maintainability, fault localisation and debugging), the programmers we studied exhibited expertise in two additional domains: AI expertise (understanding models, prompting practices, concepts such as context windows, and limitations of AI code generation), and product management expertise (forming and translating goals into features for the product). Perhaps as a consequence of the blending of prototyping and production (as discussed in Section 3.2.2), vibe coding requires (or at least is greatly facilitated by) a broad skillset spanning traditional programming skills, AI literacy, and product vision.

Across these videos, we see creators combining traditional coding knowledge with adaptation to new tools and workflows. Effective vibe coders appear to be those who can fluidly move between writing/editing code directly, formulating effective prompts for AI tools, evaluating AI outputs, and understanding the broader product and user requirements, suggesting both technical foundations and a "technology-forward" mindset are valuable components of expertise in this context.

Moreover, we speculate that these programmers are also developing a sense of what might be called *ambient competence*. This is a sense of competence wherein the programmer feels capable of tackling tasks they wouldn't have considered before, because they have access to an AI system that may be able to accomplish the task with very little effort invested, regardless of whether they themselves have the skills to tackle them manually (which they very well may). This can be thought of as the competence that arises as the result of the active fulfilment of the "awareness of the possible" (Sarkar, 2023c; Sarkar, 2023b). This could lead to a shifting locus of agency, where the programmer's confidence and the scope and complexity of projects they undertake are increasingly defined by their perceived ability to prompt and guide a model, rather than their own direct coding abilities.

3.7.3. When do practitioners transition from vibe coding to manual work?

Developers transition from AI-assisted vibe coding to manual work under several distinct circumstances and for several reasons. They might choose to work manually for efficiency, debugging, refinement, and specific episodes where the model is difficult to "steer". The transition to manual work during vibe coding is not merely a function of technical necessity but also reflects individual preferences, expertise levels, and expectations for human-AI interaction.

For straightforward edits where the overhead of prompting exceeds the effort of making the change directly, developers often opt for manual work. In YT21, the programmer attempted an inline edit with AI but then decided to perform the simple one-line edit manually after determining that AI assistance was unnecessary for such a minor change. In TW1, the developer articulates their decision-making process about when to work manually versus using AI assistance. In one instance, they switch to manual work when they believed it would be a more efficient workflow, noting that "IntelliJ's autocomplete and search is still faster than LLMs". This leads them to "race the AI" to find code locations, sometimes stopping AI generation mid-process when they locate what they need faster.

Another common transition point across videos is when AI generates a mostly-complete solution that requires refinement. This can be viewed as a special case of transitioning to manual work for efficiency. For example, TW1 accepted code when the AI "got close" to their goal and then manually edited the generated code to be more inline with their preferences for code. TW1 describes their viewpoint that LLMs should be used to "get close to what you want" but that expecting perfect outputs is "a mistake" – instead, their optimal workflow involved making post-generation adjustments to generated code, and reuse of generated code in similar features by quickly modifying code to other contexts rather than issuing refined prompts.

Similarly, in YT22, the creator performs manual edits to remove unwanted UI elements, delete redundant buttons, and make small edits in the code to better align with their vision. For instance, the creator manually removes functionality for uploading job descriptions via URL to focus solely on text-based inputs, representing a conscious design decision that required direct code manipulation.

Debugging is as a common trigger for manual intervention across multiple videos. In YT21, the programmer switches to manual work almost immediately when encountering errors, using traditional debugging techniques like browser developer tools, console inspection, and hypothesis formation. In one instance, they hypothesised a solution based on an error message and implemented a fix without AI assistance. Throughout, YT21 frequently engages in manual code review and rejection of AI suggestions based on their understanding of correctness and project requirements.

YT22 shows similar patterns, with the developer manually editing environment files to fix database connection errors and directly intervening to correct runtime issues. YT22 also manually corrects a misalignment in AI-generated documentation (changing "Next.js 14" to "Next.js 15") and deletes redundant steps in implementation plans. These examples suggest that when errors occur, developers often rely on their expertise and traditional debugging approaches rather than AI tools.

However, YT15 presents a contrasting case where the developer states explicitly that they are "not going to try and fix it on his own" when encountering an error, citing adherence to the Karpathy canon. The creator in YT15 deliberately avoids manual debugging and relies almost exclusively on re-prompting for changes throughout their workflow. Thus, the videos do exhibit individual differences in approaches to manual work.

3.8. Trust

3.8.1. What is the nature of trust in vibe coding and how is it developed?

Trust in vibe coding appears to be granular, dynamic, contingent on review, and evolves through interaction with the system, manifesting as a tension between efficiency and comprehension.

In YT15, we see a developer who demonstrates initial trust by accepting AI-generated code with minimal inspection. Trust further develops through subsequent successful outputs. When the model successfully generates complex animations, the creator expresses surprise ("It would feel a little bonkers if this actually works") followed by reinforcement of trust when it succeeds ("wow, [...] that's exactly what I asked for"). This indicates that dramatic positive outcomes can significantly strengthen trust in AI capabilities.

YT21 presents a practitioner with what appears to be a well-calibrated trust mindset. This developer articulates that they can "quickly get 80-90% of the way there with AI". However, this trust is contingent, as they explicitly state, "I like to review the changes because it helps me remain in control to some extent." The risk of overtrust is actively mitigated as the developer frequently rejects AI suggestions that use non-existent APIs or approaches they considers less maintainable. Frequent (albeit lightweight) code review was observed across all participants. Thus, reviewing changes appears to be important for vibe coding, not only for maintaining understanding of the code, but also for maintaining agency, authorial ownership, and trust.

YT22 further reinforces this notion of contingent trust. The creator explicitly states that they don't believe in "blindly following the AI", stating the importance of reading and reviewing the code. They further state: "AI is just a tool." This invocation of the tool-user relationship can be viewed as being in explicit contrast to delegation of responsibility. The developer consistently reviews code, compares it to documentation, and tests frequently, showing that trust is continuously re-established rather than statically assumed.

TW1 illustrates the nature of trust in relation to expertise. Their expertise enables the recognition of "way too complicated code" generated by the model and enables them to fix it, while speculating that "the average vibe coder might not be able to fix that" without expertise such as TW1 possessed. This suggests that trust in vibe coding may be significantly influenced by the user's own technical expertise. When discussing a complex codebase of "150 thousand lines of good code" the creator emphasises that they cannot allow "bad" code to enter the repository. Even on "simpler" features, such as an information page that describes the benefits of a VIP membership to the website, TW1 still notes they will review the pull request before committing and deploying to the website. Interestingly, this disposition was challenged by a viewer of TW1's twich stream, saying that "code review is not vibe coding", which suggests a tension between trust and verification in the vibe coding paradigm.

Users develop trust in AI-generated outputs through a process of experimentation, verification, and adaptation. The risks of overtrust emerge when users fail to critically evaluate AI outputs or lack the expertise to identify problematic code. Across these videos, we see that developers, for the most part, appear to successfully avoid overtrust. They maintain a critical stance toward AI outputs, verify results through testing and review, and possess sufficient expertise to identify problematic code.

3.9. Defining and performing vibe coding

3.9.1. How do programmers define vibe coding?

Vibe coding thus appears to be a programming approach characterised by conversational, iterative interaction with AI tools, where the model handles significant portions of the coding work. TW1 directly defines it as having "the AI do the heavy lifting" and describes the workflow as having the AI creating a feature and then "spruce it all up." In YT21, the programmer describes vibe coding as "just chatting with the app, saying do this, do that".

TW1 presents vibe coding as flexible and creative, likening it to "a Bob Ross painting" where "you can do what you want" when referring to the effort a vibe coder spends on the workflow, suggesting that the defining characteristic may be freedom and creative flow rather than the use of specific tools. For planning activities, YT15 demonstrates that vibe coding can encompass emergent intent where requirements evolve through interaction with the AI. The programmer introduces new requirements like arrow labels and animations after seeing initial results, planning fluidly throughout the process rather than strictly beforehand. This suggests that even when not strictly in an exploratory programming setting (Kery and Myers, 2017), vibe coding is accommodating of an exploratory element.

Based on these observations, vibe coding appears to occupy a higher position on the spectrum of AI reliance in programming activities than "traditional" AI-assisted coding (e.g., as with the initial generation of GitHub Copilot as documented in Sarkar et al. (2022a) or Barke et al. (2023)). It goes beyond using AI for discrete tasks or code completion, involving significant delegation of code creation and modification to AI systems. However, it is not entirely hands-off. As seen in YT21, programmers may rely on AI for code generation but readily switch to manual debugging and problem-solving when necessary. Similarly, YT22 emphasises that human oversight remains important, with the programmer insisting on reading and reviewing the AI-generated code.

The position on this spectrum also appears to vary based on individual preference and project phase. TW1's comment: "don't review just submit" (albeit humorously, as TW1's workflow relied heavily on reviewing AI-generated code), suggests some practitioners might push toward greater AI reliance, while others, like those in YT21 and YT22, maintain more balanced approaches. YT15 shows that programmers might be more willing to accept AI-generated code quickly in early stages but become more critical as development progresses. YT22 initially describes vibe coding as when "you don't really dig into the code, you just embrace the AI," suggesting minimal human intervention. However, the same programmer later clarifies that they "don't believe in blindly following the AI" and emphasises the importance of reading and reviewing code. This contradiction reflects the evolving and personal nature of how vibe coding is defined.

3.9.2. How does the performative aspect of vibe coding on streaming platforms influence the practice?

Content creation for streaming platforms (YouTube and Twitch) impacts how vibe coding is presented and practiced. The observations below are important to bear in mind when interpreting our results, as while we are attempting to characterise the phenomenon of vibe coding in general, an important aspect potentially affecting the generalisability of our findings is the fact that the videos in our dataset were *performed* for an online audience, which is not a property likely to hold for most vibe coding.

Creators frequently emphasise the impressive capabilities of AI through enthusiastic reactions, with YT15's programmer repeatedly saying "wow", such as when the AI-generated code for animations worked correctly. In YT21, the creator explicitly highlights that "because of the capabilities of AI and Cursor, [the creator] was able to accomplish [building the app],". Creators emphasise the ease and speed of vibe coding (e.g. "you can develop this code even without touching a keyboard", "boost my productivity"), invoke rhetorical flourishes like "trust the vibe" and call the results "awesome" or "incredible". Even obvious failures may be framed positively (e.g., in one instance a creator hypothesised that transcription errors force the model to "think more broadly", becoming a potential asset). These reactions amplify the perception of AI effectiveness for audiences, potentially to the point of exaggeration.

