

© 2024 Yoshee Jain

**TOWARDS IDENTIFYING DOMAIN-SPECIFIC PROGRAMMING PLANS  
AT SCALE: NEEDS, CHALLENGES, AND SOLUTIONS**

BY

YOSHEE JAIN

THESIS

Submitted in partial fulfillment of the requirements  
for the degree of Bachelor of Science in Computer Science  
in the Grainger College of Engineering of the  
University of Illinois Urbana-Champaign, 2024

Urbana, Illinois

# Acknowledgments

I have had the privilege of pursuing research as an undergraduate student, and I am immensely grateful to everyone whose support and encouragement have been invaluable to me on this journey.

First, I would like to extend my greatest appreciation to my mentor, Dr. Katie Cunningham. Your guidance has been instrumental in shaping my career as a researcher and helped me grow not only academically but also personally. Having the opportunity to be mentored by you has been my one of my greatest feats!

I have also had the great fortune of working on multiple research projects in my academic journey. I want to thank Dr. Koustuv Saha and Dr. Eshwar Chandrasekharan for their guidance and mentorship and for providing numerous opportunities to hone my skills and evolve as a researcher.

I express my heartfelt gratitude to my mother, Bhavna Jain, who has stood by me as I weathered countless challenges and made my new home at the U of I. Your pride in my successes, no matter how big or small, your inspiring words, and your belief in me, even when I doubted myself in the face of obstacles, always helped me thrive.

I am also thankful to my friends, Ineassa Dassani and Jiya Chachan; my family away from home. Your listening ear during countless sessions of complaints and rants and your comforting words during difficult moments have fueled my resilience. Thank you for reminding me to also embrace the joy of being a student and living!

I am also grateful to the TRAILS lab members whose contributions were invaluable to the completion of this document. A shoutout to Arif whose suggestions were vital in shaping the ideas presented in this document.

I also want to thank the CS STARS program for introducing me to the world of research and the University of Illinois for providing me with unending opportunities to pursue and showcase my work. While at it, I must thank the Siebel Center for Computer Science, my second home in Champaign-Urbana, where I've spent countless days and nights. Its existence has made my productivity possible.

Thank you, everyone!

# Table of contents

Chapter 1	Introduction .....	1
Chapter 2	Related Work .....	3
Chapter 3	Interview Study Methodology .....	6
Chapter 4	Interview Study Results .....	8
Chapter 5	Discussion .....	15
Chapter 6	LLM Plan Generation Methodology .....	18
Chapter 7	LLM Plan Generation Evaluation .....	20
Chapter 8	Discussion .....	26
Chapter 9	Implications for Future Work .....	28
Chapter 10	Conclusion .....	31
References	.....	32

# Abstract

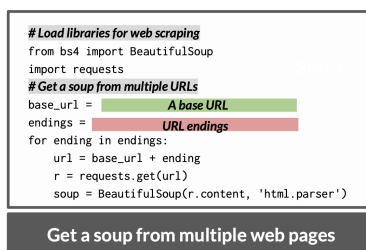
Knowledge of programming plans, which are stereotypical code patterns that achieve a goal, is key to programmers' ability to write programs. When computing educators are armed with a set of programming plans, they can take advantage of instructional techniques that may accelerate their students' learning. However, plan identification is an effortful process that is not well documented in prior work. Moreover, existing sets of plans are primarily drawn from introductory programming content, so plan-based instructional techniques are not yet possible for many programming application domains, like web scraping, data analysis, or machine learning model creation. Through interviews with ten computing educators who have identified novel plans, we describe the current plan identification process, including the plan components instructors value, the characteristics by which plans are judged, and key challenges. Building on these results, we present an LLM-supported plan identification pipeline and evaluate it in comparison to a set of expert-identified plans primarily in the web scraping domain. We also extrapolate this pipeline for other domains of interest to non-majors including data processing, visualization, and machine learning. Due to the absence of pre-identified programming plans in these domains, we evaluate these plans against code from StackOverflow and Github due to their widespread use. We find that using LLMs as a supportive tool in the design of an automated plan identification system to identify plans that meet the success criteria established by computing education researchers is a promising avenue. Our work provides important implications for automating plan identification, including where and how processes may be automated, and how these processes can be augmented with an instructor in the loop.

# Chapter 1

## Introduction

Programming plans have long been considered a significant aspect of programming knowledge [1], [2]. These plans, also known as programming patterns [3], [4] or code idioms [5], are short, stereotypical solutions that can be adapted for solving different types of problems. The ability to recognize programming plans in code differentiates novices from experts [1], and knowledge of plans seems to underlie effective code writing [6]. Moreover, instructional strategies successfully utilize plans to improve the problem-solving and abstraction skills of students (e.g. [4], [7], [8]). However, despite the rich set of works describing plans from introductory programming content (e.g. [3], [4], [9]–[11]), plans have rarely been used to support instruction beyond the introductory level. Yet, recent work has shown that domain-specific plans can be effective in teaching and motivating a variety of students when they learn application-focused topics, like CS majors learning software testing [12] and conversational programmers learning web scraping [7] (see Figure 1.1). These promising results imply that identifying plans in a wider variety of domains can support diverse learners to reach a greater range of learning objectives.

An obstacle to identifying domain-specific plans is the opaqueness of the plan identification process. Most works that identify plans in introductory programming do not describe their process and omit crucial details necessary to replicate the process in another domain. A more transparent process for plan identification may help experts identify plans in application-focused domains to support learners interested in different applications. Particularly, understanding which *components* of a plan are most important for instructors, which *characteristics* instructors look for when evaluating programming plans, and what are the primary *challenges* during plan identification would unravel the current plan identification process and support the identification of domain-specific plans.



```
# Load libraries for web scraping
from bs4 import BeautifulSoup
import requests

# Get a soup from multiple URLs
base_url = A base URL
endings = URL endings
for ending in endings:
    url = base_url + ending
    r = requests.get(url)
    soup = BeautifulSoup(r.content, 'html.parser')
```

Get a soup from multiple web pages

Figure 1.1: A programming plan educators have already identified in the domain of web scraping with BeautifulSoup [7].

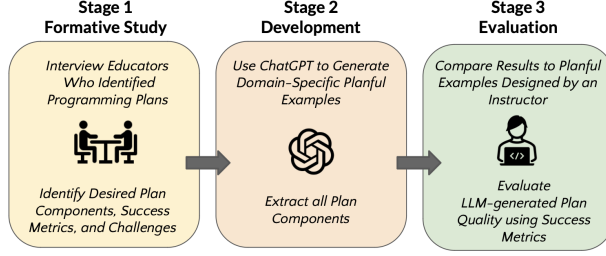


Figure 1.2: An overview of the stages of our study.

Another reason for the lack of domain-specific plans is the tedious and effortful nature of the identification process. While there have been some efforts to automatically identify common code pieces, these approaches do not consider pedagogical aspects of identified constructs [5], [13], [14] and likely do not meet the needs of instructors. However, with the latest developments in technologies such as large language models (LLMs) that can effectively explain and generate code [15], it might be possible to design tools to reduce the workload of instructors working on plan identification and enable plan identification beyond introductory content.

To address these concerns for identifying domain-specific plans, we seek the answers to two research questions:

- **RQ1:** What are the components, characteristics, and challenges relevant for the plan identification process?
- **RQ2:** How can LLMs (e.g. ChatGPT) support the identification of domain-specific plans?

To answer RQ1, we conducted semi-structured interviews with 10 computer science education researchers with varying levels of experience with programming plans. To answer RQ2, we proposed a design prototype for an LLM-powered domain-specific plan identification pipeline and evaluated it via quantitative and qualitative metrics based on findings from our interview study. Our work reveals that plan identification requires various kinds of expertise, and automated systems powered by LLMs can support plan identification. Our key contributions are as follows:

- A specific description of components of programming plans and criteria for their evaluation.
- A deep understanding of the needs and challenges of computing educators in their current plan identification process.
- An exploratory analysis of an automated plan identification pipeline and design implications for a sociotechnical system supported by LLMs for plan identification.

# Chapter 2

## Related Work

### 2.1 Improving instruction with programming plans

Informed by schema theory, Elliot Soloway and his students first described *programming plans* in the 1980s [16]–[18]. They defined plans as chunks of code that achieve particular goals, like guarding against erroneous data or summing across a collection. They provided evidence that knowledge of plans can represent progress in learning programming [19], and student errors could be explained in terms of misunderstanding plans [20], [21] and plan composition errors [22]. Inspired by this, educational designers have used programming plans as scaffolding (assistance for learners [23]) to enable novice programmers to write more code than they would be able to on their own and designed plan-based curricula and tools [24], [25]. A variety of classroom instructional techniques organized instruction on common patterns [4], [26], [27], used plans to inform the design of instruction [28], and evaluated students’ expertise on their ability to recall plans.

However, until recently, these instructions were confined to introductory courses as most well-established sets of plans were identified from introductory programming content (e.g., [3], [11], [29]). To better support novice learners meet their learning objectives in a specific application area, we need to identify programming plans in domain specific contexts. A common construct, namely domain-specific languages (e.g. Boa for mining software repositories [30]), are frameworks designed to write code in application-focused areas in order to facilitate programming learning for students with no prior programming experience [31]. While this approach intends to scale the use of programming in application-focused domains for a novice audience, [32], they lack the scaffolding needed for novices to get equipped with their use. Specifically, due to the lack of clarity of the subgoals involved in crafting a solution to the problem, student oftentimes find it challenging to write code for addressing the problem at hand. The design of environments that provide visual block-based support for domain-specific programming languages [33] similar to block-based programming in introductory computer science [34] using environments like Scratch [35] would come with the same challenges as in block-based programming. Frequently, block-based programming languages do not involve writing code but rather putting blocks or elements together to design a solution to the problem. It is important to note that this leads to the perception that programming is difficult. It also creates a larger gap between student learning and the real world of programming that mostly involves writing code from scratch. [36]. In light of these shortcomings, recent work proposed the purpose-first programming approach [7] to support conversational programmers [37], [38], showing that plan-based scaffolding can support learners with diverse learning goals. While earlier systems focused on supporting introductory programming learning, purpose-first



programming included plans drawn from more applied coding topics, like web scraping, and made such programs accessible to novices. With a more efficient plan identification process, we can more easily extend such promising approaches to new programming application areas.

## 2.2 Identifying patterns in programs automatically

In software engineering, work in the mining of *code idioms* [5], [13], [14] uses statistical natural language processing to extract semantically meaningful pieces from a code corpus. This approach has been extended to support program synthesis using these idioms [39]. In addition, techniques for representing code in ways that facilitate machine learning methods (e.g. code2vec [40]) expand the toolbox for code clustering methods. While these systems utilize a large amount of real code on platforms like Github and StackOverflow, they do not include the same pedagogical concerns experts consider when identifying programming plans, limiting their applicability to instruction. Understanding these pedagogical concerns better can also enable these systems to be incorporated into instructional processes.

Recent work in HCI has displayed commonalities in code samples in ways that support learning and instruction. Glassman et al. clustered student programming assignment data to reveal common learner misunderstandings in a way instructors can view [41]. Glassman et al. visualized varieties of API calls from StackOverflow data [42] to reveal common use cases to learners [43]. While these patterns can aid instructors, these works do not attempt to produce high-level code plans abstracted from particular implementations. Forming concrete guidelines on how experts perform this abstraction process can inform the design of systems that support instruction.

