Interruptions and Recovery: Leveraging Dynamic Code History in Development

Vo Thien Tri Pham
Dept. of Computer Science and Engineering
Washington University in St. Louis
Saint Louis, MO, USA
p.tri@wustl.edu

Haixin Zhou
Dept. of Computer Science and Engineering
Washington University in St. Louis
Saint Louis, MO, USA
haixin@wustl.edu

Caitlin Kelleher
Dept. of Computer Science and Engineering
Washington University in St. Louis
Saint Louis, MO, USA
ckelleher@wustl.edu

***Abstract*—Programmer interruptions are common and disruptive. While research has focused on understanding their nature, impact, and associated recovery cost, others have looked at visual reminders of recent development actions as recovery support mechanisms. Until now, however, these have only extended as far as the timeline of source code change alone. Thus, there are opportunities to improve this information in a codebase's history to better support programmers' interruption recovery. We conducted a within-subjects exploratory study investigating how developers use code history during interruption recovery. We compared a new sub-goal history interface, which contextually organized code changes and web activity into sub-goals, with an existing chronological timeline tool support identified in the literature. Our findings reveal that developers significantly leveraged the sub-goal history interface more frequently during post-interruption recovery, suggesting its potential to improve how developers navigate past work. While not leading to a significant reduction in overall recovery time in this study, sub-goal history provided valuable support for integrating external resources, facilitated faster initial re-engagement, and showed a generally better recovery success rate. We observed developers employing diverse, multi-modal strategies, and, importantly, our results highlight recovery as a multi-stage process rather than a single event. These insights emphasize the crucial role of history-aware tool support for recovery workflows and underscore the need for designs that balance contextual detail with usability and adaptability to effectively support both developers' passive (reading through information) and active (searching for information) engagement with the recovery tool support.**

***Index Terms*—Task interruption, Code history, Software development**

I. Introduction

Interruptions are a significant obstacle for software developers. In a typical day, developers frequently switch between updating code, attending meetings, assisting colleagues, and handling unexpected issues [1]–[3]. These activity switches result in disruptive fragmented work sessions [1], [4], [5] and limit opportunities to extend concentration on a single task.

Interruptions, whether planned or unplanned, can challenge developer productivity [1], [2], [6], [7] by requiring rapid shifts in mental focus. Frequent context switches require a considerable amount of mental work, which can quickly drain a programmer's mental resources and hinder effective task execution [8], [9]. Studies show that switching between tasks increases cognitive load and decreases overall productivity [6]–[8], [10], [11].

Given both the prevalence and negative impacts of interruption on developers, quickly and effectively resuming coding after an interruption is an important challenge. Recovering from an interruption requires developers to quickly and easily rebuild their mental context in order to continue working efficiently. Yet, previous research [2], [6], [12] suggests that programmers spend considerable time reorienting themselves after interruptions, emphasizing the need for improved strategies and tools to support interruption recovery.

Recent work suggests leveraging the codebase's history may help to support programmers in recovering from an interruption [13]–[15]. To date, this work has focused primarily on the source code alone, through exploring *cues*, or visual reminders of immediate developer context displayed within the integrated development environment (IDE). Parnin and DeLine [14] found that programmers perceived a timeline of recent code changes as being helpful in resuming development following interruptions, though there was no significant time advantage. Kersten and Murphy [13] found saving and restoring task contexts, plus filtering out irrelevant parts of the codebase, increased programmers' edits relative to navigation activities. Trafton et al. [15] found that using visual markers to indicate the programmer's last IDE action helped them resume work more quickly.

While leveraging code history suggests promise in helping programmers recover from interruptions, the improvements are modest thus far and only explore a limited space of interventions. We note that many programmers develop code in an opportunistic style that blends information searches with coding [16]–[19]. Furthermore, their queries and searches contain some information about their sub-goals as they work to build complex functionality [20]–[22]. This paper extends the existing work in interruption support through exploring a sub-goal based history that organizes code changes and web activity into sub-goals, annotated with a brief description. We conducted a within-subjects exploratory study in which programmers had access to two different types of interruption support: 1) a sub-goal history and 2) a timeline of recent code edits, modeled after the work of Parnin and DeLine [14]. We pose the following questions:

RQ1: What strategies do programmers use to try to rebuild context following an interruption?

RQ2: How does the task that programmers are performing when interrupted relate to the challenge of resuming work?

RQ3: How does historical information (i.e., sub-goal history and code timeline) support the process of resuming work?

Our results suggest developers employ a range of strategies, including code navigation, comprehension, revisiting web-based information, and program re-execution to recover context after interruptions. Recovery is not a single event, but a multi-stage process. It begins with disorientation and poses questions that are answered through navigation, code comprehension, and code editing before reaching a stage where productive work is possible. The difficulty of the recovery process varies based in part on the context in which the interruption occurred. Our results suggest that contexts in which there are multiple avenues for making additional progress are easier to recover from than those with a single logical next step. Developers were significantly more likely to recover from tasks that included multiple kinds of work (algorithmic, spatial, and user interface based) than single-type tasks. Developers engaged with the history interfaces both passively, through reading information presented in the default view, and actively, by using affordances to navigate or reveal needed information. Overall, participants used the sub-goal history presentation significantly more than the code timeline, suggesting that richer history information can be valuable in supporting recovery. However, their behavior and processes suggested there are additional opportunities to support interruption recovery in code.

II. Related Work

Understanding how task interruptions affect software development is crucial for improving developer productivity and code quality. This section reviews the literature on task interruption in software development, its impact on programmers, and approaches for aiding task resumption.

A. *Occurrences of Interruptions in Software Development*

In software development, task interruptions occur frequently. Typically, any event requiring developers to change focus before continuing their previous activity is considered an interruption [1], [2], [5]. Major sources include external factors from human-initiated disruptions (e.g., colleague questions, meetings) [1], [4], [23], [24] to non-human interruptions (e.g., notifications from communication tools, IDE alerts) [4], [25]–[27]. Developers also face self-initiated interruptions when they switch tasks on their own [4], [23], [27], [28]. Abad et al. [8] found that internal interruptions can happen when developers are stuck on a task (due to tool problems or technical issues). Developers may also get sidetracked by remembering other tasks [8], which is common during context switching between different projects or tasks [9]. Having a clear understanding of the nature and triggers of these interruptions is important for developing effective support for interruption recovery.

Interruptions can vary in frequency and duration. Research suggests that programmers are frequently interrupted and program only in short bursts [1], [3], [4], [27]. This phenomenon of work fragmentation is also supported by studies in software evolution [5] and software engineering [2], [9]. For example, Sanchez et al. [5] found developers routinely switch tasks, with continuous work sessions averaging around twelve minutes. Parnin and Rugaber [2] discovered developers spend fifteen to thirty minutes re-establishing context after interruptions. Similarly, Vasilescu et al. [9] found developers working on multiple GitHub projects simultaneously face more frequent daily context switches, creating further fragmented work in the process.

Taken together, these studies suggest that interruptions are common and hinder development progress, highlighting the need to improve recovery assistance. Our work builds on this by investigating a new type of support via sub-goal history. We emphasize the importance of efficient support systems to minimize interruption effects.

B. *Impact of Interruption on Programmers*

Though often seen as undesirable, interruptions are not always bad. In programming, they can offer benefits, such as helping find overlooked issues and providing mental breaks [28]–[30]. However, they can also raise error rates, increase task completion time, and break workflow [6]–[8], [11]. This section explores their positive and negative effects on programmers.

Work interruptions in software development can yield several beneficial outcomes, such as valuable team communication and collaboration. For example, routine review meetings in planning, specification, and release phases are productive, allowing developers to understand project scope and get helpful peer comments [29], [30]. Research [31], [32] also found team environment interruptions tend to be more relevant and brief, offering valuable input that increases productivity and reduces disruption. Such interruptions encourage instantaneous feedback and cooperation, which is particularly relevant as a fourth of developers' time in a workday is spent on collaborative activities [1]. These interactions are facilitated by instant communication and quick access to information, which are key positive outcomes of these interruptions [25], [32]–[34]. Furthermore, internal interruptions, like breaks, can help programmers reduce stress, refresh their minds, and improve concentration upon returning to the primary task [28].

On the other hand, research shows interruptions can have obstructive outcomes like delayed performance and increased error rates [6]–[8], [10], [11]. Bailey and Konstan [6] found that users take three to twenty-seven percent longer to finish the primary tasks when interrupted by peripheral tasks, making twice as many errors when interruptions occur mid-task compared to at task boundaries. Interruptions during high cognitive load tasks can overwhelm mental resources and impact productivity [7], [35], [36]. Leroy [37] showed that people often struggle to stop thinking about an unfinished task after switching. This "attention residue" reduces resources available for the latter task, thus compromising its performance [37]. Interruptions can also impact emotional state, causing unpleasant feelings like stress or displeasure [7], [28], [38], [39].

Whether positive or negative, interruptions require time to resume the suspended activity. Supporting programmers to effectively resume tasks can minimize the impact of unavoidable interruptions and make beneficial ones less disruptive. Leveraging tools like code history can help achieve this by reducing negative outcomes like emotional frustration and lost productivity associated with context switching. This ability to mitigate the costs of switching can transform potentially disruptive interruptions into beneficial problem-solving opportunities.

C. *Support for Interruption Recovery*

Research has explored support for task resumption after interruption both in general work settings and in software development. Outside of software engineering, interruptibility research offers strategies designed to avoid high interruption and resumption costs. Czerwinski et al. [27] conducted a diary study on task switching and interruptions, showing the usefulness of external memory aids like note-taking. Adamczyk and Bailey [36] explored how interruptions at different moments within a task sequence affect performance and how visual cues and reminders can help users reorient during structured interruptions and resumptions. Hu and Lee [40], [41] looked into using time-lapse recordings of screen states to help users organize and retrieve their digital artifacts, which can be useful in task resumption involving different knowledge spaces. Bailey and Iqbal [42] found that interruptions should be planned to coincide with periods of reduced mental workload to minimize disruption. Mark et al. [7] examined the impact of task switches, finding that completing interrupted tasks faster often increased stress and frustration. Kaur et al. [43] modeled opportune moments for work transitions and breaks to optimize happiness and productivity, showing that well-timed recommendations can improve overall work satisfaction. Other studies [6], [44] constructed statistical models to predict and manage interruptibility, providing a quantitative basis for effective interruption management system design.

