Journal of Information Systems Education, 36(1), 1-12, Winter 2025
https://doi.org/10.62273/BTKM5634

Teaching Tip
Rethinking How We Teach Git: Pedagogical Recommendations and Practical Strategies for the Information Systems Curriculum

Gerit Wagner
Laureen Thurner
Faculty Information Systems and Applied Computer Science
Otto-Friedrich-Universität Bamberg
Bamberg, Germany
gerit.wagner@uni-bamberg.de, laureen.thurner@uni-bamberg.de

ABSTRACT

Git, as the leading version-control system, is frequently employed by software developers, digital product managers, and knowledge workers. Information systems (IS) students aspiring to fill software engineering, management, or research positions would therefore benefit from familiarity with Git. However, teaching Git effectively can be challenging, as students in IS and other disciplines report themselves overwhelmed by the plethora of Git's detailed commands and options, including those involved in local setup and secure shell (SSH) connections. From our view, such technical considerations distract students, and even prevent them from developing a deeper understanding of the Git model and its underlying concepts. Ideally, teaching efforts should convey a solid understanding of the Git model and thereby enable students to ask the right questions and look up the relevant commands. With this teaching tip, we therefore challenge the common approach to organizing Git teaching materials. In particular, we draw on established pedagogical theory to propose a novel approach that employs a new, macro-level ordering of contents beginning with the concept of branches and then proceeding to committing and collaboration. We present several practical strategies that make this approach feasible. In addition, we recommend that teachers clearly separate conceptual from applied learning and present the more challenging transfer questions at the end of the course. Our hope is to stimulate reflection on the most effective ways to teach Git to future professionals.

Keywords: Git, Collaborative versioning, Distributed version control, Open source, Software engineering, Information systems (IS)

1. INTRODUCTION

With a dominant market share, Git is the leading version control system in use today (Dohmke, 2023). The distributed and efficient model of Git makes it particularly useful for teams that are not co-located but contribute to the same code-base in an asynchronous manner. Originating from the open-source community, the system has seen rapid adoption in the private sector, with thousands of freelancers, startups, and large tech companies using it. GitHub has turned into the largest hosting platform for Git repositories, allowing developers to manage, store, and share software being worked on concurrently. In 2018, when Microsoft and Google competed to acquire it, GitHub had approximately 83 million active accounts and 200 million repositories. Following Microsoft's acquisition for $7.5 billion, GitHub established itself as the dominant developer platform (Microsoft News Center, 2018), and, as of January 2023, it reported having over 100 million developers and hosting over 420 million repositories. Today, a range of large corporations, including Microsoft, Google, Meta (Facebook), IBM, Netflix, and Airbnb employ Git and GitHub for internal software development as well as their comprehensive open-source programs.

The diffusion of Git raises new requirements for students in the field of information systems (IS). Due to its essential role in the software development industry, surveys of skill requirements and industry recruiters increasingly recognize the need for versioning skills (Aasheim et al., 2012; Brooks et al., 2018; Cummings & Janicki, 2020). Experience with Git also figures prominently in a range of typical IS job roles (DeSanto et al., 2023):

- Software developers rely on Git in their workflow to facilitate collaboration and code management in both front-end and back-end development settings (Deshpande et al., 2016; Latinovic & Pammer-Schindler, 2021). Further, developers often integrate code formatters and code linters into their Git repositories to ensure consistency of code style and quality. Additionally, Git enables the practice of rapid prototyping and agile development, allowing developers to experiment with new features without affecting the main codebase.

- DevOps managers use Git to orchestrate the steps involved in the continuous integration pipeline, including code versioning, automated testing, and deployment (Bou Ghantous & Gill, 2017). For example, GitLab, a popular DevOps platform, provides a comprehensive set of tools for managing the software development lifecycle. These tools enable development and operations teams to seamlessly collaborate, thus streamlining the entire software delivery process.
- Git and platforms like GitHub or Gitlab provide a central location for such digital product-management tasks as specifying requirements, tracking development progress, resolving bugs, and addressing security vulnerabilities. Therefore, in addition to programmers, product owners, portfolio managers, and IT architecture managers use it (Smits & Mogos, 2013; Thummadi et al., 2017) to track requirements, coordinate development activity, and assist the development team in completing predefined milestones.

These examples show that Git, its workflows, and associated tools have become essential for professionals and students alike, as they provide the necessary infrastructure for effective code management, collaboration, and project coordination.