## 2.3 Exploring code interpretation capabilities of large language models

While large language models (LLMs) have not been used for identifying programming plans, their capabilities at processing code input have been shown through multiple studies. In addition to solving programming exercises of varying topics and difficulty with high accuracy [44]–[46], LLMs have been employed to generate programming exercises in custom contexts [47] and novel assignment types that produce code using student explanations as input [48], [49]. Moreover, using LLMs to generate code explanations produced simpler and more accurate explanations compared to student submissions [50]. Jury et al. further evaluated the code explanation capabilities of LLMs by generating worked examples, which are step-by-step solutions used for demonstrating the problem-solving process of an expert to a student, through expert assessment and a large-scale user study, showing that LLMs can not only generate code but also generate explanations on different levels of abstraction to explain the program to a novice learner [15]. However, some studies also observed that LLM-generated code may include structures unfamiliar to novices [15] or fail to follow industry best practices [51]. Moreover, in addition to the downsides of domain-specific languages proposed in the prior section, the primary objective of identifying plans and designing a system is to support the identification of these helpful learning constructs at scale to address the needs of a diverse set of learners. These learners, namely conversational programmers, desire to learn about specific application-focused domains. Domain-specific languages are also effortful to design and with the rise in the applications of computer science, it becomes challenging to create such languages for every possible area. The design of an LLM-supported

identification system would be more sensitive to the changes in the world of computer science including up-to-date datasets for identification and evaluation of programming plans leading to an easier approach for scaling the creation of programming plans motivating faster and better learning experiences for students.

## Chapter 3

# Interview Study Methodology

To understand the current process of programming plan identification by educators, we conducted semi-structured interviews with ten computing instructors (see Table 3.1) who had performed plan identification as part of their prior work. Mirroring the approach of Fowler et al. [52], we recruited instructors who authored a computing education research publication where plan identification was a part of their methodology or results. Computing education researchers possessing experience working with programming plans are an ideal source to learn instructional best practices because this population consists largely of instructor-scholars: those who not only teach computing but are also informed by research and recent practices in computing education. This allowed us to discuss their concrete programming plan identification experiences focusing on their pedagogical perspectives, rather than have participants speculate on the process in general.

We conducted online semi-structured interviews that ranged between 30 minutes and one hour. We first asked questions about the participants' backgrounds to collect basic demographic information and understand the extent of their experience teaching and researching with programming plans as well as their understanding of these plans. Next, we asked participants to describe the specific activities they undertook to identify programming plans in the paper they authored. This included questions about what they were looking for when they identified programming plans, the resources that were involved in this process, and the procedures they undertook. We also posed questions to gain insights into the difficulties the interviewees faced when they were identifying programming plans. Lastly, we prompted a discussion about how they would attempt to

Table 3.1: Participant Demographics.

	Years in CSEd Research	Years in Plans Research	Years in CS Instruction	Teaches CS1?	Uses Plans in Instruction?	Industry Experience?
P1	10-20	1-3	10-20	Yes	Yes	Yes
P2	5-10	4-6	10-20	Yes	Yes	No
P3	5-10	4-6	10-20	Yes	Yes	Yes
P4	1-5	1-3	5-10	No	Yes	Yes
P5	1-5	1-3	1-5	Yes	Yes	No
P6	10-20	1-3	20+	Yes	Yes	Yes
P7	5-10	4-6	20+	No	Yes	Yes
P8	1-5	1-3	5-10	Yes	Yes	Yes
P9	20+	20+	20+	Yes	Yes	No
P10	20+	20+	20+	Yes	Yes	Yes

identify programming plans in a new topic area they were unfamiliar with to gather these instructor-scholars' perspectives for design implications.

We used a transcription service to transcribe the video recordings and used Dedoose, a qualitative analysis software, for coding the transcripts with an inductive, reflexive thematic analysis process [53]. During coding, we highlighted the connections between the beliefs of our participants and their choices in the pattern identification process.

## Chapter 4

# Interview Study Results

The educators we interviewed described a variety of details about the approaches they utilized and the objectives they hoped to reach during the plan identification process. They also shared challenges and suggestions for potential improvements.

Note that while we use the term “programming plan” in this paper to clarify our focus on relatively small coding chunks rather than design patterns or architectural patterns, many of the educators we spoke with used “plan” and “pattern” interchangeably, or even preferred the term “pattern.”

### 4.1 Components of Programming Plans

Our participants looked for a variety of items when they identified programming plans. In this section, we discuss all plan components mentioned by two or more of the instructors we interviewed (see Figure 4.1 for a summary).

#### 4.1.1 Name

For some instructors, programming plans must have a name. As P5 put it, “[plans are] useful common problems that people name.” The plan name is often reflective of what the plan accomplishes. P6 believed that plans could have “generic names” or “problem-specific names”, but that the problem-specific names

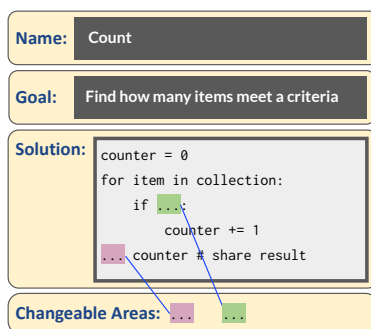


Figure 4.1: The components of a programming plan mentioned by our interviewees, illustrated with a common introductory programming plan.

better supported students during problem-solving: *“it was also important for [students] to reflect, ‘Okay, if I need a counter [plan], what exactly am I counting here?’ ”* The name connects the plan to the types of problems it can solve.

### 4.1.2 Goal

Some of our participants indicated that plans should have goals. This was a key part of the plan for P3, who believed *“a plan has to accomplish a specific goal. That was already the starting point.”* For some, a goal emphasizes the deep, underlying structure of certain problem types, even if implementation details differ in practice. This led one instructor to solely focus on goals during plan identification, because *“according to the programming language you have, the plan can vary. There might be big differences [in syntax], but the goal is the same.”* (P2)

### 4.1.3 Solution

For nearly all our participants, a programming plan included some type of solution to the problem it typically solved. These solutions could be in a variety of forms. Two of our participants thought that implementation in a specific programming language is a key part of a plan. For example, P3 said, *“The code cannot be detached from the way the plan is built.”* P1 agreed, stating *“In order to code in that language in the best way, it’s important to understand some of those language-specific patterns.”*

In contrast, some participants used pseudocode or natural language to describe solutions, in the interest of communicating ideas that would potentially cross languages. In addition to language-specific plans, P1 argued that *“you should be able to explain them abstractly, independent of the syntax of the language”* and thus *“in pseudocode you should be able to explain what’s going on and talk about what it does and why it gets used.”* However, for P6, writing plans in pseudocode got in the way, and they re-wrote plans in natural language: *“It’s really not about the notation...so, for the patterns I went really for just plain English.”*

### 4.1.4 Changeable Areas

Two of our participants used a template-based approach where specific changeable areas in the plan are highlighted. P1 described their approach: *“we use pseudocode, so we with dot, dot, dot on all the [changeable] parts just to highlight, here’s the pattern part”.* Similarly, P6 believed that it is crucial to have *“clear instantiation points”* in the solution for the learners while designing plans.

## 4.2 Characteristics Used to Judge a Good Programming Plan

While different educators had different ideas about what they value in a “good plan”, we found four themes common among many of our participants: commonality, usability, the level of abstraction, and appropriateness for learning goals.

### 4.2.1 Plans Should Be Common

All but one of the educators we spoke with considered the frequency with which a programming plan is used in practice to be a measure of its worth. However, there were differences in how they judged commonality and how much value they gave to this measure.

Some respondents believed that the definition of a programming plan itself mandates that any plan be frequently employed in standard practice. In other words, the commonality of programming plans is “*inevitable*” (P4).

For it to be a “pattern” it has to be seen in the world. [Plan identification] isn’t about creativity, but about exploration and recognition. (P10)

While the instructors we interviewed generally agreed that a good plan should be used frequently in practice, they had different ideas about the *type* of practice that should be used for this measurement. One of our interviewees urged caution in what types of programs were used to judge commonality:

It’s not something where you necessarily want to have a popular vote on things. Just because more people eat at McDonald’s than a fancy restaurant doesn’t mean that McDonald’s is better food. (P1)

P1 described two types of commonality, one that draws only from the classroom, and one that draws from professional and other programming practice: “*One is, do we see it a lot in the textbooks and or assignments that we looked at? There’s another kind of common, which is when I think about all the programming that happens in the world, is it important in that context also?*” P6 chose plans based on their frequency in class assignments: “*I was not going to go for patterns if I can’t immediately think of at least three or four exercises or questions for the assignments that would need this pattern.*”

#### 4.2.2 Plans Should Be Usable

Most of our interviewees believed a critical quality of a programming plan is that it can be easily applied to a variety of situations that share similar goals. P10 said, “*The solution has to be easy to put into practice (with practice).*” This requires that the plan be adaptable in a way that the learner understands. “*Understanding how to tune it should not be the issue*” (P2).

The importance instructors placed on usability stemmed from their reasons for teaching with programming plans in the first place. They believed that programming learners often lack the ability to interpret a problem statement and to adapt a similar solution without guidance. As P7 said, “*The bridge between there [the problem] and code is too large.*”

A good programming plan supports students to span this gulf. P1 described plans as an “*important step between understanding the syntax of a language and understanding how to do problem solving.*” P9 agreed, saying “*knowing the pattern helps getting to a good solution, an expert solution.*” In the same way that interface designers narrow the “gulf of execution” between a user’s goal and how to achieve that goal [54], instructors believed that programming plans should facilitate learners’ ability to design and implement programming solutions.

#### 4.2.3 Plans Should Have the Right Level of Abstraction

Finding the balance between concrete and general was key for most of our interviewees when identifying programming plans. Pinpointing the right level of abstraction or “*granularity*” (P3) achieves potentially conflicting goals: ensuring that the plan works across multiple contexts, while also providing sufficient concrete detail to implement the plan in practice.

If you list too many [plans], you lose the idea of the schema, the schemata to it. If it is too abstract, you cannot use it in practice. Students have difficulties in grasping the idea and using them. (P2)

P2 cautioned that plans must be applicable in multiple contexts, saying *“If it is too specific, it’s not recurrent enough.”* P2 also reasoned that it is essential to find a *“good trade-off”* so that students don’t think about plans *“as a recipe expected to be ready to be used.”*

The educators we interviewed believed that the right level of abstraction was impacted by the intended audience for the programming plan. Most participants thought it is essential to *“teach novices and experts patterns very differently”* (P1). This is because *“advanced students can handle more abstract explanations”* (P10).

P1 believed that as students gather more expertise, it could be useful to then introduce *“more advanced defined patterns”* and emphasize *“the wrong ways to do things and why this solution is better”*. By contrast, a good plan for beginners is relatively *“concrete”*, and focuses on *“how you solve this problem”*. P3 shared that *“it doesn’t make sense to talk about plans in general”* since for *“high school students, even declaring a variable is a plan and later they move on to complex plans”*.

#### 4.2.4 Plans Should Be Appropriate for Learning Goals

One of our interviewees highlighted that from a *“teaching perspective”* (P8), a quality plan should be appropriate to the learners’ goals. They elaborated, *“if you’re in the beginner level, you definitely want to understand what is a programming language, how do I do stuff?”* For those learners, better plans illustrated the operation of language features, to *“help [them] build an idea of [their] notional machine”* [55]. However, for more advanced students, *“you probably want to prepare for industry,”* so ideal plans would be representative of the *“most recent programming paradigms”* (P8).

### 4.3 The Process of Plan Identification

Our interviewees took diverse approaches to identify programming plans. While some educators focused on techniques that helped them gather insight from common practice, others relied on their personal expertise. Most of the educators we interviewed leveraged a combination of both these strategies.

#### 4.3.1 Viewing Programs and Problems

Echoing the belief that *“patterns are ”mined” from the practice”* (P10), many educators looked at some sort of code, problem statement, or instructional material during their plan identification process. More specifically, they looked at GitHub repositories (P7), programs written by industry professionals which may or may not include their own code (P7, P8, P10), textbooks (P1, P2, P5), and other instructional material including lecture notes, assignments, student programs, and testing material (P1, P2, P4, P6, P9).