Streaming workflows introduces unique dynamics, as seen in TW1, where the creator frequently responds to chat questions and comments which can cause tangents commonly found in streaming workflows (Drosos and Guo, 2021) including playing a game on their website because a viewer mentioned wanting an specific achievement. These interactive elements generate spontaneous content that wouldn't occur in private coding but can yield valuable insights, as when they explained that "the context window is big now so [they don't] have to worry about [token usage] as much" in response to a viewer's concern about instruction length.

The performance context sometimes encourages procedural shortcuts for narrative flow, as seen in YT21 where the programmer states, "I could test this API, but let's just wing it," demonstrating a desire to maintain video momentum. Similarly, YT22's creator structures development around video-friendly milestones, declaring "phase one is successfully done" at a suitable point for a video segment and explicitly mentioning video length as the reason for stopping at certain points, artificially dividing the development process to align with content creation needs.

Despite the emphasis on showcasing AI capabilities, the performances still demonstrate that realistic workflows include debugging and manual intervention (as elaborated in Sections 3.5 and 3.7.3). Moreover, creators often verbalise their understanding and expectations when they are uncertain. Thus, this running commentary might not just be purely directed at impression management (discussed later in Section 4) but also a way for the creator to maintain a sense of control and momentum during development. By narrating the process, they might be developing and iterating on their own mental model.

4. Discussion

4.1. How vibe coding differs from previous generations of AI-assisted programming

The initial phase of AI-assisted programming tools, exemplified by early analyses of tools like GitHub Copilot, marked a significant shift in how programmers interact with code generation. Research from this period, such as the studies by Sarkar et al. (2022a), Vaithilingam et al. (2022) and Barke et al. (2023), documented how the programmer's experience changes in response to AI capabilities like code completion based on context and comments. The subsequent generation of tools expanded upon these capabilities by evolving the scope and size of code generation within a codebase, and allowing the language model to access other tools that expand the scope of their operations (e.g., running terminal commands to create files, start servers, install packages, etc.). This shift in capabilities is sometimes referred to as making these tools more "agentic". Correspondingly, the experience of vibe coding, as observed in our videos, builds upon the experience of prior generations of AI-assisted programming, but introduces distinct nuances in philosophy, workflow, and the programmer's engagement with the AI.

A primary difference lies in the philosophy guiding the interaction. Vibe coding often embraces a more trusting, hands-off approach, prioritising flow or "vibe" over strict control. The explicit inspiration from the Karpathy canon encourages relying on the LLM to autonomously handle errors and work around difficult problems, moving away from traditional debugging methods. This introduces a workflow where the programmer may intentionally allow the AI to "drive" more consistently, even for familiar tasks, creating an fluid interaction style that is distinct from the segmented modes previously identified (Barke et al., 2023).

Prompting takes on a new character in vibe coding. Earlier research noted the difficulties users faced in effectively communicating intent and matching their abstraction level to the model's capabilities, often relying on explicit comments (Sarkar et al., 2022a; Barke et al., 2023). Vibe coding appears to complicate this, featuring prompts that are often significantly longer and exhibit extreme mixing of granularity. Vibe coding prompts can blend vague aesthetic desires, broad ideological goals (like "make it 10X better"), specific technical constraints (like using a particular library or file location), and may even incorporate direct code snippets or external documentation links for context. Early tools primarily relied on comments or explicit pop-up input fields for prompting (Barke et al., 2023), while vibe coding integrates voice, text, and potentially other modalities in a fluid, less structured manner.

The process of evaluation and debugging also shows differences. While earlier AI-assisted programming required validation through examination, execution, documentation lookup, and static analysis (Sarkar et al., 2022a; Barke et al., 2023), vibe coding prioritises rapid, targeted inspection. As detailed in Section 3.5.1, programmers quickly scan code diffs, looking for familiar patterns or keywords (eyeballing, glancing) rather than performing line-by-line reviews. Crucially, traditional manual debugging skills remain essential in vibe coding. When errors occur, vibe coders frequently revert to conventional methods: analysing error messages, using browser developer tools or the terminal, and forming their own hypotheses about the bug and its solution. The model might then be tasked with implementing the specific fix identified by the programmer, although sometimes raw error messages are simply provided to the model for repair. This workflow where the human diagnoses and plans, and AI executes the fix, contrasts with the user experience of understanding and repairing AI code noted in earlier studies (Vaithilingam et al., 2022). It also differs from the explicit validation strategies like detailed examination and documentation lookup observed more prominently in the exploration mode of earlier tools (Barke et al., 2023).

The programmer's expertise is critical in both paradigms, but its application is different in vibe coding. Early studies found that experienced users spent more time in "acceleration" (generating code to fulfil a well-formed intention) (Barke et al., 2023). In vibe coding, expertise is vital not just for coding but for strategically interacting with the AI and managing the broader development environment. Experts apply their knowledge to select tools and models, configure projects, perform rapid visual code assessments, manually diagnose complex bugs when the AI fails, and strategically decide when to switch between AI-driven and manual work. They can quickly identify redundant or unwanted AI output and synthesise information across multiple files or external tools like documentation or terminal output. This level of expertise is less about writing code line-by-line (as might be accelerated by earlier tools) and more about orchestrating the model and other tools within a complex development environment (what Lee et al. (2025) term "task stewardship").

Vibe coding is thus an evolution of first-generation AI-assisted programming, that leans into the conversational and generative power of large language models, while still requiring significant human expertise and judgment. It represents a move from code-generating models as an advanced autocomplete or search tool to a more integrated and capable development tool. Previous work had already identified the shortcomings of analogies between AI-assisted programming and programming via search-and-reuse, or compilation, or programming by specification (Sarkar et al., 2022a); these analogies are even weaker in the case of vibe coding.

4.2. A gestalt theory of vibe

What is the nature and function of the "vibe" in vibe coding? This contemporary colloquialism is a word meaning "the mood of a place, situation, person, etc. and the way that they make you feel",[4] "a distinctive feeling or quality capable of being sensed".[5]

We speculatively posit a connection between the vibe of vibe coding to Gestalt psychology (e.g., Kaldis (2013)) by focusing on the holistic perception and emergent understanding that characterise both. Vibe coding, in its rapid, iterative nature, encourages a programmer to perceive the AI-generated commentary, code, and agentic actions taken within the IDE as a whole, relying on a continuous "vibe check" that corresponds to the gestalt principle that sensory experience of the world is structured as organised wholes (as opposed to parts).

In vibe coding, prompts are often high-level and describe the desired outcome rather than the specific implementation (shifting the focus of programmer intention and articulation towards the whole rather than the parts that compose it). The model then generates code, which, in conjunction with the running application and the context momentum (Section 3.2.2) that builds through iteration, can be seen as an emergent gestalt.

The practice of quickly scanning and evaluating AI-generated code can potentially be accounted for through the law of prägnanz (good gestalt), which states that people tend to interpret their sensory experience using heuristics that render their experiences as structured, regular, orderly, symmetrical, and simple. A positive vibe suggests a well-formed and understandable gestalt. Conversely, a negative vibe might signal a lack of coherence or unexpected elements. As we have seen, vibe coding still requires significant human expertise, manual intervention, and critical evaluation of both code and results, especially when the vibes are decidedly off.

As detailed in Section 3.5.1, programmers spend less time deeply understanding every line generated by the model and more time validating its output against their high-level expectations and mental model. The ability to do this appears to be profoundly dependent on the programmer's existing expertise and well-established mental schemas of coding patterns, frameworks, and their specific codebase. This expertise goes beyond the expertise required to write code manually. We posit that this is expertise in understanding the gestalt of *the code within the context of the conversational session* with the model. Programmers repeatedly exhibited the ability to look at a block of code and immediately identify its purpose within the overall program, and identify potential problems or inefficiencies quickly. This is possible because the code conforms to a mental schema that the programmer possesses, and is capable of rapidly updating as the code evolves, so that the vibe gestalt continues to be useful for future steering and verification. This expert schema, unlike in traditional programming, goes beyond the well-documented expert ability to read and synthesise a code base quickly, because it incorporates several novel constituent elements in the gestalt: the AI-generated commentary, the conversational chat history and context momentum, and knowledge and prior expectations about model capabilities.

4.3. The consequences of material disengagement from code

Vibe coding workflows exhibit the phenomenon of material disengagement from the traditional material substrate of programming: code. Programmers step back from directly manipulating the code itself, instead using AI tools as intermediaries to generate and modify large sections of the codebase. AI

[4]https://dictionary.cambridge.org/dictionary/english/vibe
[5]https://www.merriam-webster.com/dictionary/vibe

assumes the role of handling the tedious material manipulation of code. This fundamentally changes the programmer's relationship with the textual material of the program.

Vibe coding represents a reorientation of material engagement towards the AI tool itself as a mediating entity. Importantly, the AI's generated code, commentary, and errors actively facilitate the formation and refinement of the programmer's intentions, which evolve iteratively from initial ideas as the session progresses based on this dialogue.

This can be understood in terms of Material Engagement Theory (MET) (Malafouris, 2019), which posits that the mind is not an isolated internal entity but is fundamentally constituted through its engagement with the material world. Thinking, in this view, is often best understood as "thinging": a process of thinking primarily with and through things, where the reciprocal interaction between mind and material actively shapes cognition and the development of skill and intention. Examples like pottery making illustrate this, where the interaction with the resistances and affordances of clay is inseparable from the potter's thinking and the development of skill and intention.

From the MET perspective, the AI tool within the vibe coding environment could be interpreted as a new form of "thing" or mediating material. The programmer's cognitive process would thus be enacted not by manipulating code syntax, but by engaging with the AI's interface, formulating prompts, and evaluating the AI's material output (the generated code and its behaviour). This interpretation of AI-as-thing suggests that vibe coding involves a shift in the object of material engagement (from code to AI) rather than its complete abandonment, with the programmer's expertise now oriented towards navigating this new material substrate.

What is potentially lost in this disengagement from direct code manipulation, of course, is the deep, enactive understanding and skill formation that arises from grappling firsthand with the material resistances inherent in code's syntax, structure, and debugging challenges in a manual workflow. Dialogue with the code's own constraints and affordances (indeed, the cognitive *dimensions* that arise as a consequence of manipulating a *notation*), like the potter's interaction with clay, is diminished or altered when mediated by an AI agent.

Nevertheless, the iteration central to vibe coding demonstrates how engagement with the model's material responses acts as a form of resistance that actively shapes and refines the programmer's intentions and strategies. The unexpected outputs, errors, or failures to meet requirements function as the resistances or affordances of this new mediating material, pushing back against the programmer's initial intent and necessitating adjustments or refinements. While vibe coding workflows often seek to minimise friction, the inherent pushback from the model's outputs, and the programmer's active evaluation and response to these, constitutes a form of material engagement that shapes intentions.