In software development, research often focuses on maintaining code comprehension, preserving code context, and using tools (in-/outside IDEs) for quick resumption after interruptions. Parnin and DeLine [14] highlighted the importance of IDE cues, finding that a chronological list of development activities helps developers re-establish context. This timeline shows a list of the developer's recent actions (code selections, code changes, saves, and builds) in order of occurrence [14]. Using this timeline, developers were twice as likely to finish assigned tasks than with notes only, and they had no difficulty relocating relevant parts of the code or misunderstanding the program or changes made, although no significant time advantage was found [14]. Parnin [2] further explored resumption strategies, identifying re-reading code and using development history for re-gaining context. Similarly, Ko et al. [19] explored how programmers find and connect important information during software maintenance tasks. Here programmers use different tools like file tabs, bookmarks, the Windows taskbar, paper notes, etc., to keep track of relevant information [19]. Kersten and Murphy [13] created the Mylar plugin (now Mylyn), which uses task context to improve productivity by focusing the workspace on relevant tasks. Their research showed that keeping a relevant task context in IDEs can support task resumption.

Prior research on task resumption explores various strategies, particularly using cues and optimizing interruption timing. However, since programmers often cannot control when interruptions occur, timing-based approaches address only part of the challenge. Moreover, research on supporting task re-engagement itself remains limited. Our work extends the understanding of software development task resumption by investigating alternative ways historical code data can be presented to support recovery after disruptions. While previous studies primarily focused on visual aids or general task management tools, we introduce an integrated approach leveraging automated version control history, linked web resources, and natural language summarization, aiming to support programmers with quick, contextual interruption recovery.

III. Code History Interface Design & Implementation

We developed our sub-goal history interface as a companion view running on a second monitor alongside the primary development environment. This dual-monitor setup aimed to help developers maintain contextual awareness without disrupting their workflow.

In the sections below, we describe a practical usage scenario demonstrating how developers interact with the sub-goal history interface to recover context after interruptions. We then share our design principles and how they are reflected within the user interface (UI). Finally, we provide an overview of the implementation of our sub-goal history.

A. *Usage Scenario*

Sam, a junior software developer, has just ended an unplanned fifteen-minute call with a coworker about an important production problem related to their login service. Before pausing her work, Sam was trying to improve the way enemies spawn in a Python game she is developing.

When Sam returns back to her desk, she is still thinking about the authentication problem she discussed with her colleague. She looks at her code editor and asks herself, "What was I even working on before that meeting?". In the absence of interruption support, Sam would need to look through her code to determine where she was working and the current state of her code. This can be a time-consuming and cognitively demanding task, potentially requiring both code comprehension and visits to web resources.

Instead, Sam opens the sub-goal history. She reviews the sub-goals in the Recent Development Highlights section (Fig. 1B). She thinks, "That's right," as she scans her progress, "I added the fly obstacle first, then worked on collision detection, and was just starting to implement the random enemy spawning when I got interrupted."

Sam notices the bookmark icon marked with "4" next to her latest changes. "I remember I was searching something about spawn rates," she thinks. When she expands her most recent highlight, she sees both her code changes (Fig. 2C) and the corresponding Helpful Resources section (Fig. 2D). "Oh right!" she realizes as she scans the saved searches for "pygame midbottom" and the Stack Overflow threads on balancing spawn rate. "I wasn't just going to increase the timer – I was planning to implement a dynamic spawn rate that scales with the player's score." Without this record of her research efforts, Sam would have to rely on her memory alone to reconstruct these contexts, potentially missing important details.

Sam clicks on the line number to navigate directly to that line in the editor. "Now I can set up that dynamic spawn rate based on the player's score," she thinks, ready to pick up right where she left off before the interruption.

B. *Design Principles and Interface Elements*

In designing the sub-goal history interface, we prioritized helping developers quickly gain context after interruptions. Our design addresses three key needs through the corresponding principles:

1) *Provide a high-level overview of development history.*

We chose to organize the sub-goal history hierarchically, allowing developers to first scan high-level summaries of the sub-goals before accessing specific details about the activities within each one. Previous research [45] suggests that chronological sub-goals that link to code can support programmers in making lightweight evaluations of program function. A hierarchical, sub-goal history also aligns with Kruger's suggestion [46] that abstraction enhances the reuse of code. At the top level, programmers can view natural language summaries of recent code activities to remind themselves of their problem-solving process. These are automatically generated, but they can be edited if programmers wish to refine them. When relevant, programmers can expand the summaries to reveal both code diffs and the web resources they consulted while working on those changes. The code diffs can be displayed either side-by-side or inline, depending on the programmers' preference. We found through formative testing that some participants had strong preferences for one or the other. This sometimes varied by task.

2) *A sub-goal should serve as an entry point to related information.*

An entry point, as defined by Kirsh [47], is a "structure or cue in the environment that represent an invitation to do something or enter somewhere." Previous work [45] has shown that the presence of sub-goals helped programmers locate entry points to related information, enabling them to directly navigate to the relevant code sections as needed. In our interface, the expanded view of each sub-goal includes both the code diffs and the consulted resources. The code diffs allow programmers to jump directly from a specific change to the corresponding section of the code, therefore directly linking the sub-goal within the development process to the relevant code context. Similarly, the consulted resources enable quick, single-click review and revisit of the material used by programmers while working towards a given goal (Fig. 2D).

3) *Show in-progress work in greater detail.*

In displaying the sub-goal history, we distinguish between completed sub-goals, from which the developer has moved on, and in-progress sub-goals that are the focus of the developer's current work. For in-progress work, our system provides a detailed, step-by-step view of unfinished changes (Fig. 3B). These are presented one change at a time for two reasons: 1) we cannot yet determine the programmer's sub-goal, and 2) prior work has demonstrated that a series of recent diffs can help programmers to re-engage in a task after a short interruption [14]. Both completed and in-progress work are presented in the same sub-goal history view, with a draggable separator allowing the developer to devote more screen real estate to the information source more relevant to their current needs (Fig. 3C).

C. *Technical Implementation*

Our technical implementation focuses on three core activities intended to preserve the context of software development:

1) Recording programmers' behavior, including both code changes and web resource access
2) Segmenting the recorded behavior into meaningful development sub-goals
3) Creating natural language descriptions of these sub-goals for easy comprehension

1) *Recording Developer Behavior*

Our implementation extends work by [48], using VS Code and Chrome extensions to record development activities. Rather than using traditional timing-based approaches like auto-saves or manual commits, we integrate with developers' natural testing workflows. When developers execute their code, the extension captures the current state of their code changes and active browser tabs. Each capture contains any code modifications and references to consulted web resources. This test-based recording approach provides a natural granularity for capturing development context, because test executions typically align with discrete development tasks and meaningful implementation goals. Developers test their code when they have a cohesive version that could benefit from the feedback of testing.

2) *Segmenting Development Activities*

As developers work on programming tasks, the system continually updates the sub-goal history automatically. This requires that the system 1) keeps a record of the accumulated changes and 2) identifies sub-goals as they emerge.

Each time the developer tests their code, the system captures a new code state. Testing represents a natural cognitive boundary in development work; developers typically run their code after completing a logical implementation step. The code states are stored using a runtime dictionary, organized by the file path for each code edit. This dictionary of changes allows us to:

1) Simultaneously track changes across multiple files
2) Preserve the temporal markers to associate related activities
3) Integrate code modifications and their related web resources
4) Maintain the starting and current code states to generate diffs

Then, after each code test, the system considers the accumulated code changes to identify new potential sub-goals. This process is inspired by prior work [45], [49] about sub-goal labeling and edit grouping. Our system identifies significant changes using the following two criteria, each motivated by observed developer behavioral patterns:

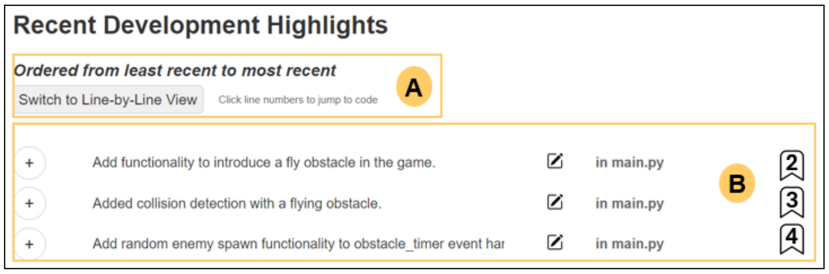
- **Identify new sub-goals through substantive changes**: The system forms new sub-goal groups (which can contain multiple execution captures) when modifications in the current execution exceed three non-empty lines. We chose this threshold based on earlier formative testing. We found that new sub-goals typically begin with the addition of several new lines of code. Then, these are tested and refined with smaller edits until the programmer is satisfied and begins another new sub-goal.
- **File-switching detection**: Additionally, the system creates new sub-goal groups when the developer begins working in a file without prior edits. While we expected to need a more robust, multi-file grouping system, in practice we found that sub-goal changes aligned well with edits in a new location.

When new sub-goals are identified using these criteria, the system then removes the code edits and web activity associated with the sub-goal from In-Progress Work and creates a new sub-goal entry in the Recent Development Highlights section. We note that our sub-goal criteria are very simple. In practice, they aligned well with the activities of participants in our study. However, this may be due in part to the programming style of the student developers who participated in our study and the modest project sizes. Future work in this area should explore a broader range of participants and development contexts to further refine the segmentation approach.