*“There was a lot of just paging through textbooks, either physical or digital versions and just looking at code, just trying to see if there’s something we haven’t seen before.”* (P1)

As the educators reviewed this content, their own expertise and experience played a role in finding the programming plans. *“You know it when you see it,”* said P7. P10 highlighted that gathering plans from



common practice is not independent of the identifier’s own perspective: *“there’s a personal selection path that always happens.”*

### 4.3.2 Discussions

Our interviewees highlighted the importance of collaboration in the plan identification process. *“The teamwork was very helpful,”* said P9. Collaboration could involve discussions with co-authors (P1, P2, P3, P7, P9, P10), co-instructors (P8, P9), TAs (P4, P8), paper reviewers (P5), study participants (P4), other researchers (P7, P9), and developers in the industry (P8).

These discussions occurred across different stages of the plan identification process including from when candidate plans are first presented, to when they are refined into their final form or removed from consideration.

While identifying patterns, P9 highlighted that they talked about most of the components of the plan including *“what to name a pattern”*, *“how to write this pseudocode so it can be generalized to different languages”*, and the difference between language constructs and plans, i.e., *“maybe this is not a plan, it’s just an idea how to do it right.”* P1 highlighted that they deliberated about which plans to include in their final inventory: *“we just talked about them, be like, “Yeah, that’s really kind of arcane. Maybe that’s not a general purpose pattern.” or “Oh yeah, that makes a lot of sense.”*

### 4.3.3 Literature Review

Almost all of our participants conducted a literature review to take inspiration from the existing patterns and tailor it to their vision. *“Having a bunch of literature of course helps a lot,”* shared P8. *“We also took those [plans from literature] because they were broadly researched, people talked about it, so we had some benchmarks.”*

But finding a plan in the literature wasn’t always the end of the search. P8 shared that they don’t take plans from the literature as the final word.

Someone telling you this is a good candidate for a programming plan, you still have to think, is it still a programming plan? Or maybe not. Is it only for this particular context? (P8)

### 4.3.4 Individual Design

All of our interviewees have teaching experience, and most of them have had long careers as instructors (see Table 3.1). Instructional expertise helps the plan identifier with aspects of plan identification related to transmitting programming plans effectively to an audience of learners. As *“plans are useful only if they can be communicated between humans”* (P1), it’s important for plans to be appropriate for the knowledge level of the audience. One of our interviewees also highlighted that a plan identifier needs to have relevant programming background or find it in someone else: *“...you need some experience in that practice or the ability to have discussions with people who have that experience”* (P10). These instructors believed that only those with expertise in a programming topic could shed light on the crucial issue of code’s intent. As P4 described such knowledge: *“Is this a pattern or is this just a construct of your language? Is this an unfortunate oddity of syntax, or is this actually something that means something to you in your domain?”*

Thus, one of the techniques employed by our interviewees was to draw on their experience in order to come up with plans on their own. By considering important problems and exploring *“how you would imagine best solving the problem”* (P9), the instructors generated first draft programming plans they could refine

and use in instruction. Envisioning assignments at the appropriate level and inferring relevant plans was a similar strategy for some: *“just also thinking about what’s an interesting ten line program that you can assign students to write.”* (P1) *“Now what other simple problem do I want?...Okay, let’s try to encapsulate that as a pattern”* (P6)

With this strategy, the instructors used their own expertise to replace a search through the literature or example programs: *“I knew that’s the way you solved the problem so, in a way, I didn’t need any kind of evidence.”* (P6)

## 4.4 Challenges

While our participants had all successfully identified programming plans to be used in an educational context, they did name some challenges they experienced in their process, or imagined challenges for future plan identification.

### 4.4.1 The Challenge of Choosing Plans from Practice

While our participants had ready access to resources that could be used to identify plans, like example programs and problems, they still found it onerous to translate this practice into plans. As P7 described, *“the challenge was trying to infer general characteristics from a large collection of specific examples.”* Another participant described this part of the process as *“tedious”* (P1).

This challenge is particularly relevant when educators “mine” plans from existing materials, a technique that the vast majority of our participants employed during plan identification (see Section 4.3.1). Understandably, determining which aspects of a given example are widely applicable enough to “count” as a programming plan is a difficult task, as the possibilities for writing programs are so extensive. Currently, plan identifiers lean on a combination of their own expertise and a manual search of examples as they tackle this challenge.

### 4.4.2 The Challenge of Finding the Right Abstraction Level for a Plan

Once plans are chosen, the challenge only begins. As mentioned previously, finding the right amount of abstraction is key in identifying good plans, according to computing educators. However, finding the appropriate balance of specific and general was a hurdle in our interviewees’ plan identification process.

The difficult part is probably defin[ing] the level. What’s the plan you are looking for, how general it is. Level of generality, abstraction to define it. (P2)

P7 illustrated their challenge by describing the conflicting goals of this process: a good plan should be *far enough above the level of code that it really does insulate students from syntax and mechanics, but not so far that it becomes useless*. Currently, educators make use of discussions with colleagues as one approach to manage this difficulty. On an even longer time frame, they may also refine their plans after seeing how they work in the classroom.

### 4.4.3 The Challenge of Identifying Plans in an Unfamiliar Domain

Identifying programming plans from more programming topic areas beyond introductory programming could enable better support of learners with diverse interests. However, when we asked our participants about

how they would identify programming plans in a new and unfamiliar domain, nearly every one said that it would be quite challenging. Our educators were quick to emphasize the importance of having programming expertise in the relevant topic when performing plan identification. *“Looking for material and try to recognize the schema without having expertise in the area, I think that would be impossible,”* said P2.

The key hurdle our participants anticipated in identifying plans from new domains was not finding common code, but understanding if that code implemented a meaningful goal. The task of identifying plans for educational purposes is multifaceted: it involves understanding *“what goals [students] need to achieve in this particular context”* and *“how much do they know at this point in their instruction”* (P3). While educators might understand what students are ready to learn, they cannot understand the important subgoals and tasks of a programming domain without the relevant expertise.

Several participants suggested collaborating with an expert from the relevant domain to understand the *“typical things [experts] need to do and how [those are] typically done”* (P2). This discussion was necessary to *“discover the intent behind code”* (P1), which is challenging to discover from looking at code alone. Some of our educators mentioned that using emerging AI tools like GitHub Co-Pilot (P6) or ChatGPT (P7) to generate solutions may also help with plan identification when plan identifiers lacked the relevant expertise.

## Chapter 5

# Discussion

In the prior section, we answered RQ1: *What are the components, characteristics, and challenges relevant for the plan identification process?* This section focuses on the implications of those results.

### 5.1 Educators Must Balance a Variety of Interconnected Values During Plan Identification

Based the findings presented in Section 4.2.3, the right level of abstraction was a key characteristic that helped educators determine the quality of a plan. In Section 4.4.2, we also acknowledge that creating a suitable level of abstraction was a challenge that most of our participants encountered. Educators identifying plans aren’t the only ones to face this challenge. In their study of a system for capturing developers’ programming strategy knowledge, Arab et al. found that developers had difficulty “finding the right scope for their strategy” to share with others [56].

Synthesizing our interviewees’ responses yields insight into why finding that correct level of abstraction is so difficult: A plan’s level of abstraction directly impacts three other desired characteristics, some positively and some negatively (see Figure 5.1). In essence, plan identifiers are performing a balancing act between a plan’s commonality, usability, and appropriateness for their students.

Abstraction level has a direct connection with the *commonality* of a plan. When a plan is more general, it

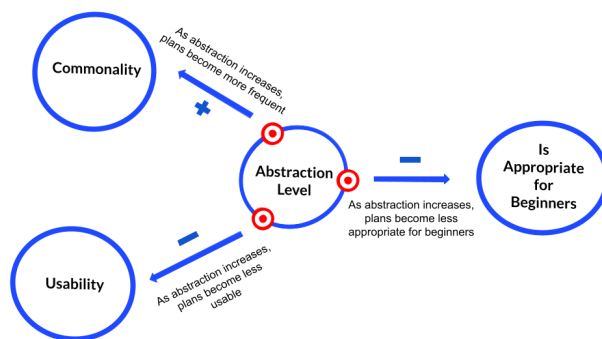


Figure 5.1: Relationships between the characteristics educators value when judging quality of programming plans for instruction.

will be seen in practice more often. However, abstraction level has an inverse relationship with how *usable* a plan is. The less specifically a plan is defined, the more interpretation a programmer needs to use to put the plan into use. For instance, with a more abstract plan, programmers have to make more implementation decisions on their own when they put the plan into practice.

The level of abstraction of a plan has a negative relationship with how appropriate it is for beginners, an important concern for educators specifically. While learners can benefit from instruction about abstract principles [57], novices have more difficulty applying more abstract representations [58]. Novices have less familiarity with programming, and therefore programs that an expert sees as similar may be seen as distinct by the novice.

## 5.2 Plan Identification for Education Requires More Than Finding Common Pieces of Code

As we note in Section 4.3.4, experience with the relevant programming domain as well as having instructional expertise is necessary in the process of plan identification to accurately identify useful plans suitable for learners. These types of expertise appear to be complementary yet distinct. Attempts to glean programming plans from domain experts alone seem unlikely to be successful without instructional expertise. When Arab et al. studied developers' ability to share strategic knowledge, about half of the developers they studied found it difficult to write strategies in a way that novice developers could understand [56].

Simply knowing that a piece of code is common in practice does not solve the problem of plan identification. While this may be a good starting place, instructional and programming expertise is necessary to create a quality programming plan. Those with programming expertise can answer questions about whether the pattern is a meaningful problem-solving step in the domain, and what the goal and intent of the code is. Those with instructional expertise can answer questions about whether the current form of the plan is understandable and usable by students with a particular background. Human expertise is a valuable, and possibly indispensable, part of the plan identification process.

## 5.3 LLMs May Provide Opportunities to Support Programming Plan Identification

By providing a more precise understanding of how plan identification is currently performed by educators, as well as the barriers in the current process of plan identification, our interviews highlighted opportunities to leverage generative AI tools like ChatGPT for improving educators' process for plan identification.

A key challenge mentioned by our participants was how plan identification requires a lengthy and tedious process where the instructor needs to get familiar with many example programs for the domain they are working in. The time-consuming nature of creating a representative, general-purpose programming plan from many examples slows the process of plan identification for educators. In order to support the challenging process of identifying common examples, a plan identification system can *generate candidate plans* by searching relevant pools of code for common pieces of code. Large language models, which are trained on large corpora of code and include data from many example programs, have been shown to perform reasonably well on code generation and interpretation tasks [15], [44]. Thus, LLMs may be able to provide a starting point for instructors by presenting them with potential plan candidates.

Moreover, LLMs may be able to allow instructors to identify plans in unfamiliar domains by presenting them with plan candidates from those domains. A system that acts as a collaborator for the instructor might expedite the domain-specific plan identification process significantly. Through such a system, an instructor can obtain initial plan candidates in any domain and refine those plans such that they are appropriate for learners. This approach could make it possible to greatly expand the number of domains where the plans can be used for instruction, as it would no longer require a single person to have both instruction expertise and in-depth domain knowledge.

## Chapter 6

# LLM Plan Generation Methodology

To enact the opportunities described in Section 5.3, we explored the extent to which large language models can contribute to the plan identification process. To this end, we developed an automated ChatGPT pipeline (see Figure 6.1) to generate *plan-ful examples*, which we define as examples of programming plans in use, with all plan components identified (see Section 4.1). Specifically, we focused on *web scraping with BeautifulSoup* as our domain, since prior work has identified it as an area of interest for novice programming learners [7]. Later, we also generated plans in the domains of data processing and analysis using Pandas, visualization via Matplotlib, and machine learning using TensorFlow. Although in this section we focus on describing the methodology for plans in BeautifulSoup, we use the same pipeline for plan generation in the other aforementioned domains.