This dynamic aligns with the concept of designing "productive resistances" (Sarkar, 2024) in AI tools to cultivate intention, counteracting tendencies towards weakened intentionality, and thereby mechanised convergence (Sarkar, 2023a) of outcomes. Intentionally designed productive resistances could formalise and enhance a process already occurring organically within vibe coding workflows.

Moreover, our analysis finds that code itself does not cease to be a material substrate in vibe coding. Programmers still engage with the code material, but the nature of this engagement shifts profoundly, involving new material manoeuvres: from rapid, high-level review, scanning diffs, identifying patterns, relying on gestalt properties and visual cues, to inspecting keywords and structure to verify correctness or diagnose issues, and indeed to direct, manual writing, editing, and debugging (albeit tactically and selectively, rather than as the norm). Crucially, as we have abundantly seen, navigating the vibe coding workflow still demands substantial expertise in manipulating the underlying code material to engage in these ways.

Interestingly, an example from YT21 sheds light on yet another complexity of how AI might change a programmer's relationship with the materiality of code. Specifically, because the programmer is using AI, they seem to be able to use more "vanilla" JavaScript and rely less on higher-level libraries like React, or abstraction-rich languages such as TypeScript. High-level libraries and languages, are, of course, abstractions created in direct response to the perceived limitations of working in a particular material substrate. Research on earlier generations of AI-assisted programming has noted how the human expertise embodied in such abstractions was a key contributor to the successful application of earlier models, which generated a relatively small number of lines of code at a time (Sarkar et al., 2022a), compared to the sweeping, cross-codebase changes made by LLM agents during vibe coding.

It is worth dwelling for a moment here on the implications of this example for manual programming versus AI-assisted programming. For a manual programmer, high-level libraries are valuable because they reduce the material barriers for engaging with code and are thus important for their workflow. For a manual programmer, writing in vanilla JavaScript can be verbose, tedious, and error-prone. And for an AI-assisted programmer working with a prior generation of AI code generation tools, the expressiveness of high-level libraries greatly amplified what could be achieved within the small contexts and size of output that those models were capable of.

However, when working with contemporary agentic AI tools, the code-generating model can handle this tedious material manipulation of the low-level abstractions while maintaining enough context to build more complex, ad-hoc abstractions as necessary. This might enable programmers to regain the benefits of working with lower-level code – principally, to avoid external dependencies and thus retain greater control and flexibility over their own codebase. Paradoxically, this suggests that the flight from material engagement in vibe coding may also enable a return to working effectively with lower material substrates.

The challenge (and at this point, this is hardly a novel observation) lies in balancing the benefits of offloading tedious manipulation against the potential loss of the distinct form of cognitive engagement and skill development that arises from directly confronting and resolving the intrinsic resistances posed by the material properties of code itself.

Vibing beyond coding. As touched upon in Section 1, the vibe coding workflow, where the user avoids direct manipulation of the "raw" material substrate (e.g., text in a document, formulas in a spreadsheet, paint on a canvas) may spread to other AI-assisted knowledge workflows. However, the potential for turning these workflows into "vibe" versions of themselves may differ between domains. The programming context has resources that the practitioner can draw upon to support their understanding of the vibe gestalt that do not easily transfer to other domains, such as the saliency of identifiers and component structure. Code can be tested through unit tests or throw formal exceptions to aid detection of errors. Code can also be compiled and deployed for visual and interactive inspection by the user to judge whether code generation has been satisfactory.

Text, on the other hand, is altogether more complex. Auditing generated text may inevitably require line-by-line reading to ensure that a "vibe written" document is accurate and reflects the user's intent. Therefore, expanding vibe coding workflows to other domains may require creating structural support for users to co-audit (Gordon et al., 2024) AI-intermediated workflows before we see vibing effectively expand to domains beyond coding.

4.4. Impression management and self-presentation: vibing for an audience

The practice of impression management involves the strategic curation of information within social encounters, enabling individuals to consciously or unconsciously direct how observers form judgments about people, objects, or situations. It is sometimes seen as synonymous with self-presentation, which is aiming to influence the perception of one's image (Goffman, 1956) (though it is straightforward to see that self-presentation is a subset of impression management). We find that creators engage in impression management: over and above the typical concerns of maintaining viewer engagement and satisfaction on streaming platforms, they act specifically to manage perceptions of their expertise and value in the context of contemporary social discourse around AI use in knowledge work.

To understand how our participants engage in impression management, it is necessary to contextualise our observations within the emerging understanding of the social dynamics of AI use. Reif et al. (2025) provides empirical evidence for a social evaluation penalty associated with using AI tools in the workplace. Their studies show that people anticipate and receive negative judgments from others, being perceived as lazier, less competent, and less diligent than those who use non-AI tools or no help at all. This penalty is linked to observers making negative dispositional inferences, attributing the use of AI to personal deficits rather than situational factors. There is thus a dilemma where the productivity benefits of AI can come at a social cost, leading some employees to conceal their AI use.

Adding to this, Schilke and Reimann (2025) describe the "transparency dilemma" of AI use. Their experimental evidence demonstrates that disclosing the usage of AI compromises trust in the user. This occurs because AI disclosure reduces perceptions of legitimacy. In many work contexts, there is a normative expectation that outputs should be the result of human expertise and judgment, and disclosing AI involvement is perceived as a deviation from these norms, undermining the legitimacy of the work process. Paradoxically, people who try to be trustworthy by transparently disclosing AI usage are trusted less. While disclosure erodes trust, being exposed for using AI by a third party has an even more detrimental effect on trust.

Sarkar (2025) offers a theoretical account for these negative social evaluations, framing AI shaming as a form of boundary work driven by class anxiety among knowledge workers. This perspective suggests that negative judgments and shaming of AI use arise from a perceived threat to the identity, value, and exclusivity of knowledge work professions. AI is seen as potentially eroding the "moat" of "extensive preparation" that historically protected these roles. Arguments used in AI shaming often question the quality, creativity, and legitimacy of AI-generated work, sometimes portraying AI users as "tasteless novices" or lacking the necessary skill, effort, or struggle associated with traditional knowledge work.

Vibe coding creators deploy impression management to directly confront this negative social landscape. Instead of concealing their AI use to avoid negative judgments, the video creators make their AI use central to their performance. They understand the potential for viewers to make negative dispositional inferences or engage in shaming by perceiving their AI use as evidence of laziness, lack of skill, or merely producing "slop". To counteract this, they strategically demonstrate their technical expertise, debugging skills, and ability to guide the model. By showing they are in control and possess the knowledge to validate and correct the AI's output, they differentiate themselves from the "unskilled opportunists" or "uncultivated dilettantes" (Sarkar, 2025) that the shaming discourse might target. They thus project an image of competence and expertise, even when relying on AI. This enables them to maintain their professional image and credibility while openly embracing AI. They are, in essence, performing a positive form of boundary work, attempting to define skilled AI-assisted programming as a legitimate and even advanced form of knowledge work.

4.5. Limitations

We note the novelty of vibe coding, which can be said to have begun in earnest in February 2025 with the publication of the Karpathy canon. This meant that our corpus of videos was limited to an initial ~35 hours of vibe coding content, ~8.5 hours of which represented high quality videos that we analysed. This is unquestionably quite a small corpus, and future work is necessary to expand our analysis. Moreover, as outlined from the outset in Section 1, vibe coding is a rapidly evolving, emerging, and negotiated practice, which means that our data bears witness to vibe coders merely *beginning* to undertake such exploration and negotiation. There is much still to learn about how the practice changes as vibe coders gain experience and develop new rhythms in the vibe coding workflow and as IDEs evolve to support them.

While we have discussed the performative aspect of these videos in detail (Sections 3.9.2 and 4.4), it must be acknowledged that performance aspects may skew workflows in ways that do not accurately reflect vibe coding in other realistic scenarios. To remedy this, in future work, it will be necessary to study vibe coding through other data sources, such as through laboratory experiments, or interviews and diary studies.

Finally, none of the selected videos depicted non-expert end-users (in particular, non-programmers). So, while our findings show how experienced coders can apply their expertise to address many of the challenges of vibe coding workflows, we do not know how non-programmers might meet these challenges and how they create barriers to effective vibe coding. Critically, we have studied the role of expertise in vibe coding in some detail (e.g., Section 3.7), and concluded that programming expertise is not only important, but essential for successful vibe coding. But we can only claim that this is true insofar as the vibe coder is in possession of said expertise, because we have not studied non-experts. Thus, research is needed to understand what kind of vibe coding – perhaps an altogether different activity, despite bearing superficial resemblances – might be practiced by non-experts.

5. Conclusion

This study is the first empirical analysis of vibe coding, an emerging programming paradigm where developers primarily author and edit code by interacting with code-generating AI through natural language prompts rather than direct code manipulation. Through framework analysis of curated think-aloud videos from YouTube and Twitch, we examined how programmers form goals, conduct workflows, and deploy expertise when engaging in this novel form of programming.

We find that vibe coding represents a meaningful evolution of traditional AI-assisted programming, characterised by episodes of iterative goal satisfaction where developers cycle through prompting, evaluation, debugging, and refinement. The workflow consistently involves strategic delegation of code generation to AI while maintaining human oversight through rapid evaluation and targeted intervention. Effective vibe coders apply varied and blended prompting strategies, ranging from high-level, diffuse, and subjective directives to detailed, fine-grained, and technical specifications.

Our observations challenge the notion that vibe coding eliminates the need for programming expertise. Instead, we observe a redistribution of expertise deployment. Traditional coding knowledge is redirected toward prompt and context management, rapid code evaluation, bug identification and resolution, and decisions about when to transition between AI assistance and manual intervention. Trust in AI tools is therefore granular, dynamic, and contingent, developed through iterative verification, and not blanket acceptance.

These findings may have broader implications for knowledge work. Vibe coding represents an early manifestation of material disengagement, where practitioners orchestrate content production through AI intermediaries rather than direct manipulation. However, our analysis suggests that such workflows still require substantial expertise in the underlying material substrate (in this case, code) to navigate effectively.

As vibe coding practices continue evolving, this research provides a foundational understanding for future investigations into human-AI workflows in programming and knowledge work more broadly.

References

Barik, T. (2017). Expressions on the nature and significance of programming and play. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 145–153. IEEE.

Barik, T., Johnson, B., and Murphy-Hill, E. (2015). I heart hacker news: expanding qualitative research findings by analyzing social news websites. In *Proceedings of the 2015 10th joint meeting on foundations of software engineering*, pages 882–885.

Barke, S., James, M. B., and Polikarpova, N. (2023). Grounded copilot: How programmers interact with code-generating models. *Proceedings of the ACM on Programming Languages*, 7(OOPSLA1):85–111.