Most teaching materials present the contents in a technical manner familiar to computer science students, but not to IS students, who prepare for job roles involving coding, but also organizing, coordination, and leadership skills (Bou Ghantous & Gill, 2017; Latinovic & Pammer-Schindler, 2021; Smits & Mogos, 2013). In this sense, the teaching of Git to IS students presents an exemplary setting in which technical and managerial skills intertwine. As our review of existing teaching resources shows, the majority of Git courses targets computer science students. Thus, these teaching materials may be less appropriate for IS students, who often have limited experience operating command-line applications, interpreting error messages, or thinking in terms of (directed acyclic) graphs. Therefore, the need to develop specific approaches and materials to teach Git to IS students is evident.

In particular, our review of extant teaching materials suggests that current teaching approaches are often constrained by technical considerations. For example, a typical Git introduction template begins with the command *git init* to set up repositories, followed by instructions for students to configure their Git name and email, as these are required for the *init* command, and concludes with guidance on setting default editors, which is necessary for writing commit messages. Further technical setup is related to the secure shell (SSH) protocol configuration, which handles remote encrypted connections between computers and includes generation and registration of asymmetric keys. While these steps are certainly needed for some use cases, they constitute specific setup tasks that are not needed frequently in day-to-day use of Git.

Our goal is to challenge and rethink existing approaches to teaching Git which typically begin with, presumedly necessary, technical setup tasks. Yet, these often produce errors for students, and interfere with their efforts to gain an in-depth understanding of Git. Unlike current approaches, the approach we propose is driven by pedagogical considerations that are focused on clearly conveying Git's fundamental model early in a course. Therefore, it is aimed at effectively preparing IS students to apply and use Git. Prior to presenting this approach, we examine previous teaching efforts and problematize the challenges inherent in the predominant approach to teaching Git. Using this as a foundation, we then describe our approach, which is based on a different ordering of contents at the macro level. In addition, we provide the pedagogical principles on which our proposed model is based, practical strategies for its implementation, and examples of its application. Based on initial feedback, the IS students to whom we taught Git using our proposed teaching model perceived it as a pleasant learning experience leading to better learning outcomes than do those approaches commonly in use today.

2. RELATED WORK

2.1 Key Competence Areas in Git

We distinguish three areas of Git competences—committing, branching, and collaborating—which are essential for efficient software development and project management. These topics, which are summarized below, are covered in most Git teaching materials.

2.1.1 Committing. Developers modifying their code *add* and *commit* selected changes to their repository. These snapshots of the project's history allow developers to track and revert changes if needed. In committing changes, knowing how to create *atomic commits* is essential. These are changes that contain a single complete and coherent unit of work, i.e., one that stands on its own, and is dedicated to accomplishing one and only one task without depending on or affecting other commits. A Git user should attach a clear and descriptive message to an atomic commit that summarizes the changes made so as to have a concise version history of the software's development and facilitate effective collaboration (Westby, 2015).

2.1.2 Branching. This refers to the creation of separate lines of development in a code repository. The operations of creating branches, adding commits to selected branches, and merging branches, effectively build a directed acyclic graph, the underlying conceptual structure of Git. It is almost impossible to use Git effectively without a clear mental model of how different branching operations contribute to the directed acyclic Git graph. In practice, branching allows developers to begin separate lines of commits from the codebase. By switching to a selected branch, they can work on different features or bug fixes without affecting the main codebase, and then merge the code once it is ready. Therefore, branching is a facility that is particularly useful in large projects with multiple developers, as it allows for efficient parallel development (Westby, 2015).

2.1.3 Collaborating. This remotely enables the development of software by teams working asynchronously in different locations. Typically, code is developed in local repositories until it is judged ready to be uploaded (or *pushed*) to the remote repository, e.g., on GitHub. Remote repositories do not allow a noncontributor to *push* changes to the repository; instead, the developer must open a *pull request* to the original remote repository. Thus, *push*, *pull*, and *pull requests* allow developers working remotely to synchronize their local changes across repositories and so enable asynchronous remote work with

considerable flexibility in software development and collaboration.

2.2 Teaching Resources

In the plethora of teaching resources, committing is typically presented prior to branching and collaborating. Table 1 provides an overview of massive open online courses (MOOCs) dedicated to Git and GitHub. Three of the courses are provided via Coursera by Google, Meta, and Atlassian University and one by Software Carpentry. These courses are professionally designed, include multimedia materials, and require from 8-16 hours to complete. The *Version Control With Git* course, offered by Atlassian University, includes interactive learning, whereas the others provide self-learning environments.