### 6.1 Generating In-Domain Programs

Our participants reviewed example programs (Section 4.3.1) and conducted literature reviews (Section 4.3.3) as key parts of their plan identification process. Inspired by this, we used Open AI's GPT-4, a state-of-the-art large language model for code generation that is trained on a large corpus of computer programs [59], to generate candidate programs along with its respective plan components in the programs. First, we prompted the model to generate 100 use cases of using BeautifulSoup. Subsequently, we asked the model to write pieces of code that use BeautifulSoup to achieve <use case>. This collection of

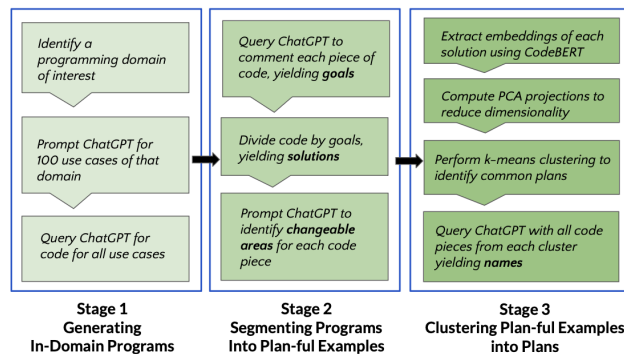


Figure 6.1: The three stage process for generating example programs, segmenting them with plan components, and clustering these plan-ful examples.

100 example programs (which we refer to as  $\mathcal{D}$ ) was used as our primary dataset for further analysis.

## 6.2 Segmenting Programs Into Plan-ful Examples

We used the generated programs from  $\mathcal{D}$  as the input in a set of prompts (see Stage 2 in Figure 6.1), where each prompt was used to generate one of the plan components identified in our interview study (Section 4.1).

### 6.2.1 Extracting Goals and Solutions

Originally, GPT-Generated programs in  $\mathcal{D}$  typically included a comment before each line, which described that line’s functionality. However, these comments did not capture the high-level purpose of the code, as required by a plan goal. To generate more abstract goals for a piece of code, we defined subgoals as **short descriptions of small pieces of code that do something meaningful** in a prompt and asked the LLM to **highlight subgoals as comments in the code**. The output from this prompt was a modified version of each program from  $\mathcal{D}$ , where blocks of code are preceded by a comment describing the goal of that block.

We split each complete program into multiple segments based on these new comments. Thus, the subgoal comments from each complete program in the modified  $\mathcal{D}$  became a plan goal, and the code following that comment became the associated solution. Each goal and solution pair was added as a single unit of data in our plan-ful example dataset,  $\mathcal{D}^{Plan-ful}$ .

### 6.2.2 Extracting Changeable Areas

To annotate the changeable areas for a plan, we defined changeable areas as **parts of the plan that would change when it is used in a different context** in our prompt and asked the model to **return the exact part of the code from the line that would change** for all code pieces from  $\mathcal{D}^{Plan-ful}$ . This data was added back to  $\mathcal{D}^{Plan-ful}$ .

## 6.3 Clustering Plan-ful Examples into Plans

We used a clustering algorithm to group similar plan-ful examples together as a candidate programming plan. For clustering the code pieces, we used the CodeBERT model from Microsoft [60] to obtain embeddings for each code piece in  $\mathcal{D}^{Plan-ful}$  and applied Principal Component Analysis (PCA) [61] to reduce the dimensionality of the embedding vectors while preserving 90% of the variance. These embeddings were clustered using the K-means algorithm [62]. The optimal number of clusters  $\mathcal{K}$  ( $\mathcal{K} = 210$  for BeautifulSoup) was determined by assessing all possible  $\mathcal{K}$  values using the mean silhouette coefficient [63]. We assigned each example in  $\mathcal{D}^{Plan-ful}$  to a cluster of similar code pieces.

### 6.3.1 Extracting Names

To generate names for the plan-ful examples, we first defined the properties for a name in the prompt by expressing that a **name reflects the code’s purpose** and it should focus **what the code is achieving and not the context**. Then, all code snippets from each cluster of examples were provided as input to the LLM along with a prompt asking it to **devise a name for that cluster of plans**.



## Chapter 7

# LLM Plan Generation Evaluation

To evaluate the plan-ful examples in BeautifulSoup created by generative AI tools, we performed a mixed methods evaluation guided by the characteristics instructors use to judge good programming plans (see Section 4.2).

We assessed our programming plans in reference to a set of programming plans identified and used by Cunningham et al. [7] to teach web scraping to undergraduate conversational programmers. These plans were designed by researchers with programming plan expertise as well as instructional experience in the domain, and were also validated with web scraping experts [7]. To obtain a control set, we extracted the same set of plan components (name, goal, solution, and changeable areas) from the publicly available curriculum<sup>1</sup> and created clusters of plan-ful examples (denoted as  $\mathcal{D}^{Control}$ ).

We also subsampled the generated plan-ful examples in  $\mathcal{D}^{Plan-ful}$  to have an equivalent number of examples as the control set. To achieve this, we chose the 10 largest clusters with the most data points and calculated the centroid of each cluster using the embeddings. We then selected the four closest plans to this centroid as the most representative examples in each cluster (compiled together as  $\mathcal{D}^{Plan-ful*}$ ).

However, as discussed in a prior section 2.1, very few researchers have identified plans in application-focused domains. This gap leads to a lack of the availability of control datasets for Matplotlib, Pandas, and Tensorflow. Thus, we perform a cross-domain analysis of the quantitative results of each of these domains, keeping the BeautifulSoup outcomes as a baseline.

We also crawled Stackoverflow to scrape code pieces in all domains: BeautifulSoup, Matplotlib, Pandas, and Tensorflow. Stackoverflow is a popular interaction platform for developers that provides its users with a space to ask questions about code and receive expert responses [64]. We believed that oftentimes, one specific query is representative of a code piece achieving a particular goal. Consequently, this led us to believe that scraping Stackoverflow would yield code pieces that could be considered similar to programming plans. We then performed a domain-wise comparative analysis between the GPT-generated plans and the code pieces from Stackoverflow for all domains. Nonetheless, both these approaches are extensions designed to overcome the shortcomings of the absence of control datasets; its results are not definitive and can only be used as inferential evidence.

---

<sup>1</sup>[runestone.academy/ns/books/published/PurposeFirstWebScraping/index.html](https://runestone.academy/ns/books/published/PurposeFirstWebScraping/index.html)

## 7.1 Quantitative Analysis

### 7.1.1 Syntactic Validity

Before comparing  $\mathcal{D}^{Plan-ful*}$  to  $\mathcal{D}^{Control}$ , we tested the syntactic validity of the generated programs from our original dataset  $\mathcal{D}$ . We note that from our set of 100 complete Python programs in BeautifulSoup, all but one were syntactically valid. That program included a syntax error and could not be parsed nor executed. In the 100 Python programs from Tensorflow, two files contained syntax errors leading to compilation failures. In the set of 100 Python programs for Matplotlib and Pandas all files were syntactically valid. Thus, we conclude that the raw code generated by the LLM is mostly accurate and reliable, at least in our target domains.

### 7.1.2 Appropriateness for Learners

Our instructors emphasized the importance of plans being suitable for their learner audience (Section 4.2). Thus, we compared  $\mathcal{D}^{Plan-ful}$  to  $\mathcal{D}^{Control}$  with standard code complexity metrics to determine their suitability for novices: non-comment lines of code, cyclomatic complexity [65], Halstead volume [66], and cognitive complexity [67].

Table 7.1 shows the mean value for all metrics across the datasets. For all metrics, a lower value indicates a simpler program that is more appropriate for beginning learners. We also conducted a two-sided non-parametric Mann-Whitney U-test [68] for each complexity metric.

For BeautifulSoup, while  $\mathcal{D}^{Control}$  is marginally less complex compared to generated code according to the metrics, we did not find any statistically significant trends ( $p > 0.05$  for all comparisons). Thus, it is reasonable to claim that the examples generated using ChatGPT can be used in instruction for novices.

Due to the absence of a control dataset, it is difficult to consider the individual nuances of the other domains (Matplotlib, Pandas, and Tensorflow) while analyzing the metric results. However, we can draw a cross-domain comparison between the examples in  $\mathcal{D}^{Plan-ful}$  and the other domains as suggestive evidence. In comparison to BeautifulSoup, the plans for Matplotlib are shorter, the Pandas plans are equivalent and the Tensorflow plans are substantially longer. The plans in Matplotlib and Pandas are seemingly much less complex than BeautifulSoup (accounted by the large difference between their cognitive complexity and cyclomatic complexity scores) despite being more dense in terms of their Halstead volumes.

To bridge the gap created by the lack of control datasets for Matplotlib, Pandas, and Tensorflow, we also scraped data from Stackoverflow, referred to as  $\mathcal{D}^{Stack}$ , for all of the domains and analyzed them relative to  $\mathcal{D}^{Plan-ful*}$  (summarized in Table 7.1). As mentioned before, we believed that the code snippets on this platform would be similar to programming plans. On the contrary, we find that the code pieces on Stackoverflow are much longer accross all domains evident by the difference in the values of the mean lines of code. The values for the other complexity metrics are also substantially larger, however, that could be attributable to the code being longer. All p-values computed after the comparison ( $\mathcal{D}^{Stack}$  vs.  $\mathcal{D}^{Plan-ful*}$ ) of each metric in all domains were statistically significant ( $p > 0.05$ ) reinforcing these findings. In any case, we infer that using code directly from Stackoverflow would not be suitable for novices and the code will likely have to go through a refinement process to be crafted as a “plan”. Moreover, datasets directly sourced from Stackoverflow would not be considered adequate control datasets in light of the disconnect between the properties of programming plans and the characteristics of code on Stackoverflow.

Table 7.1: Mean Code Complexity Metrics. The - represents the absence of data for that column.

Domain	Metric	$\mathcal{D}^{Plan-ful}$	$\mathcal{D}^{Plan-ful*}$	$\mathcal{D}^{Control}$	$\mathcal{D}^{Stack}$
BeautifulSoup		(n = 781)	(n = 40)	(n = 43)	(n = 94)
	Lines of Code	2.30	3.10	2.72	14.20
	Cyclomatic Complexity	2.43	2.21	2.40	4.20
	Halstead Volume	173.69	178.91	114.02	742.13
	Cognitive Complexity	0.217	0.375	0.233	1.83
Matplotlib		(n = 668)	(n = 40)	-	(n = 422)
	Lines of Code	1.79	1.525	-	17.14
	Cyclomatic Complexity	2.09	2.03	-	3.28
	Halstead Volume	197.31	115.59	-	1638.37
	Cognitive Complexity	0.016	0.000	-	0.70
Pandas		(n = 680)	(n = 40)	-	(n = 198)
	Lines of Code	2.11	1.8	-	16.35
	Cyclomatic Complexity	2.07	1.96	-	2.82
	Halstead Volume	205.75	54.79	-	1371.24
	Cognitive Complexity	0.039	0.000	-	0.40
Tensorflow		(n = 854)	(n = 40)	-	(n = 184)
	Lines of Code	3.21	3.375	-	16.52
	Cyclomatic Complexity	2.07	1.96	-	3.10
	Halstead Volume	372.48	273.05	-	1496.80
	Cognitive Complexity	0.064	0.000	-	1.83

### 7.1.3 Usability

Aside from generating code that is accurate and appropriate for learners, it is also important that programming plans be representative of key functionalities in the domain. To this end, we compared the number of distinct method calls used in  $\mathcal{D}^{Plan-ful}$  and  $\mathcal{D}^{Control}$ . Having examples on more distinct methods may indicate a set of examples that can be employed to solve a larger number of problems.

$\mathcal{D}^{Control}$  included four distinct method calls (`append`, `find`, `find_all`, `get`), which were also included in  $\mathcal{D}^{Plan-ful*}$ , the plan-ful examples from the 10 largest clusters generated by the LLM, such as `select` and `select_one`. Moreover, these largest clusters also included five additional methods not included in  $\mathcal{D}^{Control}$ . This shows that our pipeline generates plans with similar functionality to those designed by an instructor.