Drosos, I. and Guo, P. J. (2021). Streamers Teaching Programming, Art, and Gaming: Cognitive Apprenticeship, Serendipitous Teachable Moments, and Tacit Expert Knowledge. In *2021 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 1–6.

Drosos, I., Sarkar, A., Xu, X., Negreanu, C., Rintel, S., and Tankelevitch, L. (2024). "It's like a rubber duck that talks back": Understanding Generative AI-Assisted Data Analysis Workflows through a Participatory Prompting Study. In *Proceedings of the 3rd Annual Meeting of the Symposium on Human-Computer Interaction for Work*, CHIWORK '24, New York, NY, USA. Association for Computing Machinery.

Ensmenger, N. L. (2012). *The computer boys take over*. History of Computing. MIT Press, London, England.

Goffman, E. (1956). *The Presentation of Self in Everyday Life*. Doubleday, Scotland.

Goldsmith, L. J. (2021). Using framework analysis in applied qualitative research. *Qualitative report*, 26(6):2061–2076.

Gordon, A. D., Negreanu, C., Cambronero, J., Chakravarthy, R., Drosos, I., Fang, H., Mitra, B., Richardson, H., Sarkar, A., Simmons, S., Williams, J., and Zorn, B. (2024). Co-audit: tools to help humans double-check AI-generated content. *Proceedings of the 14th annual workshop on the intersection of HCI and PL (PLATEAU 2024).*

Green, T. R. (1989). Cognitive dimensions of notations. *People and computers V*, pages 443–460.

Kaldis, B. (2013). Gestalt psychology. In Kaldis, B., editor, *Encyclopedia of Philosophy and the Social Sciences*, volume 2, pages 383–386. SAGE Publications, Inc.

Karpathy, A. (2025). There's a new kind of coding I call "vibe coding", where you fully give in to the vibes, embrace exponentials, and forget that the code even exists. I "Accept All" always, I don't read the diffs anymore. When I get error messages I just copy-paste them in with no comment, usually that fixes it. The code grows beyond my usual comprehension … Sometimes the LLMs can't fix a bug so I just work around it or ask for random changes until it goes away. It's not too bad for throwaway weekend projects, but still quite amusing. [Tweet from account @karpathy]. `https://x.com/karpathy/status/1886192184808149383`. Online; accessed 31-May-2025.

Kery, M. B. and Myers, B. A. (2017). Exploring exploratory programming. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 25–29. IEEE.

Knuth, D. E. (1992). *Literate Programming*. Center for the Study of Language and Information Publication Lecture Notes. Centre for the Study of Language & Information, Stanford, CA.

Laugier, S. (2013). *Why we need ordinary language philosophy*. University of Chicago Press.

Lee, H.-P. H., Sarkar, A., Tankelevitch, L., Drosos, I., Rintel, S., Banks, R., and Wilson, N. (2025). The Impact of Generative AI on Critical Thinking: Self-Reported Reductions in Cognitive Effort and Confidence Effects From a Survey of Knowledge Workers. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems*, CHI '25, New York, NY, USA. Association for Computing Machinery.

Lee, M. J. L., Sarkar, A., and Blackwell, A. F. (2024). Predictability of identifier naming with Copilot: A case study for mixed-initiative programming tools. In *Proceedings of the 35th Annual Conference of the Psychology of Programming Interest Group (PPIG 2024).*

Malafouris, L. (2019). Mind and material engagement. *Phenomenology and the cognitive sciences*, 18(1):1–17.

Reif, J. A., Larrick, R. P., and Soll, J. B. (2025). Evidence of a social evaluation penalty for using ai. *Proceedings of the National Academy of Sciences*, 122(19):e2426766122.

Ritchie, J. and Spencer, L. (2002). Qualitative data analysis for applied policy research. In *Analyzing qualitative data*, pages 173–194. Routledge.

Sarkar, A. (2023a). Exploring Perspectives on the Impact of Artificial Intelligence on the Creativity of Knowledge Work: Beyond Mechanised Plagiarism and Stochastic Parrots. In *Proceedings of the 2nd Annual Meeting of the Symposium on Human-Computer Interaction for Work*, CHIWORK '23, New York, NY, USA. Association for Computing Machinery.

Sarkar, A. (2023b). Should Computers Be Easy To Use? Questioning the Doctrine of Simplicity in User Interface Design. In *Extended Abstracts of the 2023 CHI Conference on Human Factors in Computing Systems*, CHI EA '23, New York, NY, USA. Association for Computing Machinery.

Sarkar, A. (2023c). Will Code Remain a Relevant User Interface for End-User Programming with Generative AI Models? In *Proceedings of the 2023 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2023, page 153–167, New York, NY, USA. Association for Computing Machinery.

Sarkar, A. (2024). Intention Is All You Need. In *Proceedings of the 35th Annual Conference of the Psychology of Programming Interest Group (PPIG 2024)*.

Sarkar, A. (2025). AI Could Have Written This: Birth of a Classist Slur in Knowledge Work. In *Proceedings of the Extended Abstracts of the CHI Conference on Human Factors in Computing Systems*, CHI EA '25, New York, NY, USA. Association for Computing Machinery.

Sarkar, A., Gordon, A. D., Negreanu, C., Poelitz, C., Srinivasa Ragavan, S., and Zorn, B. (2022a). What is it like to program with artificial intelligence? In *Proceedings of the 33rd Annual Conference of the Psychology of Programming Interest Group (PPIG 2022)*.

Sarkar, A., Ragavan, S. S., Williams, J., and Gordon, A. D. (2022b). End-user encounters with lambda abstraction in spreadsheets: Apollo's bow or Achilles' heel? In *2022 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 1–11.

Schilke, O. and Reimann, M. (2025). The transparency dilemma: How ai disclosure erodes trust. *Organizational Behavior and Human Decision Processes*, 188:104405.

Srinivasa Ragavan, S., Sarkar, A., and Gordon, A. D. (2021). Spreadsheet Comprehension: Guesswork, Giving Up and Going Back to the Author. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, CHI '21, New York, NY, USA. Association for Computing Machinery.

Tankelevitch, L., Kewenig, V., Simkute, A., Scott, A. E., Sarkar, A., Sellen, A., and Rintel, S. (2024). The Metacognitive Demands and Opportunities of Generative AI. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*, CHI '24, New York, NY, USA. Association for Computing Machinery.

Vaithilingam, P., Zhang, T., and Glassman, E. L. (2022). Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. In *Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems*, CHI EA '22, New York, NY, USA. Association for Computing Machinery.

Weinberg, G. M. (1971). *The psychology of computer programming*. New York : Van Nostrand Reinhold.

Wu, Y., Kropczynski, J., Shih, P. C., and Carroll, J. M. (2014). Exploring the ecosystem of software developers on github and other platforms. In *Proceedings of the companion publication of the 17th ACM conference on Computer supported cooperative work & social computing*, pages 265–268.

Appendices

A. Video sources

This table lists the videos included in our analysis. Our use of videos in the various research phases are indicated as follows under the "Research phases" column: "sensitising" means that 10-20 minute excerpts of the video were viewed during the initial sensitising phase; "development" means that the entire video was viewed in detail by both researchers with a view to developing the categories for the framework and the criteria for selecting the final videos to analyse; "analysis" means the final framework analysis was applied to this entire video.

Table 2 – Sources Table

ID	URL and video title	Date published	Duration	Research phases
YT1	`https://www.youtube.com/watch?v=5k2-NOh2tk0` What is "Vibe Coding"? Here's how I do it...	23-Feb-25	00:17:19	Sensitising
YT2	`https://www.youtube.com/watch?v=z7aZBxjfDIw` Build iOS apps with 0 Knowledge | Vibe Coding | TinderTodo App | Swift	16-Mar-25	00:26:22	Sensitising
TW1	`https://www.twitch.tv/acorn1010/video/2398186371` !foony | VIP | Vibe Coding | !idea	06-Mar-25	05:30:50	Sensitising, analysis
YT3	`https://www.youtube.com/watch?v=OVnDfcRtFew` Vibe Coding: Email Campaigns & Agents | Chatbot Builder AI	14-Mar-25	01:53:49	Sensitising
YT4	`https://www.youtube.com/watch?v=YWwS911iLhg` Vibe Coding Tutorial and Best Practices (Cursor / Windsurf)	05-Mar-25	00:21:48	Sensitising
YT5	`https://www.youtube.com/watch?v=k9WrU6uR2YM` Vibe Coding my Entire Landing Page with AI (Lovable):	11-Mar-25	00:11:11	Sensitising, development
YT6	`https://www.youtube.com/watch?v=xh5PhzZZcnQ` Vibe Coding with Wispr Flow and Cursor	12-Mar-25	00:09:49	Sensitising
YT7	`https://www.youtube.com/watch?v=dan3QfN3CDU` Karpathy Vibe Coding Full Tutorial with Cursor (Zero Coding)	06-Feb-25	00:33:47	Sensitising
YT8	`https://www.youtube.com/watch?v=g84CGmelvSU` Vibe Coding: Launch Your SaaS with AI (Cursor, Supabase, & Stripe)	28-Feb-25	00:02:35	Sensitising
YT9	`https://www.youtube.com/watch?v=GwhFjhMF6OA` Vibe coding using ChatGPT	06-Feb-25	00:10:56	Sensitising

Continued on next page

Table 2 – Sources Table (continued)