Course Title (Provider)	Target	Duration
Version Control With Git (Atlassian, via Coursera)	Beginner	13h
Introduction to Git & GitHub (Google, via Coursera)	Beginner	16h
Version Control (Meta, via Coursera)	Beginner	13h
Version Control With Git (Software Carpentry)	Intermediate	8h

Table 1. Massive Open Online Courses Dedicated to Git and GitHub

As shown in Table 2, these courses all present the material in a common order, beginning with committing and then proceeding with branching or collaborating. To the best of our knowledge, no teaching resources deviate from the *committing–branching–collaborating* sequence.

Course Title	Order of Competences
Version Control With Git	1. Initializing, committing, pushing 2. Branching and merging 3. Workflows for collaborating remotely
Introduction to Git & GitHub	1. Using Git locally: committing and branching 2. Working with remotes 3. Collaborating
Version Control	1. Overview and history of Git 2. Command-line introduction 3. Working with Git and setup
Version Control With Git	1. Automated version control 2. Git setup 3. Creating a repository 4. Tracking changes 5. Collaborating

Table 2. The Common Order of Competences in Teaching Materials

While numerous teaching resources and academic papers have explored methods for teaching Git, none have challenged the conventional sequence of committing, branching, and collaborating. Haaranen and Lehtinen (2015) highlight the usefulness of version control systems in a classroom setting, incrementally presenting features of Git and incorporating them into the course workflow to distribute exercises, streamline student assessment, and facilitate project collaboration. Tafliovich et al. (2019) report teaching Git-based collaboration in a large, free open-source software project incorporating project-based learning and service learning principles. Their course incorporated effective hands-on teaching of essential software engineering topics and a capstone project. In an introduction to Git, Pathak (2020) provides a comprehensive overview, including such topics as setting up a central Git server, granting passwordless access to developers, and registering public keys with the Git server. Additional topics covered are making the first commit and pushing changes. Jabrayilzade et al. (2022) address the lack of Git knowledge among computer science and software engineering students, implementing a four-session training program as part of an object-oriented software engineering course. Finally, Vial and Negoita (2018) focus on teaching programming and the use of GitHub for both IS students or non-programmers.

A significant number of researchers discuss pitfalls, challenges, and errors repeatedly encountered in Git's use. De Rosso and Jackson (2013) outline issues that "puzzle even experienced developers" (p. 37), identifying an array of undesirable or counterintuitive properties of Git based on a conceptual design analysis. Isomöttönen and Cochez (2014) surveyed students of computing curricula about the different challenges they encountered when using the Git command line. Lippa (2016) organized a workshop titled "Get Out of Git Hell" to combat a range of systemic problems with Git's use that have slowed down or even blocked the work of teams. Lippa (2016) attributes these problems to "poor tool design, misuse or misconfiguration of the command line interface, and lack of understanding of the 'nuts and bolts' of the tool" (p. 22). Finally, Eraslan et al. (2020) draw attention to common errors and pitfalls in the use of Git, including the neglect of feature branches, committing of unrelated files, and the lack of appropriate branching and merging practices.

In conclusion, the academic research summarized above has highlighted the importance of teaching Git in different settings, while recognizing the presence of persistent challenges. We have not found any resources that suggest deviating from the *committing - branching - collaborating* sequence as a solution. Our work challenges this order and builds on prior pedagogical theory to envision a more effective strategy for teaching Git to IS students.

3. COURSE SETTING

We teach Git in two settings, specifically as a part of a lecture-based course and as part of a capstone project. These sessions were offered across three semesters and were primarily taken by undergraduate IS students. As shown in Table 3, the percentage of female students varied, but slightly exceeded the percentage of female students enrolled in the department. While introductory programming courses, including Java, algorithms, and data structures, are recommended as prerequisites, they are not mandatory. The Git sessions do not require programming skills, although familiarity with command-line interfaces or graph data structures is helpful.

The lecture, *Introduction to Digital Work*, included a four-hour session covering Git basics and a four-hour session covering the *Git Collaboration Game* (Wagner, 2024). The goal was to familiarize students with Git as an essential technology

for digital work and collaborative content creation. As such, this setting did not involve programming but used editing of *Markdown* files as a basis for practical exercises. These focused sessions were complemented by sessions covering open-source philosophy and knowledge-management approaches based on *Markdown* files and Zettelkasten principles, both areas where Git knowledge is highly beneficial.

		Number of students		
		Summer 2023	Winter 2023	Summer 2024
Course	Introduction to Digital Work (lecture)	25	-	6
	Open-Source Project (capstone)	14	4	15
Major	Information systems	37	4	18
	Other	2	0	3
Gender	Male	28	4	12
	Female	11	0	9
Total		39	4	21

Table 3. Overview of Courses and Number of Students

The capstone project, titled *The Open-Source Project*, contained a four-hour session dedicated to Git, including collaboration with specific follow-up instructions. The aim was to teach students how to apply Git in a practical group programming project. Working in groups of three or four, students contributed to the development of CoLRev (Wagner & Prester, 2024), a Python library for literature reviews. In addition to Git, we offered introductory sessions on Python and development of Python packages, each four hours long. To complete the project, each student was expected to contribute code by using the Git collaboration model of branches, forks, and pull requests.