Due to the absence of instructor generated plans for Pandas, Matplotlib, and Tensorflow, we were unable to quantitatively analyze the usability of GPT generated plans in these domains.

### 7.1.4 Commonality

A significant motive for using programming plans in instruction is to equip novices with the necessary technical skills to contribute to real-world code problems. Thus, it is essential that plans used in instruction are representative of actual practice. To obtain an estimate of how the LLM-generated plan-ful examples compare to actual practice, we compared  $\mathcal{D}^{Plan-ful}$  and  $\mathcal{D}^{Control}$  to web scraping files from GitHub. We created a new dataset  $\mathcal{D}^{GitHub}$  by collecting Python files from public repositories via GitHub’s API that met the following criteria: contained a BeautifulSoup import statement, included the BeautifulSoup constructor, and was not a test file. This resulted in the final dataset with 733 files. Then, we generated the embeddings for these programs using CodeBERT in a similar manner to Section 6.3 to compare the sets  $\mathcal{D}^{GitHub}$  and

Table 7.2: Scores for Hausdorff and Wasserstein Distances Between Various Datasets. The - represents the absence of data for that column.

Domain	Metric	$\mathcal{D}^{Github} \& \mathcal{D}^{Plan-ful*}$	$\mathcal{D}^{Github} \& \mathcal{D}^{Control}$	$\mathcal{D}^{Github} \& \mathcal{D}^{Stack}$
BeautifulSoup	Hausdorff	13.66	14.92	6.85
	Wasserstein	12.97	13.97	9.77
Matplotlib	Hausdorff	16.14	-	11.80
	Wasserstein	14.06	-	10.25
Pandas	Hausdorff	14.03	-	7.62
	Wasserstein	12.88	-	8.77
Tensorflow	Hausdorff	13.30	-	10.83
	Wasserstein	12.80	-	10.57

$\mathcal{D}^{Control}$  as well as  $\mathcal{D}^{GitHub}$  and  $\mathcal{D}^{Plan-ful*}$ .

To evaluate the similarity between sets, we computed Hausdorff distance [69] and Wasserstein distance [70], which are common metrics for comparing generated content to reference sets [71], [72]. For both Hausdorff ( $\mathcal{D}^{Plan-ful} = 13.66$ ,  $\mathcal{D}^{Control} = 14.92$ ) and Wasserstein ( $\mathcal{D}^{Plan-ful} = 12.97$ ,  $\mathcal{D}^{Control} = 13.97$ ) distances, the set of generated examples for BeautifulSoup had a smaller distance to code from GitHub in comparison to the control set of previously proposed plans, conveying that the ChatGPT can generate plan-ful code that is more representative of real-world examples compared to instructor code.

For the other domains, similar to the analysis with the complexity metrics, we perform a cross-domain analysis where we compare the results of the GPT-generated plans of Matplotlib, Pandas, and Tensorflow to the results of GPT-generated plans of BeautifulSoup to derive inferences about their commonality scores (summarized in Table 7.2). Pandas and Tensorflow have similar scores for both metrics, thus it is reasonable to claim that the GPT-generated plans are still representative of real-world practice. On the contrary, Matplotlib has a larger value for both distances so the GPT-generated plans could potentially have more variability.

Similar to evaluating usability, we also draw a comparison between GPT-generated plans and code collected from Stackoverflow for each of the domains. We find that the code on Stackoverflow is closer to the code on Github by a large difference; implying that the code on Stackoverflow is more representative of code used in practice. However, like usability, the difference in the number of lines of code in Stackoverflow and GPT-generated plans could be a confounding feature. A larger code piece would undoubtedly form a larger coverage of the code on Github; leading to a smaller Wasserstein or Hausdorff distance.

## 7.2 Qualitative Evaluation

To obtain a richer picture of the strengths and weaknesses of plan generation with LLMs, we conducted a qualitative evaluation of the generated plan-ful examples, inspired by thematic analysis approaches in prior work on code generation [73].

We started our analysis with a free-form discussion on both generated plans and previously proposed plans from  $\mathcal{D}^{Control}$  to familiarize ourselves with the data. One member of the research team prepared an initial codebook, with codes organized under two main dimensions reflecting the *components* and *characteristics* highlighted in Section 4. Two researchers coded a subset of examples (10% of the data) and obtained inter-rater reliability of 0.76 using percentage agreement [74]. The codebook was refined through discussion, and two researchers achieved an IRR of 0.89 after the second subset. One member of the team coded the rest

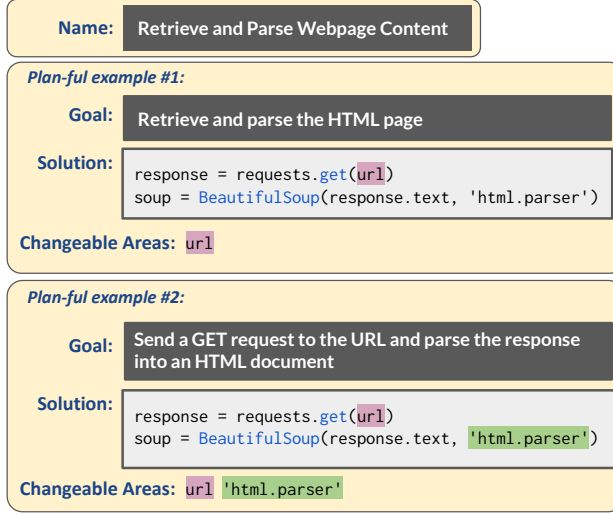


Figure 7.1: Two LLM-generated plan-ful examples from the same cluster, with an example almost identical to an instructor-generated plan from prior work (top), and an example that includes technical jargon and improbable changeable areas, making it potentially confusing for novices (bottom).

of the data according to the refined codebook<sup>2</sup>.

### 7.2.1 Components

The generated plan-ful examples were ‘mostly accurate’ (90%, n=36). Only four examples in  $\mathcal{D}^{Plan-ful*}$  had ‘mostly inaccurate’ code, indicating that LLMs can generate the solution component of a plan reliably.

Changeable areas of the examples were also somewhat successfully generated: there was only a single case where an unalterable part of the code was annotated as a changeable area. Yet, 22.5% of examples were missing changeable areas (n=9), and another 22.5% had changeable areas that were considered ‘improbable’ (n=9). For example, some default arguments of the commonly used functions were annotated as changeable. While technically correct, these areas are not likely to be modified in simpler examples and were not included in previously proposed plans from  $\mathcal{D}^{Control}$ .

The generation of goals and names was less satisfactory. On the example level, more than half of the generated goals were ‘descriptive’ (55%, n=22), but 17.5% of examples were missing a goal label (n=7), and 12.5% of examples had an ‘insufficient’ or ‘too general’ goal (n=5). On the cluster level, only 40% of generated names were ‘descriptive’ (n=4), with other names either being ‘insufficient’ to understand when to use a plan (n=2) or ‘overstating’ what the plan actually does (n=4). For example, a cluster that accesses multiple attributes of an object was named “Data Extraction and Database Management”, even though it does not have any database interaction.

### 7.2.2 Characteristics

The most consistent characteristic in generated examples was commonality: 80% of examples had ‘common syntax’ with plans placed in the same cluster (n=32) and 67.5% of them had ‘common goals’ with the plans in the cluster (n=27). Another 12.5% of examples shared ‘vague commonalities’ (n=5), where it was hard to find the overall goal of the cluster due to great differences in syntax and structure. Moreover, some code

<sup>2</sup>The codebook is available online: <https://tinyurl.com/flk6pzat8>

statements were repeated in multiple plans (30%, n=12), and the code for shorter plans, such as importing libraries or calling the BeautifulSoup constructor, was also included within some of the larger plans.

From a usability perspective, most plans were ‘cohesive’ examples of a given use case (67.5%, n=27), and they were ‘generalizable’ to new contexts (57.5%, n=23). Moreover, some of the shorter plans did not require customization but could still be useful to students, e.g. “Importing Libraries”.

Finally, the appropriateness of the generated content for beginners was questionable: while there were similarities to the ones defined in  $\mathcal{D}^{Control}$ , 42.5% of plans used ‘technical jargon’ in the name and goals (n=17). These included revealing some web technologies that were abstracted away in the previously proposed plans, such as GET requests and HTML structure, as seen in Figure 7.1. Furthermore, some plans included ‘advanced concepts’ in Python (15%, n=6) such as list comprehensions or exception handling.

## Chapter 8

# Discussion

In this section, we address RQ2: *How can LLMs (e.g. ChatGPT) support the identification of domain-specific plans?*

We found that our ChatGPT pipeline can reliably generate common domain-specific code. Our quantitative and qualitative evaluation showed that generated plan solutions were almost entirely syntactically valid (see Section 7.1.1 and Section 7.2.1). In addition, the generated code is representative of actual practice (Section 7.1.4), as shown by similarity to a reference set of plans validated by experts, and comparable similarity to Github files from the same domain (Section 4.2.1). Moreover, accounting from the similar number of distinct methods covered in the two datasets, we infer that ChatGPT can generate plans that capture a variety of use cases in the domain (Section 7.1.3). Furthermore, the complexity of the generated code appears to be similar to those generated by instructors with domain expertise, indicating that the generated examples can be appropriate for novices (Section 7.1.2). Overall, using LLMs for early phases in plan identification by generating common examples and recognizing candidates is a promising avenue.

However, our approach is not consistently able to describe the code appropriately for learners. It especially falls short on code interpretation, namely, generating other components of a plan such as names and goals. A number of its generated responses were either overly general or overstating what the code achieved (see Section 7.2.1). This might reflect the existing challenges for LLMs on in-context learning tasks, observed by prior work [75]. In addition, ChatGPT sometimes generated technical jargon in the names and goals (see 7.2.2), which makes those plan components unsatisfactory for novice learners. However, despite these pitfalls, the generated plan components were somewhat accurate, implying that they may be appropriate starting points for instructors to refine.

The results yielded by the comparative examination of the features of the GPT-generated plans with code scraped from Stackoverflow explained that the code collected by crawling StackOverflow is not suitable to create candidate programming plans or even control datasets for evaluating GPT-generated plans. We also investigated the success metrics of GPT-generated plans of Matplotlib, Pandas, and Tensorflow using a cross-domain analysis with BeautifulSoup’s GPT-generated plans. All other three domains (Pandas, Matplotlib, Tensorflow) shared similar numbers for commonality, usability, and appropriateness for learner goals metrics suggesting that the findings of this study are also applicable to domains other than BeautifulSoup.

Overall, we present suggestive evidence that using LLMs to generate candidate plans as a part of a plan generation pipeline could reduce the tedium in the identification process by eliminating the need for instructors to view programs or perform a literature review (Section 4.3.1 and Section 4.3.3) prior to creating

plans. Instead, LLMs could provide instructors with candidate plans that they would modify for their learner audience to ensure that the explanatory components are accurate and reasonable. A promising direction for the design of an automated plan identification system is to foster collaboration between an LLM and instructors in order to scale domain-specific plan identification.



## Chapter 9

# Implications for Future Work

To date, the field of designing domain-specific programming plans has been relatively restricted due to a number of concerns. Our findings in Section 4.4 suggest the following reasons: ambiguous definitions of programming plans and an unclear methodology for designing these plans. With more insight about the components of a programming plan and its success metrics, we provide instructors with a clear understanding of the specific constructs of a plan to support them in the identification of plans in more application-focused domains. As highlighted in Section 4.4.1, with the current process, it is not efficient to identify plans at scale. A substantial understanding of programming plans contributed by the results of our first study has significant design implications for the development of a socio-technical system that is capable of identifying domain-specific programming plans at scale.