ID	URL and video title	Date published	Duration	Research phases
YT10	`https://www.youtube.com/watch?v=ulJrdYXLo9I` Vibe Coding a FULL Game ?? (AI Coding)	04-Mar-25	00:11:52	Sensitising
YT11	`https://www.youtube.com/watch?v=faezjTHA5SU` Complete Guide to Cursor For Non-Coders (Vibe Coding 101)	11-Feb-25	02:28:16	Sensitising
YT12	`https://www.youtube.com/watch?v=JEU07S2WyDs` Vibe Coding in 2025: No Typing, No Stress, Just AI!	08-Mar-25	00:21:03	Sensitising, development
YT13	`https://www.youtube.com/watch?v=i0gWDz9EUgI` Vibe Coding 101	11-Feb-25	00:13:48	Sensitising
YT14	`https://www.youtube.com/watch?v=yGZtu_6E1l8` Vibe Coding in Databutton AI - FREE AI Code Editor for Full stack development in Claude Sonnet 3.7	03-Mar-25	00:10:22	Sensitising
YT15	`https://www.youtube.com/watch?v=irBnUAq3MAw` VIBE CODING with CURSOR - Draw w/ HTML and JS	17-Feb-25	00:17:45	Sensitising, development, analysis
TW2	`https://www.twitch.tv/coolaj86/video/2399572024` Vibe Coding Hangout: Live w/ Grok3 Unfurl-as-a-Service:Link Unshortener	07-Mar-25	05:57:50	Sensitising
TW3	`https://www.twitch.tv/vadimnotjustdev/video/2402985837` Vibe Coding a React Native Game	11-Mar-25	02:41:35	Sensitising
YT16	`https://www.youtube.com/watch?v=HBwXs99LFlw` VIBE CODING 3 min demo | Cursor + o3-mini + SuperWhisper	15-Feb-25	00:03:20	Sensitising
YT17	`https://www.youtube.com/watch?v=Nt2Lkdy3f5Y` ?Vibe? Coding AI Agent to do sales for me (OpenAI Agents SDK)	17-Mar-25	00:40:04	Sensitising
YT18	`https://www.youtube.com/watch?v=_QOvocOFLbo` Vibe Coding - Code & Chill	16-Mar-25	00:22:28	Sensitising, development
YT19	`https://www.youtube.com/watch?v=TwSYePsdfOk` WIP at Recall: Vibe Coding Crypto Alpha Detection Agents #1	15-Mar-25	00:27:28	Sensitising
YT20	`https://www.youtube.com/watch?v=Pe8ghwTMFlg` ?? VS Code - Agent Mode UPGRADE!	07-Mar-25	01:32:45	Sensitising
YT21	`https://www.youtube.com/watch?v=EAPWrpvIOjs` Vibe Coding with AI: Watch me fix a chat interface in real-time in this long video	06-Feb-25	01:29:37	Sensitising, analysis
YT22	`https://www.youtube.com/watch?v=SNARzs6jzWY` Vibe Coding Startup Using Most Advanced AI — Phase 1	14-Mar-25	00:32:05	Sensitising, analysis

Continued on next page

Table 2 – Sources Table (continued)

ID	URL and video title	Date published	Duration	Research phases
YT22b	`https://www.youtube.com/watch?v=u4yqSsYGpcc` Vibe Coding Startup: Dashboard — Phase 2 Part 1	21-Mar-25	00:37:02	Sensitising, analysis
YT23	`https://www.youtube.com/watch?v=DvDOb0ZezQQ` Vibe coding actually sucks	19-Mar-25	00:28:31	Sensitising
YT24	`https://www.youtube.com/watch?v=NYVaCr3T1T0` Vibe Coding is Actually INSANE... (Vibe Coding Tutorial for Beginners)	21-Mar-25	00:38:24	Sensitising
YT25	`https://www.youtube.com/watch?v=opi1s_5Dm-c` He makes $750 a day 'Vibe Coding' Apps (using Replit, ChatGPT, Upwork)	21-Mar-25	00:45:16	Sensitising
YT26	`https://www.youtube.com/watch?v=faPSZV5XwyI` Start Vibe Coding Like a Pro, Here's How	17-Mar-25	00:21:48	Sensitising
YT27	`https://www.youtube.com/watch?v=y3vQBBmE3Cc` AI Created My Game in 6 Hours... Vibe Coding so you don't have to	20-Mar-25	00:08:31	Sensitising
YT28	`https://www.youtube.com/watch?v=icRXv9RXhXI` Vibe Coding FULL Course + WIN MacBook Pro, PlayStation 5 ??	27-Mar-25	01:22:39	Sensitising
YT29	`https://www.youtube.com/watch?v=jTaqixu79qU` Vibe Code SaaS & Mobile Games (Grok, Bolt, Cursor, Prompts)	18-Mar-25	00:16:28	Sensitising
TW4	`https://www.twitch.tv/codesinthedark/video/2414451732` DAY ??x3 - Vibe coding a game using Cursor #vibejam	24-Mar-25	01:32:40	Sensitising
YT30	`https://www.youtube.com/watch?v=OZaxtm3RyCw` Vibe Coding Made Easy: The Essential Cursor AI Tool!	30-Mar-25	00:12:16	Sensitising
YT31	`https://www.youtube.com/watch?v=y9XEBnNvu2Q` Vibe Coding and Portfolio Reviews	28-Mar-25	01:12:38	Sensitising
YT32	`https://www.youtube.com/watch?v=_yKDiRlToSs` Vibe Coding For Non Coders - I built an online game in 30 seconds using AI	25-Mar-25	00:05:19	Sensitising
YT33	`https://www.youtube.com/watch?v=5qwucCaHpWY` BUILDING A GAME IN 7 DAYS	21-Mar-25	00:12:16	Sensitising
YT34	`https://www.youtube.com/watch?v=78jina4V7j4` One Shotting a Rank and Rent Site With Lovable and Going Live (Vibe Coding With Alex)	02-Apr-25	00:26:16	Sensitising
YT35	`https://www.youtube.com/watch?v=X-xiJgkqnok` Episode #503 - Vibe Coding	01-Apr-25	00:17:33	Sensitising

B. Analysis framework

Table 3 – Qualitative analysis framework for vibe coding think aloud videos

Top Level Category	Subcategories	Elaboration
Goals	a. What do they build?	i. What applications did vibe coders build? ii. What types of applications are created (e.g., landing pages, games, etc.)? iii. Is vibe coding fully appropriate to complete a project, or does it require transitions to manual coding for more complex projects like games?
	b. Expectation – for full/partial success, or just exploration?	i. Do people go into vibe coding expecting to fully succeed, or are they taking an exploratory attitude? ii. Are users generally expecting complete success with vibe coding, or do they experiment to see how far the technology can be pushed?
Intentions	a. Initial intention	i. How is the intention for an app formed and refined in vibe coding? ii. Is the intent pre-formed before coding begins, or does it evolve through iterative interactions with the AI? iii. Which parts of the intent are established beforehand, and how does intent spread across various prompts and phases?
	b. Is intention refined? How and why?	i. What AI workflow resources (e.g., exploration, seeking alternatives) support and refine this intent? ii. Does the AI itself contribute to shaping the final intent by filling in gaps? How does the back-and-forth interaction with the AI shape the final output? iii. How does the iterative and conversational nature of vibe coding affect users' creativity and problem-solving strategies? iv. Does this process encourage more exploration or lead to a reliance on the AI's suggestions?
Workflow	a. Stages of workflow	i. What are the stages of the vibe coding workflow? ii. How is time and effort allocated across different stages (e.g., programming, fault localization, debugging versus planning and testing)? iii. How do these stages compare with those in more traditional AI-enhanced programming workflows and prior identified AI workflows such as iterative goal satisfaction?
	b. Portfolio of tools and technologies	i. How do different tools work together in the context of vibe coding? ii. How are other resources (such as documentation, web searches, etc.) integrated into the vibe coding workflow?
Prompting	a. Any setup work?	i. How do creators set up a vibe coding session (e.g., editing prompt instructions vs. using default settings)?
	b. Explicit prompting strategies	i. What are the prompting strategies in vibe coding? ii. What methods are employed to initiate and guide the AI during the coding process?
	c. Granularity of prompts – high/low level	i. What is the granularity of prompts and iterations? ii. How fine-grained are the iterative steps in the vibe coding process? iii. How is aesthetic intent communicated through natural language prompts translated (or misinterpreted) by generative AI into technical specifications and code?
	d. Single/multi-objective prompts	i. Do users address multiple issues at once or handle one problem at a time?
	e. What prompting modes are used – voice/text/images?	i. What is the role of different modes (voice, text, screenshot, etc.) in vibe coding? ii. How do creators incorporate various input modes into the vibe coding process? iii. How is code reuse managed, for instance by borrowing online examples to steer the AI?
Debugging	a. What debugging strategies are applied?	i. What debugging strategies are employed in vibe coding?

Top Level Category	**Subcategories**	**Elaboration**
		ii. How do creators approach debugging code generated through vibe coding?
Challenges	a. What challenges (besides code bugs) are encountered in achieving goals?	i. What challenges and barriers are encountered during vibe coding? ii. What technical, conceptual, or workflow-related obstacles arise throughout the process?
	b. How are they addressed?	i. What strategies are adopted to address these challenges?
Expertise	a. When is expertise used?	i. How do experts deploy their expertise in vibe coding? ii. In what ways do experts deploy their expertise?
	b. What kinds of expertise are used?	i. Is the relevant expertise strictly developer knowledge, or does it include understanding system requirements, end-to-end design, and the broader technical ecosystem? ii. Could a tech-forward attitude be as significant as traditional coding skills in this context?
	c. When do they switch to manual work?	i. When do practitioners transition from vibe coding to manual editing (code, text, image, artifact)? ii. At what point in the workflow is it appropriate to jump into traditional manual editing?
Trust	a. Any reflections on trust, overtrust, reliance?	i. What is the nature of trust in vibe coding? ii. How do users develop trust in AI-generated outputs, and what are the risks of overtrust or overreliance on the technology?
Definition of vibe coding and performance	a. How do programmers define vibe coding?	i. When is an activity considered vibe coding versus simply using AI tools? ii. Does vibe coding include not only code generation but also debugging, planning, and testing? iii. Where does vibe coding fit on the spectrum of AI-assisted programming activities?
	b. How does performance for YouTube/Twitch skew production?	i. How does the performative aspect of vibe coding (e.g., YouTube demonstrations) influence the practice and perception of generative AI in programming? ii. How might public demonstrations affect tool choice, prompting strategies, and the framing of users' experiences?

Design opportunities for the psychology of programming after AI

Clayton Lewis
Emeritus Professor
University of Colorado Boulder
clayton.lewis@colorado.edu

Abstract

The advent of artificial intelligence coding tools calls for a shift in focus in the PPIG community from the psychology of writing programs toward the psychology of evaluating them. Using some simple (but real) examples in the domain of educational programming, this paper explores what approaches non-programmers might use to determine whether a program that has been given to them works as desired, including the roles of AI tools in this process, as well as in writing the code. The exploration suggests that cognitive dimensions are useful in understanding these matters, and that there are a number of research directions that may support the creation of programs that are easier for people to evaluate.