The key competencies for both courses are:

1. **Understand Git conceptually**: Students should be familiar with the key concepts in the three areas. In branching, they should be able to explain the elements of commits, including the directed acyclic Git graph, as well as branch and head pointers and corresponding operations. In committing, they should be able to explain the three areas of the working directory, the staging area, and the Git repository, as well as the operations to move file changes across these areas. In collaborating, they should be able to explain different setups for remote repositories, including synchronization through push, pull, and pull requests.
2. **Apply basic Git operations**: Students should be able to operate Git on the command line using operations to manipulate branches (*git branch*, *checkout*, *switch*, *merge*), create commits (*git add*, *commit*, *restore*, *reset*), and collaborate (*git pull*, *push*, *fetch*). They should recognize these operations in graphical software packages and online platforms.
3. **Manage Git in small projects**: Students should understand how the three Git areas work together. They should be able to assess the state of Git repositories, describe how Git could be used for open-source projects or rapid prototyping, and select a suitable setup for small projects.

4. PEDAGOGICAL APPROACH AND PRACTICAL STRATEGIES FOR TEACHING GIT

Our recommendations are the result of a prolonged effort to effectively teach Git in the IS curriculum, as summarized briefly in the following reflection. Although students generally recognized the significance of Git, our first sessions showed that their initial motivation to learn it quickly eroded when faced with technical difficulties. Through our ongoing conversations with students, teaching assistants, and colleagues, we identified two fundamental challenges with the conventional approach of teaching Git.

First, technical errors in the Git setup or process often created situations in which students were unable to complete the next step or even failed to follow subsequent tasks. This issue was exacerbated when students worked on their own devices, each with its own idiosyncratic environment. Despite the support of dedicated teaching assistants, technical errors and dependencies were the main reason students fell behind and experienced frustration. Second, as discussed previously (De Rosso & Jackson, 2013; Eraslan et al., 2020; Isomöttönen & Cochez, 2014; Lippa, 2016), the complexity of Git is a persistent challenge that can quickly overwhelm students. Much like beginning swimmers attempting to learn the front crawl swimming technique with flutter kicks, appropriately timed rotation, and breathing, students were simply overwhelmed by Git's cognitive complexity. Employing this analogy to theorize our teaching approach (Hassan et al., 2023) inspired us to experiment with changes corresponding to those employed by swimming instructors.

The focus of the first change we introduced was isolating each competence area so as to create both cognitively and technically self-contained sessions. This modification is particularly challenging for the branching and collaborating areas, which often involve distracting tasks related to the setup, the creation of changes to a file, or errors associated with these. Some of our colleagues were even surprised at our suggestions to practice branching or collaborating without first setting up a local repository. Consulting prior teaching materials revealed that few, if any, deliberate attempts had been made to practice the different competence areas separately. Throughout the semesters, we continued to disentangle learning settings cognitively and technically, gradually observing improvements that confirmed what is common knowledge among swimming coaches: the benefits of breaking down complex tasks and practicing them separately.

Second, after disentangling the three competence areas, we questioned the order of the contents of Git instruction, which was previously viewed as dictated by necessity. Stated simply, and with some degree of exaggeration, the rationale in the early stages of teaching Git was that "We cannot start with collaboration because it is first *necessary* to have a branch, which requires some commits, and the setup of a repository, with corresponding instructions to use the command line. Therefore, starting with the command line is *necessary*." Upon

disentangling the learning settings, we identified options to practice collaboration, branching, and committing without such constraining dependencies. In essence, learning each area separately made it possible to organize the macro-level order of competence areas in a way that is most conducive to the learning process. In this new setting, we quickly agreed to start with the branching and the Git graph, which not only reflects how developers begin their work but also covers fundamental meso-level concepts that help students understand the other areas. Analogously, swimming coaches start with a swimming board to help participants develop a relaxed rhythm of leg kicks as a basis to learning arm pulls, breathing, and details of timing and body position.

Third, as our disentangling and reordering efforts appeared to be working, students seemed capable of handling more challenging questions, and we consequently included transfer questions at the end of the course to stimulate thinking across areas. Apparently, our deliberate effort to isolate and connect concepts resulted in students' no longer feeling overwhelmed by complexity and equipped with a solid understanding of Git concepts. To conclude with our analogy to swimming lessons, which may simulate the challenges of open water racing conditions, once a basic skill is acquired, exposing learners to additional challenges can be an ingredient to master a skill even more effectively.