In this paper, we focused on the design of an automated pipeline for the creation of programming plans. However, different parts of this pipeline including subgoal label generation, code commonality computation, and code abstraction evaluation independently have implications beyond just programming plan generation.

## 9.1 Components as Constructs

### 9.1.1 Subgoal Labels

The Subgoal Learning Model devised by Catrambone [76] established that including subgoal labels in code for instruction is conducive to novice learning by reducing their cognitive load. Yet, their studies and results were restricted to a mathematical context. Margulieux et al. [77] take a step further and study the effect of subgoal labels in computer science education specifically, highlighting that the subgoal label scaffolding indeed leads to improved learning outcomes. Morrison et al. [78] and Margulieux et al. [79] also demonstrate similar results in studies with different introductory computer science tasks. However, very few researchers discuss how to come up with these subgoal labels and how to evaluate their worth. Our model powered by LLMs provides a promising avenue for generating subgoal labels for code being used in instruction at scale. With the availability of such a system at hand, researchers could now explore the specific elements that should constitute a subgoal label and also their success metrics.

## 9.2 Evaluation Metrics

Our approach to computing the commonality and abstraction levels of code pieces takes a step forward in evaluating new properties of code having implications in a broader spectrum of instructional design practices.

### 9.2.1 Outcomes of Computing Commonality of Code

Based on our survey of the literature, the space for determining the commonality of code has been relatively unexplored. Existing techniques including code similarity software like MOSS [80] and JPlag [81] revolve around finding a combination of exact or slightly modified structural, semantic and syntactic similarities [82]. According to Lee et al. [83], these tools fall short on detecting behaviorally similar code. Our pipeline offers a straightforward approach for evaluating commonality of programming plans that can easily be adapted and generalized for other code purposes. It could be used as suggestive evidence for code that may not look alike but is similar in terms of its purpose. This finding has potential implications in the field of computer science education for improving instructors' ability to understand whether their instruction is authentic [84]. Namely, with the supported of our pipeline, the instructor-designed code examples could be tested for authenticity via comparison with code representative of real-world programming practice.

### 9.2.2 Outcomes of Computing Abstractness of Code

The concept of abstraction has evolved with different perspectives in the literature. At present, there have been studies examining the importance of abstraction in learning in computer science [85], [86] and how students perform on abstracted computer science tasks [87], [88], but there has been little work exploring a substantial computation of abstraction levels of code. Using the findings from our interview study, we established that other metrics of evaluation: commonality, usability, and appropriateness for learner goals share a relationship with the level of abstraction of code. Analogous to the results of behavioral science work that established abstract concepts required a greater reinforcement effort and yielded in more errors ([58], [89]), in Section 5.1, we theorized that it is more suitable to incorporate detailed plans for novice instruction. We also speculated that more abstract plans would be more commonly seen in practice but would require more effort from the user to adapt it to their needs. With our contribution of a more comprehensive understanding of the abstraction level of code and a design explaining the computation of similarity, usability, and suitability for novices to compute the abstraction score of code, we present a promising direction for future work on assessing the abstraction skills of students.

### 9.2.3 Need for Quantitative Measures Appropriateness for Novices

In our proposed pipeline, we implemented the use of existing metrics [65], [67] for evaluating code suitability for beginners. However, these metrics solely measure the code's properties; there is no quantitative measure of a plan's suitability for novices. Using open-book coding analysis for our qualitative work for this measure was suitable because of its smaller scale. To design an effective system that can support instructors refine plans, there is a need for a robust quantitative measure of suitability. A viable approach would involve implementing the framework of Cognitive Complexity of Computer Programs proposed by Duran et al. [90]. They move beyond software engineering metrics of measuring code's difficulty into exploring the cognitive complexity of given code. However, the proposal is theoretical and there is a need for the design of a system that would enable this theoretical proposition.

#### 9.2.4 Need for Alternative Measures for Usability

Our methodology for analyzing the usability of generated plans was completely dependent on our control dataset of prior identified plans in the domain. However, seeking control datasets for newer domains would be a considerable challenge keeping in mind the sparse identification of plans in application-focused domains. There is a need for finding alternative measures of usability of plans for designing a system that can successfully support instructors in plan refinement. One potential direction is exploring the complexity of a plan’s changeable areas. As mentioned in Section 4.1, changeable areas are the parts of the plan that would change when used in a different context. An abundance of changeable areas in a plan would be indicative of the large scope of the plan’s adaptability for different contexts. However, the presence of too many changeable areas would also make it difficult for novices to adapt the plan to their needs. Thus, there is a need of finding the right balance of the number of changeable areas in a plan for its ideal usability levels. Consequently, insight from the percentage of changeable areas of a plan would be suggestive of its usability scores. Similar to the proposal for appropriateness for novices, we note that our system only provides an initial step for understanding the usability of a plan. Before integrating the system-generated plans in instruction, there is need for future work testing the usability of the GPT-generated plans by giving them to students and assessing their performance on using the given plans.

## Chapter 10

# Conclusion

The current state of the art in programming plan identification is a black-boxed lengthy manual procedure. With a more transparent, efficient, technologically-supported plan identification process, educators may be able to use programming plans in their instruction for a wide range of subjects beyond introductory programming. In this study, we gathered concrete details about the current state of the art in plan identification processes to understand what plan components instructors seek, what metrics they use to judge success, and where they face challenges. Then, we explored the use of ChatGPT as an aid in the plan identification process and found that it could generate candidate programming plans with significant similarity to instructor-designed plans, however, many of the plans' explanatory components were not well attuned to the needs of beginning learners. We also establish a need for future studies to examine the generalizability of these results in domains other than BeautifulSoup using our proposed design of the computation of the success metrics. Nevertheless, these results suggest that the way forward in plan identification across domains should involve collaboration between LLMs and instructors.

As computing educators are tasked with teaching an increasingly diverse set of learners with widely varied interests, we believe that a plan identification system that utilizes large language models can support instructors develop instructional material that is not only based on the psychology of programming, but also more tailored to their students' interests in a diverse set of domains.

# References

- [1] E. Soloway and K. Ehrlich, “Empirical studies of programming knowledge,” *IEEE Transactions on Software Engineering*, vol. SE-10, pp. 595–609, 1984.
- [2] J. C. Spohrer, E. Soloway, and E. Pope, “A Goal/Plan Analysis of Buggy Pascal Programs,” *Human–Computer Interaction*, vol. 1, no. 2, pp. 163–207, Jun. 1985, Publisher: Taylor & Francis, ISSN: 0737-0024. DOI: [10.1207/s15327051hci0102\\_4](https://doi.org/10.1207/s15327051hci0102_4).
- [3] V. Iyer and C. Zilles, “Pattern census: A characterization of pattern usage in early programming courses,” in *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*, ser. SIGCSE ’21, Virtual Event, USA: Association for Computing Machinery, 2021, pp. 45–51, ISBN: 9781450380621. DOI: [10.1145/3408877.3432442](https://doi.org/10.1145/3408877.3432442). [Online]. Available: <https://doi.org/10.1145/3408877.3432442>.
- [4] O. Muller, D. Ginat, and B. Haberman, “Pattern-Oriented Instruction and Its Influence on Problem Decomposition and Solution Construction,” in *Proceedings of the 12th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, ser. ITiCSE ’07, event-place: Dundee, Scotland, New York, NY, USA: Association for Computing Machinery, 2007, pp. 151–155, ISBN: 978-1-59593-610-3. DOI: [10.1145/1268784.1268830](https://doi.org/10.1145/1268784.1268830). [Online]. Available: <https://doi.org/10.1145/1268784.1268830>.
- [5] M. Allamanis and C. Sutton, “Mining idioms from source code,” in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014, Hong Kong, China: Association for Computing Machinery, 2014, pp. 472–483, ISBN: 9781450330565. DOI: [10.1145/2635868.2635901](https://doi.org/10.1145/2635868.2635901). [Online]. Available: <https://doi.org/10.1145/2635868.2635901>.
- [6] A. V. Robins, “12 novice programmers and introductory programming,” *The Cambridge handbook of computing education research*, p. 327, 2019.
- [7] K. Cunningham, B. J. Ericson, R. Agrawal Bejarano, and M. Guzdial, “Avoiding the turing tarpit: Learning conversational programming by starting from code’s purpose,” in *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, ser. CHI ’21, Yokohama, Japan: Association for Computing Machinery, 2021, ISBN: 9781450380966. DOI: [10.1145/3411764.3445571](https://doi.org/10.1145/3411764.3445571). [Online]. Available: <https://doi.org/10.1145/3411764.3445571>.
- [8] N. Weinman, A. Fox, and M. A. Hearst, “Improving instruction of programming patterns with faded parsons problems,” in *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, ser. CHI ’21, `|conf-loc|`, `|city|Yokohama|/city|`, `|country|Japan|/country|`, `|/conf-loc|`: Association for Computing Machinery, 2021, ISBN: 9781450380966. DOI: [10.1145/3411764.3445228](https://doi.org/10.1145/3411764.3445228). [Online]. Available: <https://doi.org/10.1145/3411764.3445228>.

- [9] J. C. Spohrer, E. Soloway, and E. Pope, “A Goal/Plan analysis of buggy pascal programs,” *Human-Computer Interaction*, vol. 1, no. 2, pp. 163–207, Jun. 1985, ISSN: 0737-0024. DOI: [10.1207/s15327051hci0102\\\_4](https://doi.org/10.1207/s15327051hci0102\_4). [Online]. Available: [https://doi.org/10.1207/s15327051hci0102%5C\\_4](https://doi.org/10.1207/s15327051hci0102%5C_4).
- [10] R. S. Rist, “Schema creation in programming,” *Cognitive science*, vol. 13, no. 3, pp. 389–414, 1989, ISSN: 0364-0213, 1551-6709. DOI: [10.1207/s15516709cog1303\\\_3](https://doi.org/10.1207/s15516709cog1303\_3).
- [11] C. Izu, V. Lonati, A. Morpurgo, and M. Sanchez, “An Inventory of Goals from CS1 Programs Processing a Data Series,” in *2021 IEEE Frontiers in Education Conference (FIE)*, Place: Lincoln, NE, USA, IEEE Press, 2021, pp. 1–8. DOI: [10.1109/FIE49875.2021.9637360](https://doi.org/10.1109/FIE49875.2021.9637360). [Online]. Available: <https://doi.org/10.1109/FIE49875.2021.9637360>.
- [12] N. Lojo and A. Fox, “Teaching Test-Writing As a Variably-Scaffolded Programming Pattern,” in *Proceedings of the 27th ACM Conference on on Innovation and Technology in Computer Science Education Vol. 1*, ser. ITiCSE ’22, event-place: Dublin, Ireland, New York, NY, USA: Association for Computing Machinery, 2022, pp. 498–504, ISBN: 978-1-4503-9201-3. DOI: [10.1145/3502718.3524789](https://doi.org/10.1145/3502718.3524789). [Online]. Available: <https://doi.org/10.1145/3502718.3524789>.
- [13] A. Sivaraman, R. Abreu, A. Scott, T. Akomolede, and S. Chandra, “Mining idioms in the wild,” in *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*, ser. ICSE-SEIP ’22, Pittsburgh, Pennsylvania: Association for Computing Machinery, 2022, pp. 187–196, ISBN: 9781450392266. DOI: [10.1145/3510457.3513046](https://doi.org/10.1145/3510457.3513046). [Online]. Available: <https://doi.org/10.1145/3510457.3513046>.
- [14] T. Karanikiotis and A. L. Symeonidis, “Towards extracting reusable and maintainable code snippets,” in *International Conference on Software Technologies*, Springer, 2022, pp. 187–206.
- [15] B. Jury, A. Lorusso, J. Leinonen, P. Denny, and A. Luxton-Reilly, “Evaluating LLM-generated Worked Examples in an Introductory Programming Course,” in *Proceedings of the 26th Australasian Computing Education Conference*, Sydney NSW Australia: ACM, Jan. 2024, pp. 77–86, ISBN: 9798400716195. DOI: [10.1145/3636243.3636252](https://doi.org/10.1145/3636243.3636252). (visited on 04/17/2024).
- [16] E. M. Soloway and B. Woolf, “Problems, plans, and programs,” in *ACM SIGCSE Bulletin*, ACM, vol. 12, 1980, pp. 16–24.
- [17] E. Soloway, K. Ehrlich, J. Bonar, and J. Greenspan, “What do novices know about programming?” In *Directions in Human-Computer Interaction*, A. Badre and B. Schneiderman, Eds., Ablex Publishing, 1982, pp. 87–122.
- [18] E. Soloway, “From problems to programs via plans: The content and structure of knowledge for introductory lisp programming,” *Journal of Educational Computing Research*, vol. 1, no. 2, pp. 157–172, 1985.
- [19] E. Soloway, “Learning to program= learning to construct mechanisms and explanations,” *Communications of the ACM*, vol. 29, no. 9, pp. 850–858, 1986.
- [20] E. Soloway, J. Bonar, and K. Ehrlich, “Cognitive strategies and looping constructs: An empirical study,” *Communications of the ACM*, vol. 26, no. 11, pp. 853–860, 1983.
- [21] J. G. Spohrer and E. Soloway, “Analyzing the high frequency bugs in novice programs,” in *Empirical Studies of Programmers Workshop*, E. Soloway and S. Iyengar, Eds., Ablex, 1986, pp. 230–251.
- [22] J. C. Spohrer and E. Soloway, “Putting it all together is hard for novice programmers,” in *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics*, vol. March, IEEE, 1985.