1. Introduction

This paper asks, what are the opportunities for PPIG when people don't have to write programs to have programs? Over the last few years, AI tools for programming have advanced rapidly. At PPIG 2023 I discussed a program like the one shown in Figure 1 (Lewis, 2023). GPT 4 could write such a program, but an iterative prompting process was needed. Now, Anthropic's Claude writes a working program in response to a single prompt:

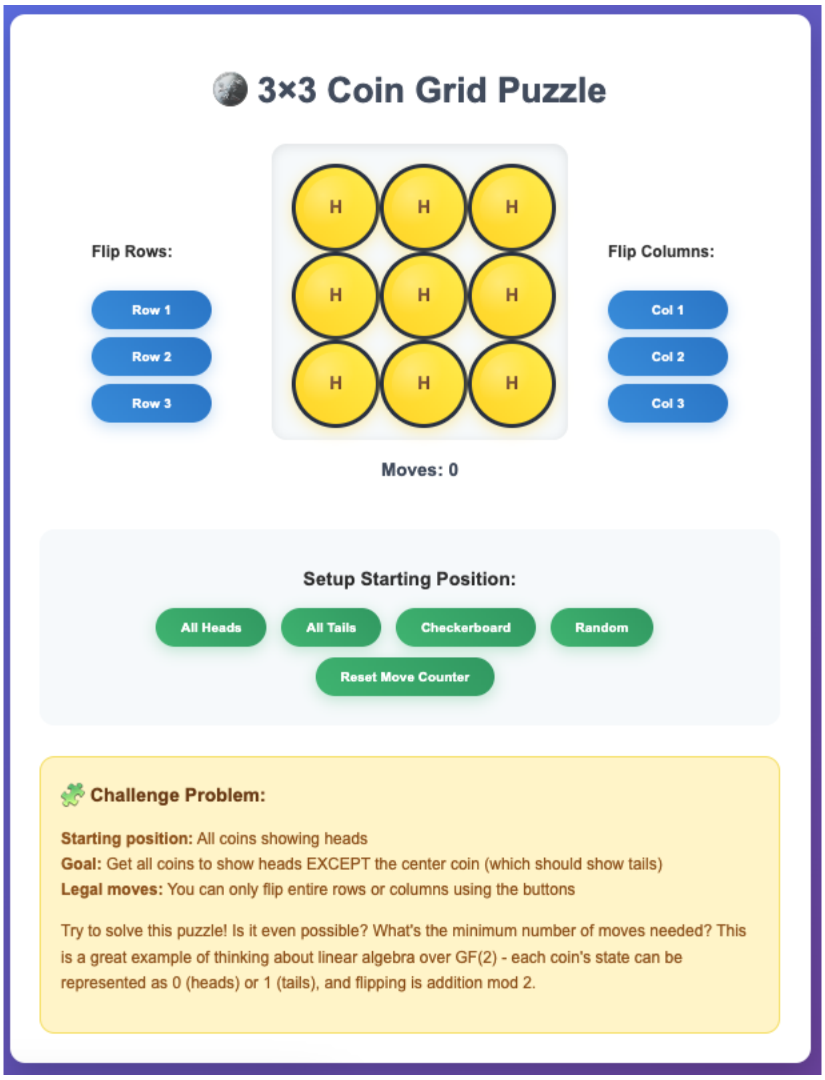
> For a math class I'm teaching, I'd like a Web app that displays a 3x3 grid of coins, with buttons that flip all the coins in each row or column. I'd also need some way to set up any starting arrangement of coins, such as all heads. My idea is that my students can work on the problem of whether you can change all heads to all heads except for the center coin, using the legal operations.

In situations within the scope of such tools, the focus of attention shifts from "How can I write this program?" toward "How can I tell if this program I've been given does what I want?" What opportunities for the PPIG community does this shift entail? We can approach this question by considering how users can approach the evaluation of programs they are given.

For some programs, looking at the output is enough. If I just want my program to create an animated cat, I can just look at the cat I get, and decide if it is good enough. But more typically we want a program to behave in some correct way in the future, not just in some one current situation. How can we tell if a program we've been given will do that? Let's use the Nine Coins problem to explore this situation, or at least, to scratch the surface.

The Nine Coins program originated as an example of the use of the Boxer language, developed by Andy diSessa as a tool to support the development of computation as a tool for thought and expression in schools (diSessa & Abelson, 1986; Mason, 1995). One aim of Boxer is to allow students and teachers to develop tools for exploring ideas, perhaps abstract algebra in this case (the operations on the grid form a group). The idea is that, with the right sort of language, programs like this can be developed easily and spontaneously, in the course of classroom discussion. Students can then use the program to explore the operations on the grid, as in the example cited in the prompt, above.

It would be bad if the program didn't act correctly. Students might draw the wrong conclusion about the behavior of the operations on the coins, for example concluding that the configuration with one tail in the center can't be reached from all heads, when it actually can, or that it can't be reached when it can. This would lead to confusion, and wasted time, at best.



Challenge Problem:

Starting position: All coins showing heads
Goal: Get all coins to show heads EXCEPT the center coin (which should show tails)
Legal moves: You can only flip entire rows or columns using the buttons

Try to solve this puzzle! Is it even possible? What's the minimum number of moves needed? This is a great example of thinking about linear algebra over GF(2) - each coin's state can be represented as 0 (heads) or 1 (tails), and flipping is addition mod 2.

Figure 1 – Claude's Nine Coins program.

In the Boxer scenario, the class would have written the program, using the Boxer language. The process of writing the program would bring with it a sense of its correctness. But what if, as is now possible, they have simply asked for the program, and are given it, with no participation in writing it at all? What approaches could they take to gain confidence that the program will work?

Before digging into this question, let's note that the Boxer project has larger goals, and that programming in Boxer potentially has benefits that go beyond just getting a program that works. DiSessa argues that computing is a fundamental intellectual tool, and that mastering it should be considered a form of literacy on a plane with textual literacy or mathematical literacy (DiSessa, 2001). We won't consider this matter further here, beyond noting that developing these ideas in the context of AI tools would be an important PPIG contribution.

2. One way to evaluate the Nine Coins programs is to test it

An obvious approach is to press the buttons, and check what they do. It's very easy to do some of this, but not completely trivial to do it completely. There are 11 buttons, and one would have to be patient in trying each of them. But just testing each button doesn't give much assurance that the program will work reliably. It could be that certain sequences of button presses don't work correctly, or that a button would fail for just certain grid patterns.

In serious software testing, constructing an adequate collection of tests isn't easy. To give a fair level of confidence a test suite should at least exercise each part of the code. But if one has no access to the code, or can't understand the code, as would be the case for our non-programming class, such test coverage isn't possible, without assistance.

2.1. AI tools can write and apply tests

One can ask Claude to test a Nine Coins program that it has provided; I've done this for a simpler form of the program than that shown in Figure 1, discussed later. The conversation can be seen at `https://claude.ai/public/artifacts/edfb6e97-78cf-4562-aed6-224935648221` (I'm including links to conversations with Claude for interested readers. But I'll try to describe the important points in the text, in part because these links may not work in perpetuity.)

Unfortunately what Claude says would not be easy to follow for a non-programmer. For example, it refers to "Boundary and Edge Case Tests"; what are those? Under that heading is "Boundary values - Tests first/last rows and columns (indices 0 and 2)". What are "indices"?

Claude did not spontaneously perform coverage analysis, but did when asked, as can be seen in the conversation. Unfortunately, the coverage display can't readily be linked to the actual code, using line labels (like "flipRow-3a") that aren't shown in the code listing users can see.

This wouldn't necessarily be a big problem. A non-programmer consumer could ask their tool to carry out this kind of evaluation, and just trust the results. In low stakes situations like Nine Coins that could well be adequate. In situations where more skepticism is warranted, could more support be offered, including to people who don't understand code? This is a design opportunity for the PPIG community: how can program tests and test results be informatively presented, for a range of audiences?

3. Another way to assess Nine Coins is to understand the code in a way that builds confidence

As mentioned earlier, part of the vision for the Boxer language is that people who write code can have some confidence based on their understanding of the code. Can people who didn't write code, and don't understand code, be helped to understand it?

Here is a conversation with Claude about its initial offering for Nine Coins: `https://claude.ai/share/520a561a-3cc0-4fad-9b08-ba066f03ae96`. Claude tries various ways to explain why the code can be trusted, none very easy to understand. For example, a piece is

> Grid Indexing: The 3×3 grid correctly maps to a linear array using index = row * 3 + col
>
> Position [0][0] $\rightarrow$ index 0, [0][1] $\rightarrow$ index 1, [0][2] $\rightarrow$ index 2

This excerpt refers to an aspect of the code that presents difficulty, the representation of the two-dimensional grid of coins with a linear array. Here is the code, discussed later in the conversation:

```
function flipRow(rowIndex) {
    for (let col = 0; col < 3; col++) {
        const index = rowIndex * 3 + col;
        grid[index] = !grid[index];
    }
}
```

This code works, but isn't easy for a non-programmer to follow, even leaving aside notational matters like how repeated operations are described. What does the index calculation in the third line have to do with the problem?

Another issue that can be seen here (by someone who understands programming) is the representation of a coin's state as a Boolean value. There are two aspects to this. First, the values in the grid are true and false, not heads and tails. That makes it hard to relate this code to the problem. Second, a non-programmer would have no idea what the exclamation point means in this case. Even if they are told that it is negation, the relationship between this operation and flipping a coin has to be further explained.

In the course of the conversation (with quite a bit of prodding) Claude developed a simpler program that addresses these issues, shown in Figure 2 (see code in Listings 1 and 2).

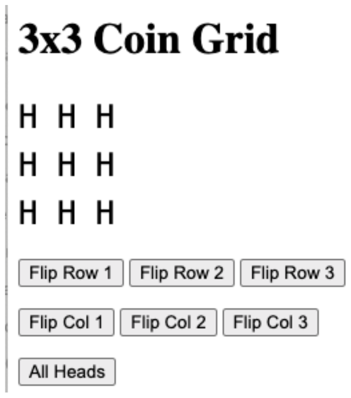


Figure 2 – Nine Coins simplified by Claude.

This code is indeed easier to understand. The grid is 3 by 3. The representation of coin states is more transparent ("H" and "T"), and the repetition is spelled out, rather than expressed with loop logic. We can see in this conversation that Claude is capable of writing code that is easier to understand than its first offering. But it took iterative guidance from the user, identifying areas of difficulty, to get it to do it.

It's quite likely that prompts could be devised that would push Claude to produce simpler code, without requiring the user to make iterative requests for clarification. This isn't simple, though. This quick stab,

> For a math class I'm teaching, I'd like a Web app that displays a 3x3 grid of coins, with buttons that flip all the coins in each row or column. I'd also need some way to set up any starting arrangement of coins, such as all heads. My idea is that my students can work on the problem of whether you can change all heads to all heads except for the center coin, using the legal operations. I'll want to be able to understand the code well enough that I'll be confident that all the buttons will do exactly what they are supposed to do. And I'm not a programmer. Can you make the code so simple that you can easily explain it to me?

was not effective in producing code as simple as what was obtained in the conversation above by iterative requests. For example, Claude used Booleans in this "simple" code to represent the coins. So here's another design opportunity for PPIG: Can we develop generic prompting language that can get coding tools to produce really simple, easy to understand code? Packing examples of what's wanted into the prompt could be an approach.