3) *Describing Development Changes*

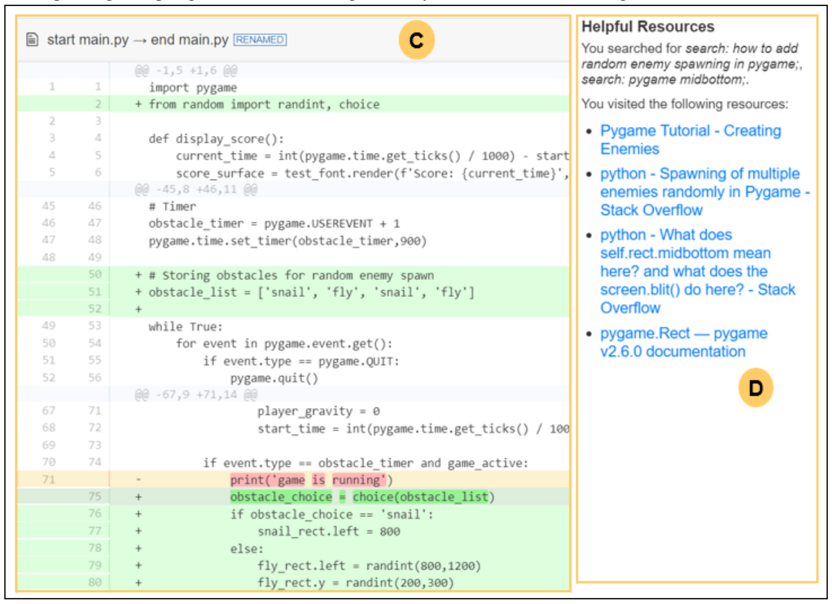
For each new sub-goal, the system generates a description of the code changes within this sub-goal using OpenAI's GPT API[1] with a structured prompt designed for concise, action-oriented summaries. We assign our system a role with the following statement: "You are a code change history summarizer that helps programmers who got interrupted from coding, and the programmers you are helping require simple and precise points that they can glance over and understand your point." When a sub-goal is formable, the system has an activity object containing information about the filename and corresponding starting and ending code states, which it then passes as parameters to our prompt (Fig. 4). The returned summary is then assigned back to the activity object's title attribute, which is later used to display the development highlight's description in the sub-goal history view.

[1]https://platform.openai.com/docs/overview


Recent Development Highlights
Ordered from least recent to most recent
Switch to Line-by-Line View
Click line numbers to jump to code
A
B
2
3
4

```
+   Add functionality to introduce a fly obstacle in the game.                 in main.py
+   Added collision detection with a flying obstacle.                          in main.py
+   Add random enemy spawn functionality to obstacle_timer event har           in main.py
```

Fig. 1. Key components of sub-goal history. A) Recent Development Highlights header with view toggle options and navigation hints; B) a development highlight with interactive elements: (+ / -) for expanding/collapsing detail, for editing summary, and a bookmark showing related resources.



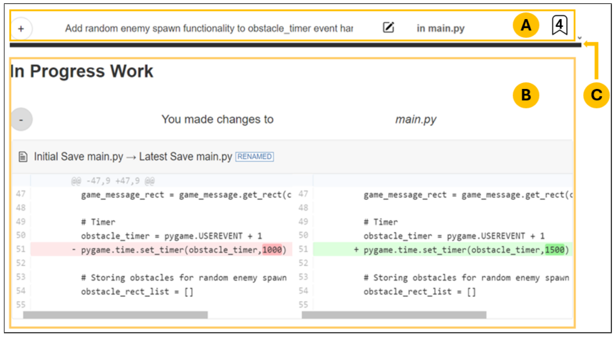
```
@@ -1,5 +1,6 @@
  import pygame
+ from random import randint, choice

  def display_score():
      current_time = int(pygame.time.get_ticks() / 1000) - start
      score_surface = test_font.render(f'Score: {current_time}',
@@ -45,8 +46,11 @@
  # Timer
  obstacle_timer = pygame.USEREVENT + 1
  pygame.time.set_timer(obstacle_timer,900)

+ # Storing obstacles for random enemy spawn
+ obstacle_list = ['snail', 'fly', 'snail', 'fly']
+
  while True:
      for event in pygame.event.get():
          if event.type == pygame.QUIT:
              pygame.quit()
@@ -67,9 +71,14 @@
                  player_gravity = 0
                  start_time = int(pygame.time.get_ticks() / 100

          if event.type == obstacle_timer and game_active:
-             print('game is running')
+             obstacle_choice = choice(obstacle_list)
+             if obstacle_choice == 'snail':
+                 snail_rect.left = 800
+             else:
+                 fly_rect.left = randint(800,1200)
+                 fly_rect.y = randint(200,300)
```

You searched for *search: how to add random enemy spawning in pygame;, search: pygame midbottom;.*

You visited the following resources:

- Pygame Tutorial - Creating Enemies
- python - Spawning of multiple enemies randomly in Pygame - Stack Overflow
- python - What does self.rect.midbottom mean here? and what does the screen.blit() do here? - Stack Overflow
- pygame.Rect — pygame v2.6.0 documentation

Fig. 2. Key components of sub-goal history. C) expanded diff view of code changes when implementing random enemy spawning; D) Helpful Resources section listing relevant documentation and discussions referenced during development.



```
+   Add random enemy spawn functionality to obstacle_timer event har      in main.py
```

```
@@ -47,9 +47,9 @@
    game_message_rect = game_message.get_rect(c      game_message_rect = game_message.get_rect(c

    # Timer                                          # Timer
    obstacle_timer = pygame.USEREVENT + 1            obstacle_timer = pygame.USEREVENT + 1
  - pygame.time.set_timer(obstacle_timer,1000)     + pygame.time.set_timer(obstacle_timer,1500)

    # Storing obstacles for random enemy spawn       # Storing obstacles for random enemy spawn
    obstacle_rect_list = []                          obstacle_rect_list = []
```

Fig. 3. Remaining components of sub-goal history. A) collapsed view of a summary; B) side-by-side diff view in the In Progress Work section; C) adjustable view handler for managing section sizes

Through iterative refining of our prompts, we found that:

1) Descriptions that start with verbs (e.g.,"Add functionality to introduce a fly obstacle") are easier to scan than those starting with nouns.
2) Avoiding explicit labeling of additions and deletions makes summaries feel more natural and less mechanical.
3) Limiting the token length to 25 ensures glanceable descriptions that developers can easily scan through.
4) Framing the task in terms of helping interrupted developers produces more relevant descriptions that focus on changes to functionality rather than changes to syntax.

IV. Methodology

To investigate how different code history interfaces impact programmers' ability to resume work after interruptions, we conducted a within-subjects exploratory study.

```
Compare the following code snippets of the file "[filename]":
Code A (before): "[before_code]"
Code B (after): "[after_code]"
Identify whether the changes are addition, deletion, or modification
without explicitly stating them. Also do not explicitly mention
Code A or Code B.
Summarize the changes in a single, simple, easy-to-read line.
No listing or bullet points. Start with a verb and no need
to end with a period. Make sure it sounds like a natural conversation.
```

Fig. 4. Prompt template for code change summarization

A. *Study Procedure*

Our study directly compared interfaces using a within-subjects design. Each participant used two interfaces (content-timeline and sub-goal history) and completed tasks in two codebases (Doodle Jump and Tile Matching Puzzle). We paired tasks with interfaces and selected the ordering of the interfaces based on a 2x2 Latin Square (task x interface) in order to mitigate ordering effects and learning advantages.

The study had six parts: 1) a demographic survey; 2) familiarization with one interface condition; 3) a 45-min programming task with interruptions using the first interface and task; 4) familiarization with the remaining interface condition; 5) a second 45-min programming task with interruptions using the second interface and task; and 6) a semi-structured interview.

1) *Interface Conditions*

Participants worked with two different interface conditions:

Content Timeline: Adapted from the work of Parnin and DeLine [14], the content timeline showed a chronological list of the programmer's activities. We included this interface as it represents an existing, successful approach for leveraging development information history after interruptions. The list included every code selection, edit, save, and build, representing an activity log without semantic grouping or contextual integration.

Sub-goal History: The sub-goal history, detailed in Section III, implemented our principles for effective code history presentation. Different from the content timeline, it organized information into Recent Development Highlights and In Progress Work sections, integrating code changes, descriptive summaries, and related web resources.

2) *Interruption Procedures*

We designed our interruptions to mimic real workplace disruptions that force developers to completely shift their focus away from coding. We considered two key design decisions: when to interrupt participants and what task to give them.

1) **Interruption Timing:** We elected to use random timing, interrupting between eleven and sixteen minutes into each programming task. This timing is taken from existing research in knowledge work that found uninterrupted working sessions of approximately this duration [3]. We interrupted each participant two times per task, with at least five minutes between interruptions to allow participants to become re-engaged in their work.
2) **Interruption Task:** Between programming periods, participants spent five minutes answering multiple-choice code comprehension questions. Pilot tests showed these were more disruptive to short-term memory than the kinds of mathematical problems that have been used in other interruption work [35], [44], [50], [51]. An example code comprehension question might ask participants to determine a function's return value when calculating the perimeter of a shape in a two-dimensional grid. To ensure engagement, participants received a twenty-cent bonus per correct answer, as performance incentives help clear attention residue between tasks [52].

B. *Materials*

We used three codebases for the in-person user study sessions: one to introduce participants to the two history interfaces and the other two as contexts for experimental tasks. Their attributes and related tasks are described below.

1) *Familiarization Codebase*

The familiarization codebase consists of a simple runner game in Python built with Pygame.[2] Its architecture maintains a clear separation between the game mechanics and the display elements. This was intended to allow participants to make targeted changes without needing extensive programming knowledge.

The familiarization codebase served two main goals: 1) ensuring participants could navigate and change code in a simpler setting before the main experiment, and 2) introducing both interface conditions. We created two familiarization tasks to achieve this, one for each interface. In the content-timeline view, participants modified parameter values and visualized discrete changes to parameter values. In the sub-goal history, participants built a basic scoring system, leveraging the interface to see the connections between code elements.

2) *Experimental Codebases*

Our study sought to examine how developers handle interruptions during programming that requires understanding and modifying interconnected code. We needed codebases featuring:

1) Several independent but connected systems
2) Clear component boundaries with well-defined interactions
3) Tasks requiring changes across system boundaries
4) Opportunities to build mental models during development

We developed two complementary codebases, each with three enhancement tasks designed to create natural opportunities for context management throughout development.