- [23] D. Wood, J. S. Bruner, and G. Ross, “The role of tutoring in problem solving,” *Journal of child psychology and psychiatry*, vol. 17, no. 2, pp. 89–100, 1976.
- [24] M. Guzdial, M. Konneman, C. Walton, L. Hohmann, and E. Soloway, “Layering scaffolding and CAD on an integrated workbench: An effective design approach for project-based learning support,” *Interactive Learning Environments*, vol. 6, no. 1/2, pp. 143–179, 1998.
- [25] L. Hohmann, M. Guzdial, and E. Soloway, “Soda: A computer-aided design environment for the doing and learning of software design,” in *International Conference on Computer Assisted Learning*, Springer, 1992, pp. 307–319.
- [26] M. J. Clancy, *Designing Pascal Solutions: Case studies using data structures*. WH Freeman & Co., 1996.
- [27] O. Muller and B. Haberman, “Supporting abstraction processes in problem solving through pattern-oriented instruction,” *Computer Science Education*, vol. 18, no. 3, pp. 187–212, 2008, Publisher: Routledge .eprint: <https://doi.org/10.1080/08993400802332548>. DOI: [10.1080/08993400802332548](https://doi.org/10.1080/08993400802332548). [Online]. Available: <https://doi.org/10.1080/08993400802332548>.
- [28] R. Duran, J. Sorva, and S. Leite, “Towards an analysis of program complexity from a cognitive perspective,” in *Proceedings of the 2018 ACM Conference on International Computing Education Research*, ser. ICER ’18, Espoo, Finland: Association for Computing Machinery, 2018, pp. 21–30, ISBN: 9781450356282. DOI: [10.1145/3230977.3230986](https://doi.org/10.1145/3230977.3230986). [Online]. Available: <https://doi.org/10.1145/3230977.3230986>.
- [29] E. Wallingford, “Elementary patterns and their role in instruction,” in *OOPSLA ’98*, 1998.
- [30] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen, “Boa: A language and infrastructure for analyzing ultra-large-scale software repositories,” in *2013 35th International Conference on Software Engineering (ICSE)*, 2013, pp. 422–431. DOI: [10.1109/ICSE.2013.6606588](https://doi.org/10.1109/ICSE.2013.6606588).
- [31] M. Guzdial, “Creating an on-ramp to programming for arts and humanities students with teaspoon languages and custom block languages,” in *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 2*, ser. SIGCSE 2024, {conf-loc}\_, {city}\_Portland\_/city\_, {state}\_OR\_/state\_, {country}\_USA\_/country\_, {/conf-loc}\_: Association for Computing Machinery, 2024, p. 1898, ISBN: 9798400704246. DOI: [10.1145/3626253.3633414](https://doi.org/10.1145/3626253.3633414). [Online]. Available: <https://doi.org/10.1145/3626253.3633414>.
- [32] M. Mernik, J. Heering, and A. M. Sloane, “When and how to develop domain-specific languages,” *ACM Comput. Surv.*, vol. 37, no. 4, pp. 316–344, Dec. 2005, ISSN: 0360-0300. DOI: [10.1145/1118890.1118892](https://doi.org/10.1145/1118890.1118892). [Online]. Available: <https://doi.org/10.1145/1118890.1118892>.
- [33] A. Kurihara, A. Sasaki, K. Wakita, and H. Hosobe, “A programming environment for visual block-based domain-specific languages,” *Procedia Computer Science*, vol. 62, pp. 287–296, 2015.
- [34] D. Weintrop, “Block-based programming in computer science education,” *Communications of the ACM*, vol. 62, no. 8, pp. 22–25, 2019.
- [35] J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond, “The scratch programming language and environment,” *ACM Transactions on Computing Education (TOCE)*, vol. 10, no. 4, pp. 1–15, 2010.
- [36] D. Weintrop and U. Wilensky, “Comparing block-based and text-based programming in high school computer science classrooms,” *ACM Trans. Comput. Educ.*, vol. 18, no. 1, Oct. 2017. DOI: [10.1145/3089799](https://doi.org/10.1145/3089799). [Online]. Available: <https://doi.org/10.1145/3089799>.

- [37] P. K. Chilana, C. Alcock, S. Dembla, *et al.*, “Perceptions of non-CS majors in intro programming: The rise of the conversational programmer,” in *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, Atlanta, GA: IEEE, Oct. 2015, pp. 251–259, ISBN: 978-1-4673-7457-6. DOI: [10.1109/VLHCC.2015.7357224](https://doi.org/10.1109/VLHCC.2015.7357224). [Online]. Available: <http://ieeexplore.ieee.org/document/7357224/> (visited on 07/15/2021).
- [38] P. K. Chilana, R. Singh, and P. J. Guo, “Understanding Conversational Programmers: A Perspective from the Software Industry,” in *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, New York, NY, USA: Association for Computing Machinery, May 2016, pp. 1462–1472, ISBN: 978-1-4503-3362-7. [Online]. Available: <https://doi.org/10.1145/2858036.2858323> (visited on 07/15/2021).
- [39] R. Shin, M. Allamanis, M. Brockschmidt, and O. Polozov, “Program synthesis and semantic parsing with learned code idioms,” *arXiv preprint arXiv:1906.10816*, 2019.
- [40] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, “Code2vec: Learning distributed representations of code,” *Proc. ACM Program. Lang.*, vol. 3, no. POPL, 40:1–40:29, Jan. 2019, ISSN: 2475-1421. DOI: [10.1145/3290353](https://doi.org/10.1145/3290353). [Online]. Available: <http://doi.acm.org/10.1145/3290353>.
- [41] E. L. Glassman, J. Scott, R. Singh, P. J. Guo, and R. C. Miller, “Overcode: Visualizing variation in student solutions to programming problems at scale,” *ACM Transactions on Computer-Human Interaction (TOCHI)*, vol. 22, no. 2, p. 7, 2015.
- [42] T. Zhang, G. Upadhyaya, A. Reinhardt, H. Rajan, and M. Kim, “Are code examples on an online q&a forum reliable? a study of api misuse on stack overflow,” in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE ’18, Gothenburg, Sweden: Association for Computing Machinery, 2018, pp. 886–896, ISBN: 9781450356381. DOI: [10.1145/3180155.3180260](https://doi.org/10.1145/3180155.3180260). [Online]. Available: <https://doi.org/10.1145/3180155.3180260>.
- [43] E. L. Glassman, T. Zhang, B. Hartmann, and M. Kim, “Visualizing api usage examples at scale,” in *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, ACM, 2018, p. 580.
- [44] J. Finnie-Ansley, P. Denny, B. A. Becker, A. Luxton-Reilly, and J. Prather, “The Robots Are Coming: Exploring the Implications of OpenAI Codex on Introductory Programming,” in *Proceedings of the 24th Australasian Computing Education Conference*, ser. ACE ’22, New York, NY, USA: Association for Computing Machinery, Feb. 2022, pp. 10–19, ISBN: 978-1-4503-9643-1. DOI: [10.1145/3511861.3511863](https://doi.org/10.1145/3511861.3511863). (visited on 04/16/2024).
- [45] J. Finnie-Ansley, P. Denny, A. Luxton-Reilly, E. A. Santos, J. Prather, and B. A. Becker, “My AI Wants to Know if This Will Be on the Exam: Testing OpenAI’s Codex on CS2 Programming Exercises,” in *Proceedings of the 25th Australasian Computing Education Conference*, ser. ACE ’23, New York, NY, USA: Association for Computing Machinery, Jan. 2023, pp. 97–104, ISBN: 978-1-4503-9941-8. DOI: [10.1145/3576123.3576134](https://doi.org/10.1145/3576123.3576134). (visited on 04/16/2024).
- [46] T. Wang, D. V. Díaz, C. Brown, and Y. Chen, “Exploring the Role of AI Assistants in Computer Science Education: Methods, Implications, and Instructor Perspectives,” in *2023 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, Washington, DC, USA: IEEE, Oct. 2023, pp. 92–102, ISBN: 9798350329469. DOI: [10.1109/VL-HCC57772.2023.00018](https://doi.org/10.1109/VL-HCC57772.2023.00018). (visited on 04/17/2024).