4. What could make code really much easier to understand?

What we've seen Claude do seems to be just nibbling around the edges of understandability. A problem that's been recognized by the PPIG community, that Claude hasn't touched, is what Steve Draper calls the pumping problem (Draper, 1986). In the code in Table 1, consider the function show(). There's the question of what's being said there: What is "+" in this context? What is "InnerHTML"? There's also a more basic question: why is this code needed at all? Isn't updating the grid of coins all the program needs to do?

As programmers know, no, it isn't. In most programming systems, the entities being manipulated can't be seen. Rather, code is needed to pump information from inside the computer to the outside (other pumping code is needed to take information in.) As Draper observed, spreadsheets avoid pumping code: data in a spreadsheet are visible, without the user having to do anything to make them so. The success

Listing 1 – Code for simplified Nine Coins program.

```
<!DOCTYPE html>
<html>
<head>
    <title>Simple Coin Grid</title>
</head>
<body>
    <h1>3x3 Coin Grid</h1>
    <div id="display" style="font-family: monospace; font-size: 24px;
         line-height: 1.5;">
        <!-- Grid shows here -->
    </div>
    <p>
        <button onclick="flipRow(0)">Flip Row 1</button>
        <button onclick="flipRow(1)">Flip Row 2</button>
        <button onclick="flipRow(2)">Flip Row 3</button>
    </p>
    <p>
        <button onclick="flipCol(0)">Flip Col 1</button>
        <button onclick="flipCol(1)">Flip Col 2</button>
        <button onclick="flipCol(2)">Flip Col 3</button>
    </p>
    <p>
        <button onclick="allHeads()">All Heads</button>
    </p>
    <script>
        // The grid
        let grid = [
            ["H", "H", "H"],
            ["H", "H", "H"],
            ["H", "H", "H"]
        ];

        // Show the grid
        function show() {
            let text = "";
            text += grid[0][0] + " " + grid[0][1] + " " + grid[0][2]
                 + "<br>";
            text += grid[1][0] + " " + grid[1][1] + " " + grid[1][2]
                 + "<br>";
            text += grid[2][0] + " " + grid[2][1] + " " + grid[2][2];
            document.getElementById('display').innerHTML = text;
        }
```

Listing 2 – Code for simplified Nine Coins program, continued.

```
        // Flip a row
        function flipRow(row) {
            if (grid[row][0] === "H") { grid[row][0] = "T"; }
            else { grid[row][0] = "H"; }
            if (grid[row][1] === "H") { grid[row][1] = "T"; }
            else { grid[row][1] = "H"; }
            if (grid[row][2] === "H") { grid[row][2] = "T"; }
            else { grid[row][2] = "H"; }
            show();
        }

        // Flip a column
        function flipCol(col) {
            if (grid[0][col] === "H") { grid[0][col] = "T"; }
            else { grid[0][col] = "H"; }
            if (grid[1][col] === "H") { grid[1][col] = "T"; }
            else { grid[1][col] = "H"; }
            if (grid[2][col] === "H") { grid[2][col] = "T"; }
            else { grid[2][col] = "H"; }
            show();
        }

        // Reset to all heads
        function allHeads() {
            grid[0][0] = "H"; grid[0][1] = "H"; grid[0][2] = "H";
            grid[1][0] = "H"; grid[1][1] = "H"; grid[1][2] = "H";
            grid[2][0] = "H"; grid[2][1] = "H"; grid[2][2] = "H";
            show();
        }

        // Start by showing the grid
        show();
    </script>
</body>
</html>
```

of spreadsheets suggests that changing the stuff programs work with can help with understanding. Code that works with stuff that's visible by default could be easier to understand.

4.1. Understanding more about understanding could help

To go farther, it would be useful to have more insight into how understanding works. Existing conceptions of understanding have focused on knowledge structures, and relationships among them; see for example (Gentner, 1983). Another idea is that understanding includes identifying causal mechanisms at work in a situation (Lewis, 1988; Russ, Coffey, Hammer, & Hutchison, 2009).

The unexpected success of Large Language Models, that seem to function largely without explicit provision for structures or causal mechanisms, suggests that mental processes result from the action of a large collection of small predictive regularities, linked by analogical relationships. This idea is explored in (Lewis, 2025).

A plausible reconception of understanding is that it is itself a predictive process. One has understood a novel situation when one can predict what will happen in it, as different events occur. Regularities observed in one situation can be used to make predictions in another, when analogies link the situations.

4.2. Phenomena in physics learning illustrate this process of understanding

Andy diSessa, in addition to his work on the Boxer system, has made substantial contributions to our understanding of how understanding develops in physics. One of these contributions is the idea of phenomenological primitive, or p-prim (DiSessa, 1993). For example, Ohm's p-prim describes a collection of analogies that link the predictive regularity "more voltage and same resistance gives more current" to the regularity "more push and same friction gives more motion", in a domain of pushing things on a surface (and many more, in the same domains, and other domains.) This constellation of some push, and some inhibition, and some result shows up in many situations, and can readily be understood.

Jim Minstrell, in earlier work (Minstrell, 1982), showed how students could understand how it is that a table, seemingly a passive object, can exert force, when an object is lying on it. By shining a laser beam off a mirror on the table top, Minstrell was able to show that the table deforms very slightly when an object is placed on it. That allowed the students to see that the table acts like a spring, something they generally accept as capable of exerting force. Minstrell's approach builds understanding of force exerted by a table from understanding of springs.

4.3. This view of understanding implicates the cognitive dimension, closeness of mapping

We can apply these ideas to understanding code, using the Nine Coins program. Cognitive dimensions analysis (Green, 1989; Blackwell et al., 2001), among many other references, highlights the importance of closeness of mapping, how easy it is to connect a representation to a user's idea of what it represents. Focusing on the code that flips the coins, we can see that both versions of the program pose challenges. In the original program, the Boolean negation operation was used, something non-programmers will almost certainly have little or nothing to connect to:

```
grid[index] = !grid[index];
```

There's a gap opened up in that program between the familiar domain of coins, with their natural behavior, and the unfamiliar domain of Boolean values and operations on them. As discussed earlier, there is also a mapping gap between the two-dimensional grid of coins, and the one-dimensional list of Boolean values.

The simplified program, in Figure 2, uses a closer mapping, between "H" or "T" and the heads or tails of coins. But there is still trouble, as looking at the code shows. Here's the code for flipping one of the coins in the grid:

```
if (grid[1][col] === "H") { grid[1][col] = "T"; } else {
    grid[1][col] = "H"; }
```

Even leaving aside the cryptic notation, such as the distinction between "===" and "=", this seems an odd representation of the idea of flipping. Flipping doesn't seem to require any conditional logic, but this representation of it does.

These two code snippets illustrate, in different ways, that understanding code can take people beyond things they already know about. With programming as it is, people have to spend weeks or longer to be able to reason about code. Could that be avoided? There's another PPIG design opportunity here: Could programs be constructed from materials whose behaviors are closer to behaviors non-programmers are already familiar with?

5. Capitalizing on knowledge of the behavior of physical objects in a programming system

A domain people have a lot of experience with is physical objects. Making computational objects more like physical objects could have advantages, in making their behavior easier to understand.

One advantage is that physical objects don't require pumping code, while common computational objects do. Pumping code poses multiple challenges for understanding. One has to understand the code itself, for example the concatenation of text in the show() function in Listing 1. But also one has to understand why pumping code is needed at all. That is, one has to understand all of what the code is doing, how it is doing it, and why it is doing it. If computational objects were more like physical objects, none of this would be needed. Values in spreadsheets show that doing without pumping is possible, at least to some extent.

It is possible, though not easy, to write Nine Coins as a spreadsheet. Current spreadsheets push one into a scripting environment that has its own challenges, inherited from the scripting language. For example, it's not simple to access the value of a cell in the script.

Claude can navigate these challenges, just as it can write the code for the Nine Coins versions we've been looking at. But the code isn't any easier to understand. In particular, flipping a coin still involves a conditional construction, much like the one in Claude's simple program (Listings 1 and 2).

The trouble is that spreadsheet values share only some of the behaviors of physical objects. In particular, a physical object like a wooden block can be flipped, but a spreadsheet value cannot.

5.1. A fantasy coding system would have computational objects that obey the naive physics of real objects

Here are some of the familiar affordances of common physical objects, like children's blocks:

- They are always visible.
- They don't move or change on their own.
- They can't easily be modified, but they can be rotated.
- They exist in places, and can be referred to by location.
- Groups of locations can be referred to (grids, rows, columns, etc.)

A coding tool for working with such objects would write scripts using natural references ("flip all the coins row 1", "turn all the coins in the grid to heads") and attach them to buttons. The script attached to a button could be shown by the tool as part of an explanation, or displayed by the user. It's easy to see how Nine Coins could be implemented in a system like this, and that it would be easy to understand, and have confidence in, the implementation.

(Code that controls the appearance of objects would not normally be shown. We've not discussed this, but such code is of course there. In the simple program in Table 1 there's styling on the <div> where the coins are shown. This is potentially a tricky matter for our fantasy system, because physical objects don't have styles.)

5.2. A more complex example for the fantasy system

The possibilities of this fantasy system can be dramatized with another example from (Mason, 1995):

> A cube is placed on each square of a chessboard. The faces of the cubes are congruent to the squares of the board. Each of the cubes has at least one black face. We are allowed to rotate a row or column of cubes about its axis. Prove that by using these operations, we can always arrange the cubes so that the entire top side is black.

As discussed in (Lewis, 2022), it's possible to solve this problem fairly easily (for a programmer, using an AI tool) using three numbers to represent the orientation of each cube. One of the numbers indicates the upper face of the cube, and the other two indicate two other faces. This allows the rotations to be simulated quite simply. (As discussed in the reference, two face numbers would be enough, but one has to assume the handedness of the cube numbering to make that work.)

A solution in the fantasy system would be much easier to understand, though. The computational cubes would be rotated in the same way as real cubes, with no representational trickery required.

5.3. Another example is the swapping problem

Novices often have trouble swapping the value of two variables in common programming languages. As programmers learn, a common attempt,

```
X = Y
Y = X
```

fails:

```
(start with X contains 2, Y contains 3)
X=Y now X contains 3, Y contains 3
Y=X now still X contains 3 and Y contains 3
```

To avoid this one can use a third variable, like this:

```
(start with X contains 2, Y contains 3, T contains anything)
T=X X contains 2, Y contains 3, T contains 2
X=Y X contains 3, Y contains 3, T contains 2
Y=T X contains 3, Y contains 2, T contains 2
```

The trouble with the first attempt is that with ordinary computational objects, the first statement destroys the value of X. That doesn't happen with physical objects. If values are represented by wooden blocks, I can move them around without fear of destruction.