Doodle Jump Game: Our first codebase is a two-dimensional platformer inspired by Doodle Jump, built in Python using Pygame (Fig. 5A). The game features a character "Doodler," who jumps between platforms to reach maximum height. The implementation includes several interlinked systems: physics (movement and collisions), platform generation (managing the environment), and scoring (tracking player progress). This architecture creates natural boundaries for understanding and modifying code. For instance, the platform system handles both static and moving platforms (requiring careful state management), and the scoring system communicates with both physics and platform systems to track the player's highest point. Participants tackled three tasks:

- **Score Tracking**: Adding height-based scoring, requiring coordination between physics and UI systems
- **Game Over Screen**: Building an end-game screen when the player falls, involving state management and UI integration
- **Moving Platforms**: Creating horizontally moving platforms, requiring changes to platform generation and collision detection

Tile Matching Puzzle Game: Our second codebase is an animal-themed tile-matching game using HTML, CSS, and JavaScript (Fig. 5B). Players match tiles by creating paths of no more than three straight lines while avoiding obstacles. The implementation presents typical software engineering challenges in pathfinding, state tracking, and validation. The path-checking system simultaneously tracks chosen tiles, connection points, and obstacle collisions, mirroring real-world scenarios where multiple interacting states must be managed. Participants worked on three tasks:

- **Match Counting & Score Display**: Implementing score tracking across asynchronous events and UI changes
- **Hint System**: Creating a pathfinding system to highlight possible matches following the game's three-line rule
- **Countdown Timer with Game Over**: Implementing a timer with UI updates and game state transitions

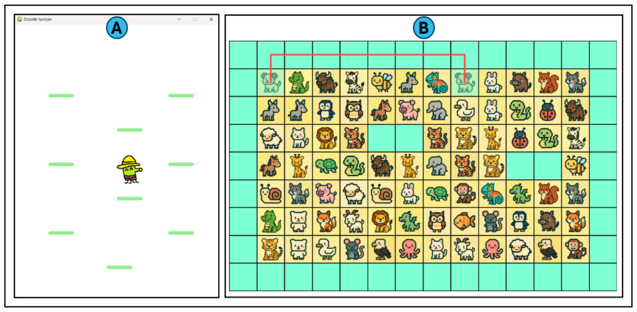


Fig. 5. A) Screenshot from Doodle Jump game. The player controls Doodler to jump between platforms while avoiding falling; B) Screenshot from Tile Matching Puzzle game. Players match tiles by drawing paths between them, with no more than three straight lines and no interference from other tiles. Animals' arts were generated using Microsoft Copilot, with a general prompt of "cute [animal] in pixel art/Microsoft Paint style." Conversations and arts are publicly available online.[3]

These codebases complement each other in examining programmer behavior: Doodle Jump focuses on continuous systems interaction in a traditional environment, while the tile-matching game presents complex validation logic and asynchronous operations in web development. Together, they represent varied technical challenges that developers face in production. Involving interconnected components, these tasks naturally encouraged opportunistic programming [16]–[19]. Developers built understanding across multiple program elements, tested frequently, and revised approaches as they discovered new constraints during implementation.

C. *Participants*

We recruited study participants through our university's Computer Science and Engineering department email list. All participants had at least three months of work experience in software engineering or four or more Computer Science courses. We conducted twenty-three in-person sessions, including six pilot studies. Out of the twenty-three participants, five were female, seventeen were male, and one preferred not to say. Fourteen participants were undergraduate students, and nine were graduate students. Participants averaged 22.7 years old with 5.2 years of programming experience. At least three-quarters used VS Code as their main programming editor. Participants received a base $25 Amazon gift card and a twenty-cent bonus for each correctly answered comprehension question in their interruption tasks.

D. *Data*

We collected four kinds of data, including demographic surveys, programming and web activity logs, session screencasts, and interview feedback. Our logs included coding and web activity annotated with timestamps. We logged changes to the active window or file,

[2]https://www.pygame.org/docs

[3]https://copilot.microsoft.com/shares/n6Z2yiS4Vb7SCM369u1r7

code saves, code selection, program execution, web search, web page visits, and interruptions, allowing us to reconstruct participants' workflows in detail. To supplement the log data, we recorded more than 35 hours of screen and audio footage. The screencasts captured participants' work, enabling understanding of their development choices and reactions. The post-session, semi-structured interviews explored programmers' experiences, their perceptions of how disruptive interruptions were, their thoughts and strategies for recovering after interruption, and their feedback on the provided tool support.

V. Results

Our analysis focuses on understanding the developers' rebuilding process after interruptions. We examine:

RQ1: What strategies do programmers use to try to rebuild context following an interruption?

RQ2: How does the task that programmers are performing when interrupted relate to the challenge of resuming work?

RQ3: How does historical information (i.e., sub-goal history and code timeline) support the process of resuming work?

To address these research questions, we first established metrics for measuring interruption recovery time.

A. *Measuring Interruption Recovery*

We quantified recovery time using two key measurements: edit lag and execution lag.

Edit Lag: We define edit lag as the time between the end of the interruption and the first code edit that is not a comment. The first edit signifies at least a partial recovery; participants have regained enough context to be willing to make a modification. Participants averaged 1m 34s (SD = 1m 21s) to initially re-engage back into their work.

Execution Lag: We define the execution lag as the time until the first meaningful program execution following the end of an interruption. Participants averaged 6m 31s (SD = 3m 19s) to meaningfully progress their work. We define an execution as meaningful if it:

1) Represented a novel update beyond reverting code or fixing syntax/reference errors.
2) Demonstrated appropriate contextual understanding for that codebase. In particular, the edits in the execution needed to correctly interact with existing components, respect established dependencies (e.g., how the scoring system interacted with physics in Doodle Jump or how pathfinding considered obstacles in Tile Matching), and follow code conventions. A successful execution reflected a working mental model of how the added code fit into the system.
3) Produced expected output indicating task progress (e.g., verifying score updates, adding a new game over screen, extending the hint system to highlight paths). This ensured the measure reflected actual task advancement.

We chose to include two measures of recovery because our observations suggested that programmers often made a first edit while still actively trying to re-orient in the codebase. Thus, the **edit lag** represents a partial recovery, where the **execution lag** represents an upper bound. At the point of the first meaningful execution, the programmer has made positive progress towards a new goal, suggesting that they have already regained enough context to be able to effectively work. We note that there was a substantial difference between these. Programmers' edit lag averaged 1m 34s (SD = 1m 21s), and their execution lag averaged 6m 31s (SD = 3m 19s).

B. *RQ1: Context Recovery Strategies*

After interruptions, participants approached recovery primarily through interacting with code and resources.

1) *Code Interaction Strategies*

When recovering using code, developers directly engaged with the codebase, examining structures and tracing execution flows to rebuild their mental model. The strategies observed were addressing questions relating to testing, debugging, and location identified by LaToza and Myers [53]. 77.9% of the 68 post-interruption periods included at least one code understanding strategy:

- **Program Execution**: In 41.2% of recoveries (44.1% after first interruption, and 38.2% after second), participants re-executed the program within five seconds with minimal or no edits to observe its state. This enabled developers to answer questions like "What works or doesn't?" and "What is the current behavior of the program?".
- **Debugging Output**: In 32.4% of recoveries (32.4% first, 32.4% second), participants used console logging or print statements to verify program behavior. This enabled them to answer questions like "What is the value of this variable?" and "Is this code block being reached?".
- **Code Navigation**: In 52.9% of recoveries (58.8% first, 47.1% second), participants performed keyword searches to locate relevant functions and variables. This addressed questions like "Where is this defined?" and "How was it implemented?".

2) *Interface and Web Resources Strategies*

When recovering using resources, developers engaged with either the interface tools (i.e., content-timeline or sub-goal history) or visited external resources in order to recover understanding and fill knowledge gaps. 94.1% of sessions included at least one interface or web resource strategy for recovery:

- **Web Resources**: In 82.4% of recoveries (94.1% first, 70.6% second), participants reopened previously visited websites to answer "What syntax or parameters did I need?" and "How does this function work?" questions. 12.5% used sub-goal history's browser links; most (87.5%) used native browser features (e.g., "Reopen Closed Tab"), indicating preferences for common practices despite valuing prior information. However, this started to change with additional tool experience. In the first interruption, 6.3% of participants used sub-goal history links to revisit websites. During the second interruption, this increased to 20.8%.
- **Interface Tools**: In 61.8% of recoveries (55.9% first, 67.6% second), developers used interface tools (sub-goal history and content-timeline) to recall recent edits, answering questions like "What was I working on?" and "What changes had I made?". We note that while the usage of both tools increased from the first interruption to the second, the usage of the sub-goal history increased more dramatically. The content timeline usage increased from 41.2% to 47.1% versus 70.6% to 88.2% for the sub-goal history. Among tool users, 42.9% used direct line navigation feature to quickly return to recent edits, addressing "Where in the code was I working?" As U20 explained: "I guess the one thing [to get back into the workflow] is that I have to try to remember, which I also think is very helpful from both interfaces, what change I made most recently. Because I usually have to look at what change I made most recently, and then I have to think about what problem I'm trying to solve." U20's use of recent changes to re-engage is consistent with previous research on programmer interruptions [14]. Non-users typically resorted to manual searching.

Overall, we note that the strategy the developers used was related to their success. Analyzing 68 periods with a point-biserial correlation test, we found a strong negative correlation ($r = -0.27$, $p = 0.026$) between strategy use and edit lag time. Periods with quick edits (<1m 30s) accompanied by web visits or interface interaction (n=28) showed significantly faster initial re-engagement than those without this strategy (n=40). We refer to this strategy as the multi-modal strategy. Developers who used the multi-modal strategy averaged an edit lag of approximately 53s compared to approximately 2m 1s for non-strategy recoveries (Mann-Whitney U, $z = -3.35$, $p < 0.001$). Slower recovery involved extended code reading without changes, verbalizing confusion ("What was I doing?"), or making syntax errors indicating forgotten implementation details. While strongly correlated with faster initial edit times, success rates were not significantly different between groups, suggesting other factors (e.g., inherent difficulty, interruption scenario context) likely contributed.