- [47] S. Sarsa, P. Denny, A. Hellas, and J. Leinonen, “Automatic Generation of Programming Exercises and Code Explanations Using Large Language Models,” in *Proceedings of the 2022 ACM Conference on International Computing Education Research - Volume 1*, ser. ICER ’22, vol. 1, New York, NY, USA: Association for Computing Machinery, Aug. 2022, pp. 27–43, ISBN: 978-1-4503-9194-8. DOI: [10.1145/3501385.3543957](https://doi.org/10.1145/3501385.3543957). (visited on 04/16/2024).
- [48] P. Denny, J. Leinonen, J. Prather, *et al.*, “Prompt Problems: A New Programming Exercise for the Generative AI Era,” in *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1*, Portland OR USA: ACM, Mar. 2024, pp. 296–302, ISBN: 9798400704239. DOI: [10.1145/3626252.3630909](https://doi.org/10.1145/3626252.3630909). (visited on 04/17/2024).
- [49] P. Denny, D. H. Smith IV, M. Fowler, J. Prather, B. A. Becker, and J. Leinonen, *Explaining Code with a Purpose: An Integrated Approach for Developing Code Comprehension and Prompting Skills*, Mar. 2024. arXiv: [2403.06050 \[cs\]](https://arxiv.org/abs/2403.06050). (visited on 04/17/2024).
- [50] J. Leinonen, P. Denny, S. MacNeil, *et al.*, “Comparing Code Explanations Created by Students and Large Language Models,” in *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1*, Turku Finland: ACM, Jun. 2023, pp. 124–130, ISBN: 9798400701382. DOI: [10.1145/3587102.3588785](https://doi.org/10.1145/3587102.3588785). (visited on 04/17/2024).
- [51] B. P. Cipriano and P. Alves, “GPT-3 vs Object Oriented Programming Assignments: An Experience Report,” in *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1*, ser. ITiCSE 2023, New York, NY, USA: Association for Computing Machinery, Jun. 2023, pp. 61–67, ISBN: 9798400701382. DOI: [10.1145/3587102.3588814](https://doi.org/10.1145/3587102.3588814). (visited on 04/16/2024).
- [52] M. Fowler, B. Chen, and C. Zilles, “How should we ‘explain in plain english’? voices from the community,” in *Proceedings of the 17th ACM Conference on International Computing Education Research*, ser. ICER 2021, Virtual Event, USA: Association for Computing Machinery, 2021, pp. 69–80, ISBN: 9781450383264. DOI: [10.1145/3446871.3469738](https://doi.org/10.1145/3446871.3469738). [Online]. Available: <https://doi.org/10.1145/3446871.3469738>.
- [53] V. Clarke and V. Braun, “Thematic analysis: A practical guide,” *Thematic Analysis*, pp. 1–100, 2021.
- [54] D. A. Norman and S. W. Draper, *User Centered System Design; New Perspectives on Human-Computer Interaction*. USA: L. Erlbaum Associates Inc., 1986, ISBN: 0898597811.
- [55] J. Sorva, “Notional Machines and Introductory Programming Education,” *Trans. Comput. Educ.*, vol. 13, no. 2, 8:1–8:31, Jul. 2013, Place: New York, NY, USA Publisher: ACM, ISSN: 1946-6226. DOI: [10.1145/2483710.2483713](https://doi.org/10.1145/2483710.2483713). [Online]. Available: <http://doi.acm.org/10.1145/2483710.2483713>.
- [56] M. Arab, T. D. LaToza, J. Liang, and A. J. Ko, “An exploratory study of sharing strategic programming knowledge,” in *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*, ser. CHI ’22, New Orleans, LA, USA: Association for Computing Machinery, 2022, ISBN: 9781450391573. DOI: [10.1145/3491102.3502070](https://doi.org/10.1145/3491102.3502070). [Online]. Available: <https://doi.org/10.1145/3491102.3502070>.
- [57] J. R. Anderson, L. M. Reder, and H. A. Simon, “Situated learning and education,” *Educational Researcher*, vol. 25, no. 4, pp. 5–11, 1996. DOI: [10.3102/0013189X025004005](https://doi.org/10.3102/0013189X025004005). eprint: <https://doi.org/10.3102/0013189X025004005>. [Online]. Available: <https://doi.org/10.3102/0013189X025004005>.
- [58] J. A. Kaminski, V. M. Sloutsky, and A. F. Heckler, “Do children need concrete instantiations to learn an abstract concept,” in *Proceedings of the XXVIII annual conference of the cognitive science society*, Erlbaum Mahwah, NJ, 2006, pp. 1167–1172.

- [59] J. Liu, C. S. Xia, Y. Wang, and L. ZHANG, “Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation,” in *Advances in Neural Information Processing Systems*, A. Oh, T. Neumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, Eds., vol. 36, Curran Associates, Inc., 2023, pp. 21 558–21 572.
- [60] Z. Feng, D. Guo, D. Tang, *et al.*, “Codebert: A pre-trained model for programming and natural languages,” *arXiv preprint arXiv:2002.08155*, 2020.
- [61] I. T. Jolliffe *et al.*, “Principal component analysis and factor analysis,” *Principal component analysis*, vol. 372, pp. 115–128, 1986.
- [62] J. A. Hartigan and M. A. Wong, “Algorithm as 136: A k-means clustering algorithm,” *Journal of the royal statistical society. series c (applied statistics)*, vol. 28, no. 1, pp. 100–108, 1979.
- [63] R. Lletí, M. C. Ortiz, L. A. Sarabia, and M. S. Sánchez, “Selecting variables for k-means cluster analysis by using a genetic algorithm that optimises the silhouettes,” *Analytica Chimica Acta*, vol. 515, no. 1, pp. 87–100, 2004.
- [64] T. Ahmed and A. Srivastava, “Understanding and evaluating the behavior of technical users. a study of developer interaction at stackoverflow,” *Human-centric Computing and Information Sciences*, vol. 7, pp. 1–18, 2017.
- [65] T. McCabe, “A complexity measure,” *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308–320, 1976. DOI: [10.1109/TSE.1976.233837](https://doi.org/10.1109/TSE.1976.233837).
- [66] M. H. Halstead, *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc., 1977.
- [67] G. A. Campbell, “Cognitive complexity: An overview and evaluation,” in *Proceedings of the 2018 International Conference on Technical Debt*, ser. TechDebt ’18, Gothenburg, Sweden: Association for Computing Machinery, 2018, pp. 57–58, ISBN: 9781450357135. DOI: [10.1145/3194164.3194186](https://doi.org/10.1145/3194164.3194186). [Online]. Available: <https://doi.org/10.1145/3194164.3194186>.
- [68] T. W. MacFarland, J. M. Yates, T. W. MacFarland, and J. M. Yates, “Mann–whitney u test,” *Introduction to nonparametric statistics for the biological sciences using R*, pp. 103–132, 2016.
- [69] A. A. Taha and A. Hanbury, “An efficient algorithm for calculating the exact Hausdorff distance,” *IEEE transactions on pattern analysis and machine intelligence*, vol. 37, no. 11, pp. 2153–2163, Nov. 2015, ISSN: 1939-3539. DOI: [10.1109/TPAMI.2015.2408351](https://doi.org/10.1109/TPAMI.2015.2408351).
- [70] A. Ramdas, N. Garcia, and M. Cuturi, *On Wasserstein Two Sample Testing and Related Families of Nonparametric Tests*, Oct. 2015. DOI: [10.48550/arXiv.1509.02237](https://doi.org/10.48550/arXiv.1509.02237). arXiv: [1509.02237 \[math, stat\]](https://arxiv.org/abs/1509.02237). (visited on 04/28/2024).
- [71] M. Arjovsky, S. Chintala, and L. Bottou, “Wasserstein generative adversarial networks,” in *Proceedings of the 34th International Conference on Machine Learning*, D. Precup and Y. W. Teh, Eds., ser. Proceedings of Machine Learning Research, vol. 70, PMLR, Jun. 2017, pp. 214–223. [Online]. Available: <https://proceedings.mlr.press/v70/arjovsky17a.html>.
- [72] W. Li, Z. Liang, P. Ma, R. Wang, X. Cui, and P. Chen, “Hausdorff gan: Improving gan generation quality with hausdorff metric,” *IEEE Transactions on Cybernetics*, vol. 52, no. 10, pp. 10 407–10 419, 2022. DOI: [10.1109/TCYB.2021.3062396](https://doi.org/10.1109/TCYB.2021.3062396).

- [73] M. Kazemitabaar, X. Hou, A. Henley, B. J. Ericson, D. Weintrop, and T. Grossman, “How Novices Use LLM-based Code Generators to Solve CS1 Coding Tasks in a Self-Paced Learning Environment,” in *Proceedings of the 23rd Koli Calling International Conference on Computing Education Research*, Koli Finland: ACM, Nov. 2023, pp. 1–12, ISBN: 9798400716539. DOI: [10.1145/3631802.3631806](https://doi.org/10.1145/3631802.3631806). (visited on 04/29/2024).
- [74] M. B. Miles and A. M. Huberman, *Qualitative data analysis: An expanded sourcebook*. sage, 1994.
- [75] T. Li, G. Zhang, Q. D. Do, X. Yue, and W. Chen, *Long-context llms struggle with long in-context learning*, 2024. arXiv: [2404.02060](https://arxiv.org/abs/2404.02060) [cs.CL].
- [76] R. Catrambone, “The subgoal learning model: Creating better examples so that students can solve novel problems.,” *Journal of experimental psychology: General*, vol. 127, no. 4, p. 355, 1998.
- [77] L. E. Margulieux, M. Guzdial, and R. Catrambone, “Subgoal-labeled instructional material improves performance and transfer in learning to develop mobile applications,” in *Proceedings of the Ninth Annual International Conference on International Computing Education Research*, ser. ICER ’12, Auckland, New Zealand: Association for Computing Machinery, 2012, pp. 71–78, ISBN: 9781450316040. DOI: [10.1145/2361276.2361291](https://doi.org/10.1145/2361276.2361291). [Online]. Available: <https://doi.org/10.1145/2361276.2361291>.
- [78] B. B. Morrison, L. E. Margulieux, B. Ericson, and M. Guzdial, “Subgoals help students solve parsons problems,” in *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, ser. SIGCSE ’16, Memphis, Tennessee, USA: Association for Computing Machinery, 2016, pp. 42–47, ISBN: 9781450336857. DOI: [10.1145/2839509.2844617](https://doi.org/10.1145/2839509.2844617). [Online]. Available: <https://doi.org/10.1145/2839509.2844617>.
- [79] L. E. Margulieux, B. B. Morrison, B. Franke, and H. Ramilison, “Effect of implementing subgoals in code.org’s intro to programming unit in computer science principles,” *ACM Trans. Comput. Educ.*, vol. 20, no. 4, Oct. 2020. DOI: [10.1145/3415594](https://doi.org/10.1145/3415594). [Online]. Available: <https://doi.org/10.1145/3415594>.
- [80] A. Aiken, *Moss (measure of software similarity)*, <https://theory.stanford.edu/~aiken/moss/>, Accessed: 2024-05-22.
- [81] L. Prechelt, G. Malpohl, and M. Philippsen, “Jplag: Finding plagiarisms among a set of programs,” 2000.
- [82] C. Liu, C. Chen, J. Han, and P. S. Yu, “Gplag: Detection of software plagiarism by program dependence graph analysis,” in *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2006, pp. 872–881.
- [83] G. Lee, J. Kim, M.-s. Choi, R.-Y. Jang, and R. Lee, “Review of code similarity and plagiarism detection research studies,” *Applied Sciences*, vol. 13, no. 20, 2023, ISSN: 2076-3417. DOI: [10.3390/app132011358](https://doi.org/10.3390/app132011358). [Online]. Available: <https://www.mdpi.com/2076-3417/13/20/11358>.
- [84] N. Brown and G. Wilson, “Ten quick tips for teaching programming,” *PLoS Computational Biology*, vol. 14, no. 4, e1006023, 2018. DOI: [10.1371/journal.pcbi.1006023](https://doi.org/10.1371/journal.pcbi.1006023). [Online]. Available: <https://doi.org/10.1371/journal.pcbi.1006023>.
- [85] J. S. Bruner, “The process of education, harvard, univ,” *Press, Cambridge, Mass*, 1960.
- [86] J. M. Wing, “Computational thinking,” *Communications of the ACM*, vol. 49, no. 3, pp. 33–35, 2006.

- [87] D. Statter and M. Armoni, “Teaching abstraction in computer science to 7th grade students,” *ACM Trans. Comput. Educ.*, vol. 20, no. 1, Jan. 2020. DOI: [10.1145/3372143](https://doi.org/10.1145/3372143). [Online]. Available: <https://doi.org/10.1145/3372143>.
- [88] D. Statter and M. Armoni, “Learning abstraction in computer science: A gender perspective,” in *Proceedings of the 12th Workshop on Primary and Secondary Computing Education*, ser. WiPSCE ’17, Nijmegen, Netherlands: Association for Computing Machinery, 2017, pp. 5–14, ISBN: 9781450354288. DOI: [10.1145/3137065.3137081](https://doi.org/10.1145/3137065.3137081). [Online]. Available: <https://doi.org/10.1145/3137065.3137081>.
- [89] H. B. Reed and R. D. Dick, “The learning and generalization of abstract and concrete concepts,” *Journal of Verbal Learning and Verbal Behavior*, vol. 7, no. 2, pp. 486–490, 1968, ISSN: 0022-5371. DOI: [https://doi.org/10.1016/S0022-5371\(68\)80037-4](https://doi.org/10.1016/S0022-5371(68)80037-4). [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0022537168800374>.
- [90] R. Duran, J. Sorva, and S. Leite, “Towards an analysis of program complexity from a cognitive perspective,” in *Proceedings of the 2018 ACM conference on international computing education research*, 2018, pp. 21–30.