But an emergent problem with physical objects is that when I move the block in location Y to location X, now I have two objects there. How do I move just the original block to location Y? That can be expressed in natural reference as "move the block that was in X to location Y".

Another swap technique, that works only for numbers, highlights another failure of physical intuition:

```
(start with X contains 2, Y contains 3)
X=X+Y X now contains 5; Y contains 3
Y=X-Y X contains 5; Y contains 2
X=X-Y X contains 3; Y contains 2
```

This works, but it is far from easy to see that it does. On the face of things, after the first step, the value 2 has completely disappeared!

In the fantasy block system the solution would be something like

```
Move the block in Y to X
Move the old block in X to Y
```

or even just

```
Swap the blocks in X and Y
```

Note: Some programming languages have added a swap operation to deal with this problem. For example, in Python one can form and assign tuples, using commas:

```
X,Y = Y,X
```

But then this notation has to be explained.

One might think that this little problem, swapping, is too simple to warrant attention. But a quick Google search for "swapping two variables" will show a great many people asking about it, and posting suggestions.

5.4. The intent of the fantasy project is not to make it easier to write programs, but to make them easier to explain

It would likely be easier for a non-programmer to write the natural references that are imagined in the fantasy system, than references in current languages. But that's not the point. Rather, the idea is that they don't have to write them at all. The coding tool writes the references. The coding tool is also responsible for ensuring that the actual implementation code it writes (in some conventional language) honors the explanatory description.

This approach follows a suggestion of Antranig Basman that AI tools open up the design space for communities like PPIG. That's because the tools can do the work of translating from novel representations to existing ones. Thus there's another PPIG Design opportunity: make something like this fantasy coding system actually work, leveraging the resources of AI coding tools.

Basman's work contains many other relevant reflections on the stuff of computing, and how much computational stuff suffers in comparison to physical stuff (Basman, 2016, 2017). In the same vein, Lida Kindersley, in a contribution to the joint workshop of PPIG and Art Workers Guild in 2020 (`https://ppigattheartworkersguild2018.wordpress.com/2020/12/01/postcard-from-lida-kindersley/`) reported that she found working as an artist/craftsperson in stone was "much more fun, much more direct, and much more surprising (emphasis in the original)" than working with computational stuff.

6. This fantasy project likely raises more questions than it answers

These ideas only deal with some pretty simple issues with data. How would a system like this cope with the complexity of control structures?

The Forms2 system (Ambler & Burnett, 1989) uses an approach in which different layers of a recursive process are shown simultaneously, rather than earlier values being replaced by later ones. That involves copying, not a simple process for physical objects, but the visibility in the scheme could help people understand what's happening.

How much "real programming" could be done this way? That remains to be seen. Computer scientists were initially baffled that something as obviously limited as spreadsheets could be interesting to anybody. But spreadsheets have of course been wildly successful.

More generally, sacrificing expressiveness for other values can be worthwhile. (Kell, 2009) notes that programs in languages that aren't Turing complete can be much easier to reason about than others. This of course is quite relevant to the understanding problem we are considering.

7. Applying the cognitive dimensions approach to these design challenges

Little if any extension is needed to accommodate the design ideas we've considered so far. Rather, one needs only to consider how existing dimensions apply to representations intended to support code understanding. Since the key suggestion here is that understanding requires mapping what code does to intuitions the user already has, this is mostly about closeness of mapping. There's the (minor) difference that the intuitions one wants to map to needn't be specifically in the user's problem domain, as is looked for in other applications of the closeness of mapping dimension. They just have to be things the user is used to thinking about.

An issue we haven't discussed, because the examples we've considered have been so simple, is understanding systems that likely have to be understood in parts. For example, a program that uses a complicated function can be understood by separating the problem of understanding how the function works, from the problem of understanding the system as a whole.

As in this example, such decompositions often themselves use technical apparatus that has to be understood itself, like parameters and different forms of argument passing. Perhaps a kind of cognitive dimension could reward designs for which the means of combining parts are themselves simple. While aspects of this problem may be covered in the existing system of dimensions, for example "hard mental operations", and "progressive evaluation", some more specific attention might be given to this aspect of understanding.

An example of an innovation that could be explored in this connection is Lanier's phenotropic programming (Lanier, 2002, 2003)(Lanier, 2002, 2003; see also (Lewis, 2018). In this conception, system components communicate by measurement of attributes of one another, rather than by a protocol like argument passing. Possibly this approach engages users' intuition better than protocols do.

So here are two more PPIG design opportunities: (1) Are there other concerns in code understanding and explanation that should be addressed by cognitive dimensions? (2) Are there ways of coupling the components of complex systems that are easier to understand than existing schemes?

8. Did I ask my AI tool the right question?

There's one more aspect of the program evaluation problem that should be addressed. I may be given a program that implements exactly what I asked for. It would pass any tests my tooling would devise, and an explanation of it would make me confident that the code does what I asked for. But it could be wrong, because I didn't ask for the right thing.

I experienced exactly this problem, in using an AI tool to create an agent-based simulation of similarity-based imitation. I wanted abstract agents that would carry out actions in such a way that they would imitate the neighbor that was most similar to themselves, based on their history of actions. The AI tool I was using easily wrote the simulation for me, but the results didn't seem right. I spent a lot of time (and used my knowledge of programming) trying to pin down the mistake. I assumed the tool had gotten something wrong. Eventually I realized that the program was exactly what I had asked for, but I had asked for the wrong thing. I told the tool to use cosine similarity in comparing action histories, whereas I should have used a different measure. Cosine similarity evaluates the angle between two vectors, and quite different vectors can have zero angle between them, for example [1,1,1] is parallel to [9,9,9].

In a later interaction I asked Claude whether it thought the cosine measure was appropriate: `https://claude.ai/share/c69ce235-826e-4a03-812f-70597b57f31b`. It provided a helpful discussion that raised the issue I had found, though it did not assert that I was wrong to use cosine. In another interaction I asked Claude to create the simulation, without specifying the similarity measure. Claude used a better measure, but it didn't comment on any connection between my goals and that measure. So there's no indication that it reasoned about my goals, other than in a very general way, in making its choice.

There's another PPIG design opportunity here. Is there a way to prompt AI tools that makes it more likely that the tool will provide feedback on what was asked, rather than just complying?

9. Conclusion

We've discussed only a tiny corner of the wide landscape of program evaluation. In an earlier PPIG paper (Lewis, 2017) I argued that the reason that the cognitive dimensions idea is so powerful, and such "rigorous" methods as Randomized Controlled Trials are so unhelpful, is that programming, as a design space, is cut through by an enormous number of consequential distinctions. For example programs like Nine Coins ought actually to work, but it's not worth expending enormous effort to be sure they do. For the programs that control a car's braking system, it is worth making that effort. So completely different approaches, such as formal methods, and technically sophisticated test coverage disciplines, are called for in such situations. How to make techniques like these work well is a different design opportunity for PPIG. It's also one where AI tools will play a big role, as they can in the work discussed here.

10. Acknowledgements

I thank Antranig Basman and Owen Lewis for many important ideas. I thank the PPIG community, and the Hanse-Wissenschaftskolleg, Delmenhorst, Germany, for intellectual support over many years.

11. References

Ambler, A. L., & Burnett, M. M. (1989). Visual languages and the conflict between single assignment and iteration. In *1989 ieee workshop on visual languages* (pp. 138–139).

Basman, A. (2016). Building software is not a craft. In *Proceedings of the psychology of programming interest group* (p. 142).

Basman, A. (2017). If what we made were real. In *Proceedings of the psychology of programming interest group.*

Blackwell, A. F., Britton, C., Cox, A., Green, T. R., Gurr, C. A., Kadoda, G., ... Young, R. M. (2001). Cognitive dimensions of notations: Design tools for cognitive technology. In *Cognitive technology: Instruments of mind: 4th international conference, ct 2001 coventry, uk, august 6–9, 2001 proceedings* (pp. 325–341). doi: 10.1007/3-540-44617-6_31

DiSessa, A. A. (1993). Toward an epistemology of physics. *Cognition and instruction*, *10*(2-3), 105–225.

DiSessa, A. A. (2001). *Changing minds: Computers, learning, and literacy.* Mit Press.

diSessa, A. A., & Abelson, H. (1986). Boxer: A reconstructible computational medium. *Communications of the ACM*, *29*(9), 859–868.

Draper, S. W. (1986). Display managers as the basis for user-machine communication. In *User centered system design* (pp. 339–352). CRC Press.

Gentner, D. (1983). Structure-mapping: A theoretical framework for analogy. *Cognitive science*, *7*(2), 155–170.

Green, T. R. (1989). Cognitive dimensions of notations. *People and computers V*, 443–460.

Kell, S. (2009). The mythical matched modules: overcoming the tyranny of inflexible software construction. In *Proceedings of the 24th acm sigplan conference companion on object oriented programming systems languages and applications* (pp. 881–888).

Lanier, J. (2002). The complexity ceiling. In J. Brockman (Ed.), *The next fifty years: Science in the first half of the twenty-first century* (pp. 216–229). Vintage.

Lanier, J. (2003, November). *Why gordian software has convinced me to believe in the reality of cats and apples.* edge.org. Retrieved from `https://www.edge.org/conversation/jaron_lanier-why-gordian-software-has-convinced-me-to-believe-in-the-reality-of-cats`

Lewis, C. (1988). Why and how to learn why: Analysis-based generalization of procedures. *Cognitive Science*, *12*(2), 211–256.

Lewis, C. (2017). Methods in user oriented design of programming languages. In *Proceedings of the 28th annual conference of the psychology of programming interest group (ppig 2017).*

Lewis, C. (2018). Phenotropic programming? In *Proceedings of the 29th annual conference of the psychology of programming interest group (ppig 2018).*

Lewis, C. (2022). Automatic programming and education. In *Companion proceedings of the 6th international conference on the art, science, and engineering of programming* (pp. 70–80).

Lewis, C. (2023). Large language models and the psychology of programming. In *Proceedings of the 34th annual conference of the psychology of programming interest group (ppig 2023)* (pp. 77–95).

Lewis, C. (2025). Artificial psychology. *Synthesis Lectures on Human-Centered Informatics*.

Mason, J. (1995). Exploring the sketch metaphor for presenting mathematics using boxer. In *Computers and exploratory learning* (pp. 383–398). Springer.

Minstrell, J. (1982). Explaining the "at rest" condition of an object. *The physics teacher*, *20*(1), 10–14.

Russ, R. S., Coffey, J. E., Hammer, D., & Hutchison, P. (2009). Making classroom assessment more accountable to scientific reasoning: A case for attending to mechanistic thinking. *Science Education*, *93*(5), 875–891.