Twelve participants used the multi-modal approach, which included interface use, code comprehension, and revisiting resources. We defined recovery effectiveness primarily based on speed (measured by edit lag), with recoveries under a minute and a half or so considered effective. Figure 6 shows effective recoveries (U10, U17) with balanced activities and transitioning strategically rather than getting stuck. Ineffective recoveries (U14, U23) had broken transitions with prolonged comprehension and sequential code scanning. While interface tools (sub-goal history and content-timeline) were used in both, during ineffective recoveries, developers struggled to move from context rebuilding into finding and taking actionable steps. Generally, a multi-modal approach, especially quick editing with web/interface tools, correlated with faster, more effective recovery than fixating on a single activity (e.g., extensive comprehension).

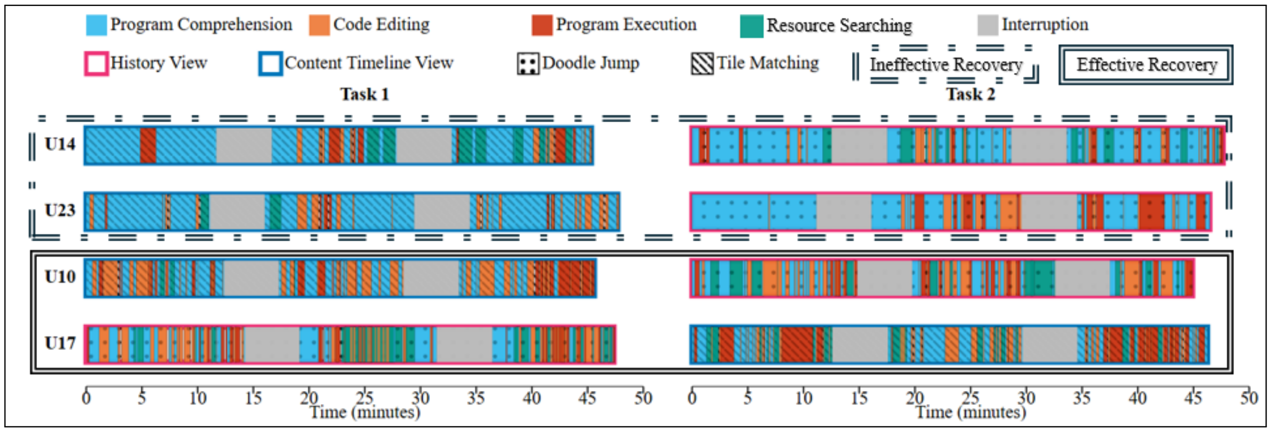


Fig. 6. Post-interruption patterns for users U14 and U23 (ineffective) and for U10 and U17 (effective) are shown. Activity sequences after interruptions reveal characteristic recovery, typically starting with code comprehension (blue) or revisiting web resources (cyan) before productive coding. Effective recoveries feature balanced transitions, contrasting with prolonged comprehension and delayed editing in ineffective ones. The figure highlights approach variation and evolution across interruptions.

C. *RQ2: Task Characteristics and Recovery Difficulty*

We investigated how task characteristics relate to recovery difficulty by analyzing 68 post-interruption periods, classifying them by the development activity programmers were working on: 1) **UI** (e.g., display elements); 2) **Spatial** (e.g., managing coordinate logic in Doodle Jump); or 3) **Algorithmic** (e.g., implementing pathfinding in Tile Matching). We also recorded whether recovery periods were largely focused on a single type of development activity (single) or included multiple kinds (mixed). We explore the impact of task type on recovery in two ways: by recovery success and by recovery time.

1) *Recovery Success by Task Type and Structure*

To investigate the impact of task type on success, we first need to define recovery success. We chose to represent recovery in binary terms as either 1) successful (the developer returned to their pre-interruption task, made relevant code changes, achieved a meaningful execution, and contributed towards task completion before the next interruption or the end of task); or 2) unsuccessful (the developer returned but made no measurable forward progress). These criteria provided a clear, objective basis for assessing recovery across task characteristics. Our analysis revealed that while there are observable differences in the recovery success rates across task types, these were not statistically significant ($\chi^2(2) = 2.63$, $p = 0.268$) (Table I).

TABLE I
RECOVERY DISTRIBUTION BY TASK TYPE

Task Type	Successful Recovery	Unsuccessful Recovery
UI	54.8% (23/42)	45.2% (19/42)
Spatial	66.7% (8/12)	33.3% (4/12)
Algorithmic	35.7% (5/14)	64.3% (9/14)

Our results suggest that programmers tended to struggle the most in recovering from interruptions during algorithmic tasks, followed by spatial tasks, and then user interface tasks.

Algorithmic tasks rely on maintaining complex logical relationships in working memory, which are disrupted by an interruption. U23 articulated this challenge: “When I’m still processing the logic and then the interruption comes, it definitely did, like, mess the [logic] flow up.” Thus, developers needed to rebuild their understanding of these logical structures. U17 approached this by adding debugging statements and reviewing the code structure before attempting major changes. This kind of cautious re-engagement reflects the difficulty of rebuilding the mental models required for algorithmic work.

When recovering from a spatial task interruption, programmers may have been able to rely on visual memory, which often remains partially activated during interruptions [54]. U21 was able to quickly begin making modifications to coordinates without a lot of re-orientation, suggesting that they made use of some recalled information. That said, some participants (U7, U10, U11) expressed frustration when they had to recover from inopportune interruptions. U10, for instance, stated: “I already searched up the documentation and was ready to implement; then the interruption happened so I lost my thought process.” This highlights how spatial tasks, while potentially benefiting from visual memory retention, remain vulnerable to disruption when developers are interrupted during critical information-gathering phases.

Interestingly, mixed tasks, those that included more than one type of programming activity, showed remarkably higher recovery rates; all 14 mixed-task periods were successful. Developers often began with UI modifications and then moved to spatial or algorithmic changes. In contrast, single tasks showed only 40.7% recovery success (22/54 periods).

2) *Recovery Time by Task Type and Structure*

The time developers needed to regain productivity after interruption also reveals critical insights into the cognitive mechanics of resumption. We found that recovery shows different patterns by task type (Table II). Since recovery time data typically does not follow a normal distribution, we applied the following statistical tests: 1) Kruskal-Wallis tests revealed no significant difference in edit lag across task types ($H(2) = 5.06$, $p = 0.078$) and in execution lag across task types ($H(2) = 2.34$, $p = 0.31$); 2) Mann-Whitney U tests revealed no significant difference in edit lag across task structure ($z = -0.86$, $p = 0.39$), but there was a significant difference in execution lag across task structure ($z = -3.19$, $p < 0.005$), where that of mixed tasks was substantially shorter compared to single tasks, suggesting task diversity may help facilitate quicker recovery.

TABLE II
RECOVERY TIME BY TASK TYPE AND STRUCTURE

Category	Edit Lag	Execution Lag	Sample Size
UI	1m 43s	9m 11s	42
Spatial	1m 49s	7m 10s	12
Algorithmic	0m 54s	8m 36s	14
Mixed Tasks	1m 23s	5m 41s	14
Single Tasks	1m 37s	9m 29s	54

Algorithmic tasks showed the fastest initial engagement (edit lag: 54s) but delayed meaningful progress (execution lag: 8m 36s). In reviewing our screencasts, we observed that developers were able to begin typing quickly, but these edits were often exploratory or tentative. For instance, U13 chose to begin with a simple subtask, decreasing the hint count, before re-engaging with a more complex task. This quick but tentative start may be partially explained by the fact that the algorithmic sections of the code were easy to locate within the codebase. However, they were less easy to grasp, which is reflected in the slower progress towards a meaningful execution.

Developers took longer to make an initial edit when completing spatial tasks (edit lag: 1m 49s) because they invested time studying visual relationships before making any changes. Because the coordinate systems were abstract, developers may have been less able to identify an easy change as a starting point. U21, for example, spent over two minutes re-executing to understand a y-position connection before resuming implementation. However, they progressed the most quickly to meaningful execution (7m 10s).

Perhaps the most surprising pattern was that mixed tasks reached meaningful execution 40% faster than single tasks (5m 41s vs. 9m 29s). As with algorithmic tasks, developers appeared to see multiple first steps they could take and were thus able to select one that was achievable. Developers recovering on mixed tasks often strategically resumed through concrete elements first (e.g., U19 on hint button display before the algorithmic part). The ability to detect programming tasks with diverse progress paths may be helpful to time system-generated interruptions in order to lessen their cost.

D. *RQ3: Impact of Historical Information Interfaces*

Study participants interacted with two different interfaces leveraging historical information to aid in interruption recovery: 1) sub-goal history, which provided a development history grouped by programmers' sub-goals, and 2) content-timeline, which presented a simple chronological listing of all development activities. Across all interruption periods, sub-goal history was used more often (73.5% of post-interruption periods) compared to content-timeline (44.1%). A chi-square test showed a statistical significance between interface condition and usage ($\chi^2(1, N = 68) = 6.08$, $p = 0.014$). Users were more likely to use the sub-goal history, suggesting that they might find a summary of their problem solving more valuable for recovering contexts than a simple chronological listing. Participants who were actively using web resources in the five minutes before their interruption relied on the sub-goal history even more strongly. 87.5% of web-active users, those with at least two web visits prior to their interruption, engaged with the sub-goal history, as compared to 44.4% of web-inactive users. Usage of the content-timeline view was lower, with 58.8% of web-active participants and 42.9% of web-inactive participants engaging with the tool. Robust history support may be particularly valuable when programmers are interrupted when engaging in both problem solving and learning, as indicated by higher use of web resources.

User feedback reinforced this stronger preference for sub-goal history, with U8 noting that it "was more helpful because you can track what you were googling," U14 saying it being "cleaner, to the point, like it can write its own commit message," or U23 describing it as "a faster way to access GitHub." That said, U16 and U19 did explicitly express their preference for content-timeline, with U16 stating "It's just simpler, so it's easier to understand," and U19 discussing the similarity of their usual approach: "I feel like most of the time I just get in the terminal, anyway, to see the diff."