Figure 1 depicts our overall approach to teaching Git. Following it is a presentation of the pedagogical principles, including their bases in cognitive and pedagogical theory, and practical strategies to implement them. To increase readability, we proceed from the most fundamental changes in the traditional teaching of Git to the follow-up adjustments and provide advice related to the practical implementation of our approach in the classroom.

4.1 Pedagogical Recommendation 1: Reorder the Competence Areas to Start with Branching

With our first and most fundamental recommendation, we suggest reconsidering the order in which the contents are introduced. Specifically, our approach starts with the competence in branching because understanding it enables a quick and high-level understanding of Git and then proceeds with committing and collaborating whose understanding benefits from a prior knowledge of branching. While most Git courses typically begin with versioning and then move on to collaboration, we suggest that a better order is to start with branching, followed by versioning and collaboration.

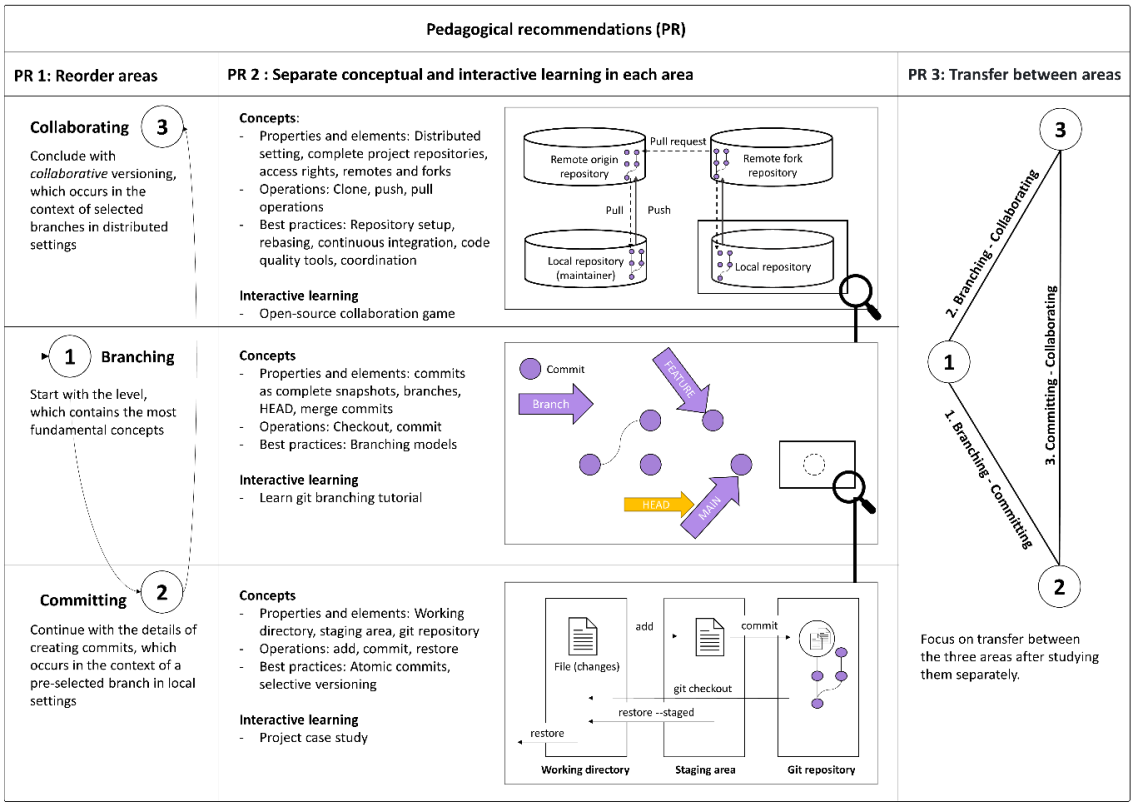

Figure 1. Overview of the Pedagogical Recommendations for Teaching Git

There are both practical and theoretical rationales to support this recommendation. First, the typical everyday workflow of Git commonly begins with checking the status of branches and selecting the appropriate one on which to work. By starting with the competence area of branching, students become accustomed to this real-world scenario. Second, committing in Git benefits from a deeper understanding of the underlying Git model. It is beneficial for students to have a clear understanding of branching before delving into committing or collaborating. This foundation will help them comprehend versioning operations like *checkout*, *revert*, and *reset*, which rely on knowledge of commits and the commit history within a branch. Similarly, synchronizing work with remote repositories and organizing contributions online requires a conceptual understanding of branches (see Figure 1). As a result, we recommend prioritizing branching as the initial topic taught, followed by committing and collaborating.