1) *Interface Engagement and Recovery Success*

A critical question is whether active engagement with the interface impacted recovery. Recoveries with sub-goal history had a higher success rate than recoveries with content-timeline, 60.0% versus 33.3%. The chi-square test was suggestive but not statistically significant, $\chi^2(1, N = 40) = 2.66$, $p = 0.103$.

Both sub-goal history and content-timeline can provide value to programmers, even when programmers do not actively interact with them. To explore this, we chose to compare the success rates of programmers who interacted with each interface and those who only viewed the interfaces (Table III). For sub-goal history, using the interface resulted in 60.0% successful recovery compared to 66.7% when not used. In contrast, the content timeline showed a more pronounced difference: using the content timeline resulted in a noticeably lower successful recovery rate (33.3%) versus when it was not used (52.6%), though this difference was not statistically significant ($\chi^2(1) = 1.27, p = 0.26$). One possible interpretation of these results is that the use of these interfaces hinders recovery. However, a similar study that included a condition in which participants could take notes but had no interface support for interruption recovery found that note-taking participants were less successful, though the differences they reported were not large and were of marginal significance [14]. Based on our observations, programmers tended to engage with the recovery interfaces when they were unsure of what steps to take, suggesting that the sub-goal history better supported recovery. One explanation for this difference may be that as development activities accumulated over longer tasks, the content timeline became increasingly information-dense, requiring more scrolling and visual search effort to locate relevant information after an interruption.

These findings highlight an important distinction: interface designs supporting passive awareness versus those requiring active engagement. Sub-goal history appears to scaffold developers' mental models through contextual grouping, potentially benefiting users even when not directly consulted, perhaps via concise summaries that aid internalizing work context. Conversely, the content timeline's strictly chronological organization, effective in shorter tasks, appears less effective for complex, longer tasks. As tasks grow, the chronological view may impose additional cognitive load, forcing searches through a dense timeline. Without supportive organization, developers may rely more on existing mental models.

TABLE III
RECOVERY DISTRIBUTION BY INTERFACE USAGE

Interface	Usage	Successful Recovery	Unsuccessful Recovery
Sub-goal History	Used	60.0% (15/25)	40.0% (10/25)
	Not Used	66.7% (6/9)	33.3% (3/9)
Content Timeline	Used	33.3% (5/15)	66.7% (10/15)
	Not Used	52.6% (10/19)	47.4% (9/19)

2) *Interface Support and Execution Time*

Our analysis revealed differences in how interfaces affect recovery time after interruptions. Sub-goal history users took longer to reach meaningful execution after successful recovery (7m 11s) than content-timeline users (5m 35s), though this difference was not statistically significant (Mann-Whitney U, $z = -1$, $p = 0.32$). When participants successfully recovered, those actively using interfaces took significantly longer to reach meaningful execution (7m 20s) than those not interacting with the history interfaces (5m 30s; Mann-Whitney U, $z = -2.26$, $p = 0.024$). We suspect that this may in part be a proxy for recovery difficulty. In cases where recovery was easiest, developers did not need to seek extra information beyond what was already displayed in either interface. In more complex recovery scenarios, they sometimes needed to more actively answer questions in order to rebuild their task context.

3) *Interface Effectiveness for Different Task Types*

Given the differences in performances using the two history interfaces, we also examined how the effectiveness of interface tools varied by task type (Table IV). The most notable performance difference between the two was in recoveries with an algorithmic task. While the numbers are small, three of four algorithmic recoveries using the sub-goal history were successful as compared to two in ten for the content-timeline. Analyzing how developers used these interfaces reveals important differences. Sub-goal history facilitated recovery through relational understanding, helping developers grasp connections between activities rather than just their sequence. This was evident in documentation access: several participants (U15, U17, U19, U20) leveraged sub-goal history's context grouping to quickly reopen API references and search queries. For example, U20 stated "I think there was a website about this" before navigating directly to the visited page via sub-goal history's links.

TABLE IV
RECOVERY SUCCESS RATE BY TASK TYPE AND INTERFACE CONDITION

Task Type	Sub-goal History	Content Timeline	Execution Lag (Successful)
UI	56.5% (13/23)	52.6% (10/19)	7m 15s
Spatial	71.4% (5/7)	60.0% (3/5)	5m 00s
Algorithmic	75.0% (3/4)	20.0% (2/10)	4m 31s

Participants also provided insights into how these interfaces might integrate with different programming workflows. U15 noted that "history would be useful, [the web] link part especially for web dev things such as CSS look up," while U18 suggested the content-timeline would be "a lot more helpful" for game development because of "little changes of positions you did, or colors that may have worked best." U20 also highlighted that sub-goal history would be valuable for team collaboration with its knowledge transfer utility, stating that "okay, somebody was doing this a month ago, and I don't know what they did, but the people coming after can use this [history] interface to see what they were trying to do." The content-timeline was seen as more beneficial towards individual projects, "if there is a lot of research" (U17), and "if it was just all my code, and this [content-timeline] tool recorded all of my edits from the beginning, this would have been useful because then I could go back and just be like, what was I thinking when I made this?" (U16). These perspectives provide insights for our findings that different historical information presentations support different recovery contexts.

VI. DISCUSSION

Our study on programmer interruption recovery provides insights into the processes during context rebuilding and the value of support tools. We discuss implications across three key areas: interruption recovery measurement and process, task-specific patterns, and the role of history representation in supporting context rebuilding.

A. *Measuring and Supporting Interruption Recovery*

The observed interruption costs (edit lag: 1m 34s; execution lag: 6m 31s) align with previous research on interruptions in knowledge work and software development [2], [14], [39]. However, the observable difference between these measures (5 minutes) supports our expectation that first edits do not indicate complete context recovery. This finding has significant implications for measuring and understanding interruption recovery in programming environments.

1) Analyzing Recovery as a Process

Based on our observations, recovery from interruptions follows a series of discrete stages rather than occurring at a single moment. We identified four primary stages in the rebuilding process (Fig. 7):

1) **"Asking" (Initial Disorientation)**: Developers exhibit confusion, asking questions like "What was I doing?" (U7) or "Where were we?" (U14)
2) **"Answering via Navigation" (Re-orientation)**: Developers rebuild context through interface review (U15, U17), code navigation (U11, U12), or search features (U19, U23) to re-orient themselves within their task context, returning to an interrupted task-relevant location within the codebase.
3) **"Answering via Edits" (Tentative Resumption)**: Developers make exploratory edits (U8's deletion/reversion), test executions (U12), or search online (U9, U16, U19) to verify their understanding.
4) **"Productive Work" (Productive Resumption)**: Meaningful progress occurs only after sufficient context rebuilding.

Both the content timeline and sub-goal history supported the early stages of recovery. Using the content timeline, developers could sometimes remember their task by looking at the recent edits and could return to their last edit location. In the sub-goal history, the list of sub-goals provided a recent history of completed tasks as well as the most recent edits, helping to remind developers of their recent actions and supporting their return to that code context. Our metrics included the edit and execution lags, which correspond best to stages three and four of the recovery process.

Further, we noted in some cases the initial navigations and edits were tentative as developers tried to determine if they were in fact in the right code location. In some cases, this resulted in additional navigations and edits, which would also be interesting to capture more formally. U18's experience illustrates a complex recovery. Despite making their first post-interruption edit at 1m 18s (deleting and reverting a line while expressing frustration: "What's different from [Tile Matching's drawMatchLine method] that doesn't work in mine?"), they required around 10 minutes of information searching before achieving meaningful execution. Judging recovery by first-edit metrics would miss this extended context rebuilding process.

The complexity of the recovery processes helps explain the modest improvements observed in our study. Interruption recovery is inherently complex, and our findings reveal that different information presentation approaches serve different recovery needs. The content-timeline view focuses on supporting immediate context restoration through quick access to recent changes via diff views and navigation features, while the sub-goal history view better supports broader context rebuilding, although with the potential cost of processing more information. Nevertheless, casting recovery as a process with stages has implications both for the design of interruption recovery support and for the design of metrics for recovery.

Recovery Support: As we noted, both content-timeline and sub-goal history provided support for the first two stages of recovery, although there is certainly a broader space of designs that would be helpful to explore for the early stages of recovery. It is possible that programmers used the recent edits and sub-goals to support code re-comprehension. Future studies that remove access to this information once in a code context could help to determine the impact of the information in the existing history interfaces for later-stage recoveries. Future research should also explore embedding recovery support directly within that editing context. This might include easy access to resources that the programmer consulted while working on code related to that area, automatic code summarization of existing code, or visualizations of the sequence of sub-goals and related edits that are specific to that context.

Recovery Metrics: Edit and execution lags are best suited to the later stages of recovery. In trying to better understand and measure the recovery process, it would be helpful to introduce metrics that capture the early stages. One possible metric would measure the time it takes the programmer to return to the correct code context. We also observed that recovery contexts vary in complexity. Developing methods to articulate recovery needs, given different interruption circumstances, may also be helpful both to measure the performance of interruption support systems and to identify areas where additional support is needed.

Characterizing Context: There is a wide range of difficulty associated with recovering from an interruption, some of which is attributable to the complexity of the programming context in which a programmer is interrupted. In addition to considering the stages of recovery, approaches for characterizing what information a programmer needs to re-establish in order to begin may be helpful both in measuring recovery and in identifying opportunities to better support it. Our results suggest that some tasks provided multiple ways forward, enabling a programmer to select a first task that was most immediately approachable. In addition to capturing information needs, an important attribute of the context may be how many ways it is possible to make progress towards the current sub-goal. While not significant, we found that algorithmic interruptions were more difficult for our participants to recover from. We suspect that a relative lack of possible paths forward may have contributed to the difficulty associated with algorithmic interruption recovery.

B. History Representation for Recovery Support

Our comparison of sub-goal history and content-timeline views provides insights into supporting interruption recovery through historical information presentation.