This recommendation is consistent with *schemata theory* (Anderson, 2018), a cognitive theory that emphasizes the importance of developing mental schemata and associating new information with existing schemata. The cognitive perspective of *schemata theory* offers an explanation of how knowledge is organized and stored in our minds. According to this theory, a schema is a mental framework or structure that enables us to organize and interpret information. Schemata are developed through experience and learning. As we encounter new information, we assimilate it into existing schemata or create new schemata to accommodate novel knowledge. In line with this theory, instructors have the discretion to teach content areas that have strong internal coherence and arrange them in an order that facilitates students' discovery of associations and understanding across these areas. With respect to Git instructions, starting with a high-level overview of branching, rather than diving into specific details, may allow students to form associations and better comprehend the Git model before adding more technical details. This approach enables them to build a solid foundation before delving into more intricate concepts.

4.2 Pedagogical Recommendation 2: Proceed From Conceptual to Interactive Learning

Sessions on branching, committing, and collaborating should start with focused explanations of the key concepts because understanding them requires concentrated effort. Afterwards, the concepts should be practiced interactively, as group-based learning sessions ensure that students can apply these concepts in practice.

Initially, instructors should select and explain the key concepts, enabling learners to focus on selected contents while eliminating errors and distractions related to their application. In line with the work of De Rosso and Jackson (2013), it is essential to carefully choose Git concepts, taking into account their relative complexity. In the initial phase, learners can thus concentrate on a limited set of fundamental concepts, aiming to isolate this learning experience from concepts associated with the other Git areas. The primary objective is to reduce the necessity of context-switching and thus minimizing time spent on unrelated tasks such as file changes and collaboration. In a nutshell, cognitive dependencies on other competency areas should be minimized, especially in the early stages of learning Git.

The first part of the recommendation aligns with *cognitive load theory* (Plass et al., 2010), which posits that the way information or tasks are presented can impact the level of extraneous cognitive load experienced by learners. In educational psychology, cognitive load theory (Sweller, 1988) examines how the cognitive load imposed on learners affects the learning process. The selection and separation of conceptual and interactive learning reduces cognitive load in the first phase. One crucial aspect of cognitive load theory is the concept of *split-attention effects* (Chandler & Sweller, 1992), which occur when learners must simultaneously process information from different sources or modalities, leading to cognitive overload and reduced comprehension. When learners are presented Git commands on static slides or on the command-line, they have to imagine how the corresponding Git graph evolves. This is a typical situation in which split attention can be expected to have negative effects on learning outcomes. Furthermore, research on the constraints of human memory sheds light on the limitations and capacities of memory systems (Baddeley & Hitch, 1974). Staying within these constraints is essential for designing effective instructional strategies that optimize learning outcomes. Taken together, theories of cognitive load and human memory constraints provide valuable insights into the cognitive processes involved in learning and inform instructional practices aimed at reducing cognitive load and enhancing learning efficiency.

After focusing on the conceptual foundations, students should be provided with opportunities to apply and test their conceptual understanding in practical settings. The instructional approach should transition to interactive and group-oriented learning sessions that progressively advance in complexity. As learners gain proficiency in the fundamentals, instructors can gradually introduce more intricate exercises. The complexity of the Git conceptual model, as outlined by De Rosso and Jackson (2013), aligns with this approach, starting with simpler tasks and gradually incorporating more challenging elements and leveraging on the advantages of collaborative group work.

This part of the recommendation is consistent with three pedagogical theories. First, *interactive learning theory* (Kolb, 2014) suggests that interactive learning tools can be particularly effective in computer science education. This theory assumes learning to be a continuous process involving four stages: concrete experience, reflective observation, abstract conceptualization, and active experimentation. The theory emphasizes the importance of actively engaging with the learning process and suggests that different learners may prefer different learning styles in the four stages of the learning cycle. This approach further emphasizes connecting practical experiences to existing knowledge. Group-based learning has emerged as a potent educational strategy for addressing challenging tasks, as evidenced by numerous studies in the field. As an example, Johnson and Johnson (2009) demonstrate the effectiveness of group-based learning in fostering successful educational outcomes. By emphasizing social interdependence—the reliance of individuals on one another to achieve common goals—cooperative learning approaches harness the collective potential of groups, thus underscoring the significance of group-based learning as a dynamic educational strategy suitable for addressing challenging tasks. By embracing social interdependence and collaborative learning principles, educators can create enriching learning

environments that empower students to thrive academically and socially.