1) Value of Contextual Organization

The higher usage rate and numerically higher success rate when used (60.0% vs. content-timeline's 33.3%, Table III) demonstrated the value of sub-goal history's contextual organization, suggesting that understanding relationships between activities offers more value for recovery than a sequence of code diffs. The organization by sub-goal helps developers reconnect with their process (e.g., U20 recalling "I think there was a website about this"). Its presence alone may have influenced how developers organized information mentally during their initial work, creating more robust mental models. Then, when they returned to the IDE, reviewing the sub-goals may have helped them replay the story of their work mentally. Sub-goal history's intention-oriented organization helped re-establish high-level context ("What was I doing? Oh right, defining jump", U7). Developers also found value in the links to resources embedded within each sub-goal. 22.6% of all web visits originated from the sub-goal history. Particularly in contexts where code re-comprehension was difficult, developers made use of easy re-visits to the resources that guided their initial development process.

C. Active and Passive Support

A key challenge in designing interruption recovery support is balancing access to needed information against the cognitive costs of information processing and presentation. Our study explores this trade-off by examining recent versus longer-term information through sub-goal summaries, finding evidence that both are needed in different contexts. As this is an area that is only lightly explored, we expect more research will be needed. Our work suggests that interruption interfaces will need to flexibly support both the quick "Oh right I was in the middle of X," while also supporting deeper investigations "Wait, how did I even get here?"

Developers in our study leveraged the history interfaces both passively, through using information in the default view to help reconstruct their task context, and actively, through selecting parts of the history to review in more detail, navigate to the active code, and re-open resources previously consulted. This dynamic of designing for both passive and active consumption of information is an important challenge for recovery interfaces.

1) Passive Recovery with History

For passive views, one important aspect of the design challenge is the fact that history's information, even for relatively short periods of development, can be overwhelming. We saw evidence of this in the usage of the content timeline. In some cases, reviewing the sequence of code diffs was simply too much for developers. The summaries of changes displayed in the sub-goal history provided an opportunity to simply scan through the activities, and may have served as an index into their own memory of their development process: "First I added the jump. Oh right, and then it didn't work correctly so I looked into event handling." However, in the sub-goal history, only the changes that have reached the threshold to be a sub-goal are annotated in this way. Thus, for the most recent changes, the developer is again relying on code comprehension, albeit with modifications highlighted. This

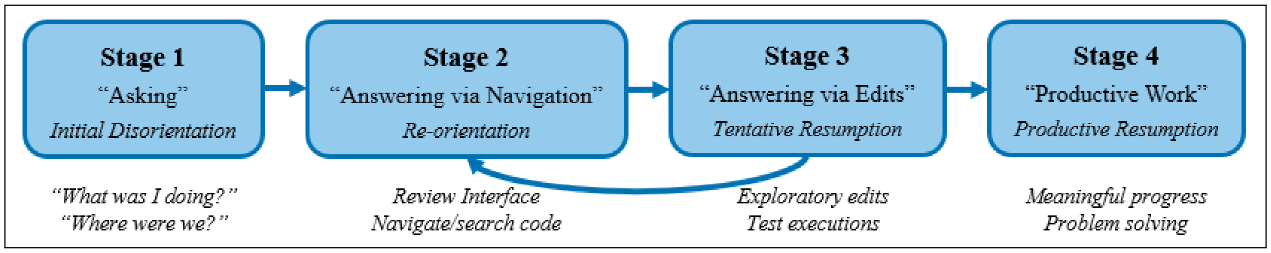


Fig. 7. Developer Interruption Recovery Process illustrating the four-stage process developers experience when recovering from task interruptions: Initial Disorientation ("Asking"), Re-orientation ("Answering via Navigation"), Tentative Resumption ("Answering via Edits"), and Productive Resumption ("Productive Work"). Recovery is a process, not a single moment, suggesting edit lag alone may not capture full resumption.

detailed information for the most recent changes is likely necessary to reconstruct in-progress work, but it might be more easily digested in combination with a summary or overview.

Our results, though inconclusive, suggest there may also be differences in the supports that developers need, which vary based on task. Future work on passive support for interruption recovery should more deeply explore how information needs vary by task type and how best to support those information needs while minimizing the amount of information developers must process in support of recovery.

2) *Active Recovery with History*

When developers interacted with history, it was to extract more detailed information or to use the history as a navigational support. Both of these are important roles. While we have started to map history information to common development questions [53], these are general development questions. Future research should explore the subset of questions that programmers need during recovery and use that as a driver of future designs, making the answers to less frequently asked questions available through active interaction with recovery histories. Additionally, as noted in the discussion of recovery as a process, exploring views supporting programmers once they have re-established a particular location within the code could focus on questions that arise as they begin to actively consider modifications.

VII. Future Work

Based on our discussion of programmers' interruption recovery, several promising future directions remain:

- **Understanding the Diversity of Recovery Needs:** Our work suggests that the specific context in which programmers are interrupted may influence their recovery support needs. In this study, we focused on a particular (brief/externally-initiated) interruption scenario and a limited set of programming tasks. Additional work is needed to understand how programmers' recovery processes are impacted by different types and lengths of interruptions. Additionally, the nature of programmers' work when they are interrupted may result in differing needs. Characterizing different interrupted work profiles may enable us to better identify what supports are necessary to resume work.
- **AI-powered Resumption Assistance:** While LLMs alone are not a complete solution for resumption aid, they could provide targeted support at specific stages of the interruption recovery process. This may be particularly effective when they are able to leverage history information. For example, in the initial orientation stage of a new programming session, LLMs could provide a quick contextual summary of recent goals achieved and summarize partial progress: "During your last working session, you were implementing dark theme support to the UI components." A conversational interface might help programmers to quickly find and re-immerse themselves in historical context needed to move forward. A staged-recovery lens to identify different needs may help to inform the design of nuanced support for interruption recovery.
- **Task Recovery Support:** While our current codebases present a starting point, they cannot represent the full spectrum of tasks developers might engage in. Future work could consider exploring the diversity of codebases and tasks. As task type and context contributed to task-switching disruptiveness in software development [8], understanding the interruptibility of different task types is equally important for designing non-disruptive passive support. Certain tasks may be highly sensitive to interruption, while others allow for more flexibility. Research is needed to establish a taxonomy of task interruptibility and to inform the design of passive support tools that can be easily toggled or configured by developers based on their current work context.

VIII. Limitations

Several limitations should be considered when interpreting our findings:

Generalizability and Ecological Validity: Our current work has focused on a controlled study with a relatively small size of student participants and modest codebases (150-250 LOC). Although the study tasks were designed to simulate real-world scenarios, they may not fully replicate the complexities of naturalistic software development. More work remains to extend user populations, diversity of codebases, and settings. While our study design does not cover all possible interruption settings, we believe it has helped to build a solid foundation for understanding how users recover from interruptions. The controlled environment helped us isolate different factors that contribute to recovery and formulate the four-stage recovery process, as well as allowed us to systematically compare the post-interruption periods and study different recovery strategies.

Internal Validity: Although users were assigned codebases randomly, the distribution of task types encountered with respective interfaces may not be perfectly balanced, potentially confounding interface comparisons with task-specific recovery challenges. Further, the inherent difficulty of interruption scenarios varied across sessions because of elements like interruption moment and code familiarity, making it challenging to separate the effects of interface design from the inherent difficulty of resuming work.

Construct Validity: Participants had limited exposure to both the content timeline and the sub-goal history interface. Consequently, their usage may not reflect that of experienced users. We did see an increase in usage for both interfaces between the first and second interruption, suggesting that developers were beginning to incorporate the capabilities of these histories for recovery into their processes. However, longer-term studies would be valuable in assessing the impact of history support for recovery.

IX. Conclusion

The ability to support developers in recovering from interruptions has the potential to dramatically improve productivity within software development. To date, relatively little work has explored how to support interruption recovery. We contribute an exploratory study of two interfaces that incorporate historical information: the content timeline interface and the sub-goal history interface. Programmers in our study approached recovery as a process that involved posing and answering questions about the state of the program. They leveraged a combination of code navigation, code comprehension, and revisiting web resources to reconstruct a sufficient understanding of the code to resume work. History information primarily supported the early stages of interruption recovery, helping programmers to return to their code context and recall recent changes. Recoveries using a multi-modal strategy with both an early edit and the use of a history tool or web visits were significantly faster to make progress after an interruption. The sub-goal history supplemented this with summaries of code changes and easy access to the resources that supported those changes. Overall, we found that programmers relied more on the sub-goal history than the content-timeline interface. Further, their usage increased from the first to the second interruption, suggesting a move to incorporate history tools into their recovery processes.

Acknowledgement

This material is based upon work supported by the National Science Foundation, USA, under Grant No. 2128128. We thank our pilot study and usability study participants. We also sincerely thank the reviewers for their thoughtful and constructive feedback.

References

[1] A. N. Meyer, L. E. Barton, G. C. Murphy, T. Zimmermann, and T. Fritz, "The work life of developers: Activities, switches and perceived productivity," *IEEE Transactions on Software Engineering*, vol. 43, no. 12, pp. 1178–1193, 2017.

[2] C. Parnin and S. Rugaber, "Resumption strategies for interrupted programming tasks," *Software Quality Journal*, vol. 19, pp. 5–34, 2011.

[3] V. M. González and G. Mark, "" constant, constant, multi-tasking craziness" managing multiple working spheres," in *Proceedings of the SIGCHI conference on Human factors in computing systems*, 2004, pp. 113–120.

[4] G. Mark, V. M. Gonzalez, and J. Harris, "No task left behind? examining the nature of fragmented work," in *Proceedings of the SIGCHI conference on Human factors in computing systems*, 2005, pp. 321–330.

[5] H. Sanchez, R. Robbes, and V. M. Gonzalez, "An empirical study of work fragmentation in software evolution tasks," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 2015, pp. 251–260.

[6] B. P. Bailey and J. A. Konstan, "On the need for attention-aware systems: Measuring effects of interruption on task performance, error rate, and affective state," *Computers in human behavior*, vol. 22, no. 4, pp. 685–708, 2006.