Second, the *learning-by-doing-approach* (Reese, 2011) focuses on learning from direct experiences resulting from one's own actions. This approach focuses on experiential problem-based learning in real-world contexts, combined with student collaboration and reflection. Overall, the learning-by-doing approach emphasizes active engagement, practical application, and reflection as essential components of effective learning experiences. It highlights the importance of engaging in exercises of involvement, which lead to subconscious learning.

Third, the *socio-cultural learning theory* (Connolly et al., 2022) recognizes the value and meaning of everyone's unique perspective and promotes the social sharing of personal experiences. It incorporates Vygotsky's *zone of proximal development*, which suggests that learning with assistance is more effective and successful than learning alone (Chaiklin, 2003). In line with this theory, we observed that groups achieved progress at a higher rate in the practice sessions than did individuals working alone, because students helped each other solve errors.

Overall, Pedagogical Recommendation 2 underscores the importance of finding a balance between the focused introduction of concepts and their application in self-learning settings, as recommended by Vial and Negoita (2018). After focusing on understanding and applying contents within each competence area, learners should be well-prepared to solve transfer questions involving different areas.

4.3 Pedagogical Recommendation 3: Stimulate the Transfer of Knowledge Between the Areas of Competence

After the three areas are studied independently, the focus turns to the transfer of knowledge between these areas of competence, as cognitive connections between *committing*, *branching*, and *collaborating* are an essential part to acquiring a solid understanding of Git.

In this phase, acquired knowledge is used to solve more complex and realistic tasks that involve interconnections across different areas. The questions in Table 4 prioritize grasping the mechanics and reasoning behind Git usage and require students to transfer knowledge, beyond simply conveying factual knowledge about Git. By understanding the reasons and methodologies behind Git operations, learners gain the capacity to employ their expertise across various scenarios and manage more intricate version control situations. Mastering such interconnected settings enables learners to develop a deeper comprehension of Git's fundamental model. Including a set of transfer questions at the end is an effective measure to prepare leaners for the real-world challenges in software development and collaborative projects. These questions can also serve as an assessment tool to measure participants' understanding of the key concepts covered during the session, as well as their ability to connect these concepts with other areas.

Schemata theory supports this recommendation for two reasons. First, it highlights the role of activating prior knowledge, which challenging transfer questions effectively stimulate, strengthening existing schemata and making them more accessible for future use cases (Anderson, 2018). Second, it contends that schema construction relies on the addition of associations with existing knowledge, suggesting that associations across thematic areas may encourage the development of more complex and interconnected schemata (Anderson, 2018). As such, this theory suggests that transfer questions may stimulate students to recognize connections between Git areas, enable a deeper understanding, and improve their ability to apply this knowledge in novel situations.

Areas and Rationale	Example Questions
Branching – committing • Rationale: Start with the most recent commit and continue with commits in the history of the branch, given that the recent commits are consulted relatively often compared to analyzing and editing older commits.	• How do the different options to restore changes from the last commit (soft/mixed/hard reset, revert) differ? • How does switch/checkout/reset/revert affect the areas, i.e., the working directory and staging area? Why do we need a clean working directory? • How can we enter the 'detached HEAD mode' and for which purposes could it be useful? • How can older commits be edited and what is the effect of rewriting history" on the following commits?"
Branching – collaborating • Rationale: Proceed from single-branch synchronization to workflows in highly collaborative settings, which can involve forks, more remotes, and more complex branching models.	• What would you expect if you push a branch with a divergent history? • Why should 'rewriting history' be avoided when pushing shared branches? • How does 'Git pull –rebase' work and when is it useful? • How are local branches associated with remote branches? How are 'tracking branches' set up during push or pull? • How can changes from your local branch be contributed to a remote repository owned by another maintainer? If you continue your development on the branch, how can these changes be contributed?
Committing – collaborating • Rationale: Proceed from simple linear commits to merging strategies in collaborative settings, in which community conventions must be considered.	• How does GitHub's web-based functionality to edit and commit files work? Are the working directory and staging area available on GitHub? • What is the difference between merge, rebase, and squash options in pull requests? • In the context of merging or squashing pull-requests, what is the difference between committer and author?

Table 4. Examples of Transfer Questions

Our proposed model encourages educators to teach Git by breaking the material down into easily digestible topics before introducing brain teasers that involve combining concepts across areas (Eysenck & Keane, 2020). Breaking the learning process into smaller, easily manageable parts enables students to grasp fundamental principles and gain expertise in specific areas before facing intricate challenges (Goldstein & Vanhorn, 2008). After becoming proficient in the individual areas, students can then move on to solving brain teasers that incorporate multiple concepts. These challenges not only reinforce learning but also foster critical thinking and showcase the practical application of knowledge across different contexts and encourage reflective observation, as suggested by Kolb (2014).