[7] G. Mark, D. Gudith, and U. Klocke, "The cost of interrupted work: more speed and stress," in *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, 2008, pp. 107–110.

[8] Z. S. H. Abad, O. Karras, K. Schneider, K. Barker, and M. Bauer, "Task interruption in software development projects: What makes some interruptions more disruptive than others?" in *Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering 2018*, 2018, pp. 122–132.

[9] B. Vasilescu, K. Blincoe, Q. Xuan, C. Casalnuovo, D. Damian, P. Devanbu, and V. Filkov, "The sky is not the limit: multitasking across github projects," in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 994–1005.

[10] T. Gillie and D. Broadbent, "What makes interruptions disruptive? a study of length, similarity, and complexity," *Psychological research*, vol. 50, no. 4, pp. 243–250, 1989.

[11] J. P. Borst, N. A. Taatgen, and H. van Rijn, "What makes interruptions disruptive? a process-model account of the effects of the problem state bottleneck on task interruption and resumption," in *Proceedings of the 33rd annual ACM conference on human factors in computing systems*, 2015, pp. 2971–2980.

[12] S. T. Iqbal and E. Horvitz, "Disruption and recovery of computing tasks: field study, analysis, and directions," in *Proceedings of the SIGCHI conference on Human factors in computing systems*, 2007, pp. 677–686.

[13] M. Kersten and G. C. Murphy, "Using task context to improve programmer productivity," in *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, 2006, pp. 1–11.

[14] C. Parnin and R. DeLine, "Evaluating cues for resuming interrupted programming tasks," in *Proceedings of the SIGCHI conference on human factors in computing systems*, 2010, pp. 93–102.

[15] J. G. Trafton, E. M. Altmann, and D. P. Brock, "Huh, what was i doing? how people use environmental cues after an interruption," in *Proceedings of the human factors and ergonomics society annual meeting*, vol. 49, no. 3. SAGE Publications Sage CA: Los Angeles, CA, 2005, pp. 468–472.

[16] J. Brandt, P. J. Guo, J. Lewenstein, M. Dontcheva, and S. R. Klemmer, "Two studies of opportunistic programming: interleaving web foraging, learning, and writing code," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2009, pp. 1589–1598.

[17] ——, "Writing code to prototype, ideate, and discover," *IEEE software*, vol. 26, no. 5, pp. 18–24, 2009.

[18] J. Brandt, P. J. Guo, J. Lewenstein, and S. R. Klemmer, "Opportunistic programming: How rapid ideation and prototyping occur in practice," in *Proceedings of the 4th international workshop on End-user software engineering*, 2008, pp. 1–5.

[19] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung, "An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks," *IEEE Transactions on software engineering*, vol. 32, no. 12, pp. 971–987, 2006.

[20] D. E. Rose and D. Levinson, "Understanding user goals in web search," in *Proceedings of the 13th international conference on World Wide Web*, 2004, pp. 13–19.

[21] C. Sadowski, K. T. Stolee, and S. Elbaum, "How developers search for code: a case study," in *Proceedings of the 2015 10th joint meeting on foundations of software engineering*, 2015, pp. 191–201.

[22] X. Xia, L. Bao, D. Lo, P. S. Kochhar, A. E. Hassan, and Z. Xing, "What do developers search for on the web?" *Empirical Software Engineering*, vol. 22, pp. 3149–3185, 2017.

[23] F. J. Stangl and R. Riedl, "Interruptions in the workplace: an exploratory study among digital business professionals," in *International Conference on Human-Computer Interaction*. Springer, 2023, pp. 400–422.

[24] E. R. Sykes, "Interruptions in the workplace: A case study to reduce their effects," *International journal of information management*, vol. 31, no. 4, pp. 385–394, 2011.

[25] S. Addas and A. Pinsonneault, "The many faces of information technology interruptions: a taxonomy and preliminary investigation of their performance effects," *Information Systems Journal*, vol. 25, no. 3, pp. 231–273, 2015.

[26] D. C. McFarlane and K. A. Latorella, "The scope and importance of human interruption in human-computer interaction design," *Human-Computer Interaction*, vol. 17, no. 1, pp. 1–61, 2002.

[27] M. Czerwinski, E. Horvitz, and S. Wilhite, "A diary study of task switching and interruptions," in *Proceedings of the SIGCHI conference on Human factors in computing systems*, 2004, pp. 175–182.

[28] J. Jin and L. A. Dabbish, "Self-interruption on the computer: a typology of discretionary task interleaving," in *Proceedings of the SIGCHI conference on human factors in computing systems*, 2009, pp. 1799–1808.

[29] Z. S. H. Abad, M. Noaeen, D. Zowghi, B. H. Far, and K. Barker, "Two sides of the same coin: Software developers' perceptions of task switching and task interruption," in *Proceedings of the 22Nd International Conference on Evaluation and Assessment in Software Engineering 2018*, 2018, pp. 175–180.

[30] A. N. Meyer, E. T. Barr, C. Bird, and T. Zimmermann, "Today was a good day: The daily life of software developers," *IEEE Transactions on Software Engineering*, vol. 47, no. 5, pp. 863–880, 2019.

[31] J. Chong and R. Siino, "Interruptions on software teams: a comparison of paired and solo programmers," in *Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work*, 2006, pp. 29–38.

[32] S. Tremblay, F. Vachon, D. Lafond, and C. Kramer, "Dealing with task interruptions in complex dynamic environments: are two heads better than one?" *Human factors*, vol. 54, no. 1, pp. 70–83, 2012.

[33] R. Harr and V. Kaptelinin, "Unpacking the social dimension of external interruptions," in *Proceedings of the 2007 ACM International Conference on Supporting Group Work*, 2007, pp. 399–408.

[34] L. Dabbish and R. E. Kraut, "Controlling interruptions: awareness displays and social motivation for coordination," in *Proceedings of the 2004 ACM conference on Computer supported cooperative work*, 2004, pp. 182–191.

[35] S. T. Iqbal and B. P. Bailey, "Investigating the effectiveness of mental workload as a predictor of opportune moments for interruption," in *CHI'05 extended abstracts on Human factors in computing systems*, 2005, pp. 1489–1492.

[36] P. D. Adamczyk and B. P. Bailey, "If not now, when? the effects of interruption at different moments within task execution," in *Proceedings of the SIGCHI conference on Human factors in computing systems*, 2004, pp. 271–278.

[37] S. Leroy, "Why is it so hard to do my work? the challenge of attention residue when switching between work tasks," *Organizational Behavior and Human Decision Processes*, vol. 109, no. 2, pp. 168–181, 2009.

[38] Y. Ma, Y. Huang, and K. Leach, "Breaking the flow: A study of interruptions during software engineering activities," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–12.

[39] M. Züger and T. Fritz, "Interruptibility of software developers and its prediction using psycho-physiological sensors," in *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, 2015, pp. 2981–2990.

[40] D. Hu and S. W. Lee, "Screentrack: Using a visual history of a computer screen to retrieve documents and web pages," in *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, 2020, pp. 1–13.

[41] ——, "Scrapbook: Screenshot-based bookmarks for effective digital resource curation across applications," in *Proceedings of the 35th Annual ACM Symposium on User Interface Software and Technology*, 2022, pp. 1–13.

[42] B. P. Bailey and S. T. Iqbal, "Understanding changes in mental workload during execution of goal-directed tasks and its application for interruption management," *ACM Transactions on Computer-Human Interaction (TOCHI)*, vol. 14, no. 4, pp. 1–28, 2008.

[43] H. Kaur, A. C. Williams, D. McDuff, M. Czerwinski, J. Teevan, and S. T. Iqbal, "Optimizing for happiness and productivity: Modeling opportune moments for transitions and breaks at work," in *Proceedings of the 2020 CHI conference on human factors in computing systems*, 2020, pp. 1–15.

[44] J. Fogarty, A. J. Ko, H. H. Aung, E. Golden, K. P. Tang, and S. E. Hudson, "Examining task engagement in sensor-based statistical models of human interruptibility," in *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, 2005, pp. 331–340.

[45] J. Allen and C. Kelleher, "An exploratory study of programmers' analogical reasoning and software history usage during code re-purposing," in *Proceedings of the 2024 IEEE/ACM 17th International Conference on Cooperative and Human Aspects of Software Engineering*, 2024, pp. 109–120.

[46] C. W. Krueger, "Software reuse," *ACM Computing Surveys (CSUR)*, vol. 24, no. 2, pp. 131–183, 1992.

[47] D. Kirsh, "The context of work," *Human–Computer Interaction*, vol. 16, no. 2-4, pp. 305–322, 2001.

[48] V. T. T. Pham and C. Kelleher, "Code histories: Documenting development by recording code influences and changes in code," *Journal of Computer Languages*, vol. 82, p. 101313, 2025.

[49] H. Jin and J. Kim, "Codetree: A system for learnersourcing subgoal hierarchies in code examples," *Proceedings of the ACM on Human-Computer Interaction*, vol. 8, no. CSCW1, pp. 1–37, 2024.

[50] A. Oulasvirta and P. Saariluoma, "Surviving task interruptions: Investigating the implications of long-term working memory theory," *International Journal of Human-Computer Studies*, vol. 64, no. 10, pp. 941–961, 2006.

[51] C. K. Foroughi, P. Malihi, and D. A. Boehm-Davis, "Working memory capacity and errors following interruptions," *Journal of Applied Research in Memory and Cognition*, vol. 5, no. 4, pp. 410–414, 2016.

[52] S. Leroy and A. M. Schmidt, "The effect of regulatory focus on attention residue and performance during interruptions," *Organizational Behavior and Human Decision Processes*, vol. 137, pp. 218–235, 2016.

[53] T. D. LaToza and B. A. Myers, "Hard-to-answer questions about code," in *Evaluation and usability of programming languages and tools*, 2010, pp. 1–6.

[54] C. Schneegass, V. Füseschi, V. Konevych, and F. Draxler, "Investigating the use of task resumption cues to support learning in interruption-prone environments," *Multimodal Technologies and Interaction*, vol. 6, no. 1, p. 2, 2021.