5. PRACTICAL STRATEGIES

To effectively implement the pedagogical recommendations given above, we suggest three practical strategies, as outlined below.

5.1 Practical Strategy 1: Create Self-Contained Learning Environments by Minimizing Technical Dependencies Between Areas

While existing teaching resources provide some examples for use in self-contained practice sessions on committing, creating self-contained learning settings to practice branching and collaborating skills is more challenging. Applying Git operations related to these two areas often requires having preceding commits and related Git setup steps. Our learning materials allow students to directly manipulate branches (Git's directed acyclic graph) or collaborate through remote workflows supported by such hosting platforms as GitLab or GitHub.

To practice branching in a self-contained environment, we recommend the interactive *Learn Git Branching* tutorial, which provides immediate visual feedback and allows for focused practice without the need for extensive setup (as shown in the screenshot in Figure 2). In this way, learners can concentrate on understanding and applying the core concepts of branching and the Git graph without spending time on setting up local repositories, defining Git parameters, and committing changes for each version.

To practice collaboration in a self-contained environment, we developed the *Open-Source Collaboration Game* on GitHub (Wagner, 2024), which allows students to practice committing, branching, submitting pull requests, and merging online without the need for local setup or SSH connection. It further simplifies the workflow by eliminating the distinction between working directory and staging area and removes the need for *git add* operations. Overall, this strategy enables learners to efficiently use their learning time to apply the focal concepts without distractions related to setup and technical details.

5.2 Practical Strategy 2: Start by Illustrating Concepts *Dynamically*, and Have Students Practice It in *Small Groups* Afterwards

The second practical strategy proposes a two-pronged approach, consisting of a focused and dynamic illustration of the key concepts followed by practice sessions in small groups (illustrated in Figure 3). It begins with concept teaching, prioritizing the comprehension of fundamental models such as *branching* and the underlying directed acyclic graph (DAG). While static slides may offer a convenient way to present information, they often fall short in effectively teaching complex, dynamic concepts like Git. Corresponding split-attention effects were discussed as a rationale for Pedagogical Recommendation 2.

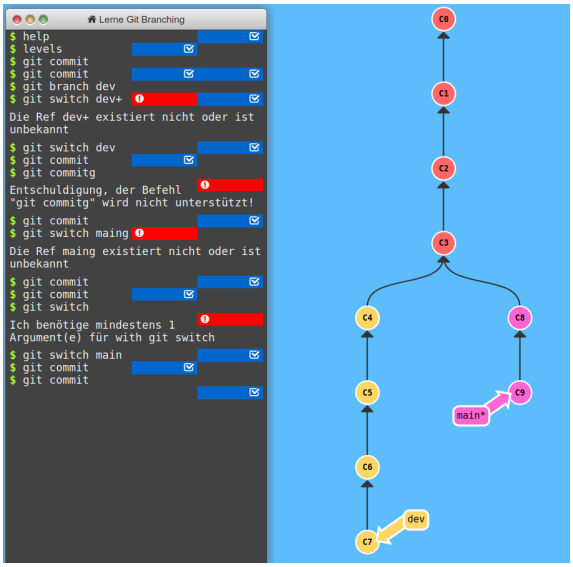

Figure 2. The Learn Git Branching Tutorial (Screenshot)

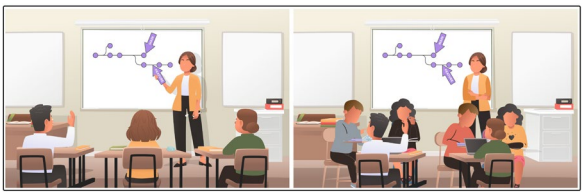

Figure 3. Start by Illustrating Concepts Dynamically, Followed by Interactive Application in Small Groups

By embracing interactive, hands-on learning approaches, educators can create more engaging, immersive learning experiences that promote deeper understanding of and proficiency with Git. Our teaching materials delve into the dynamic interplay between data structures and operations through dynamic live demonstrations, which appear to be more instructive than static slides. To further enrich this strategy, our proposed model incorporates practical sessions that gamify the learning process and utilize diverse groups and interactive settings like the *Open-Source Collaboration Game* (Wagner, 2024). These sessions allow students to assume different roles and engage actively with the material. Additionally, we cover Git conventions and sensitize students to poor practices, ensuring a comprehensive learning experience, as outlined by Eraslan et al. (2020). The resulting integration ensures that students progress steadily and acquire various concepts effectively. For instance, students should practice branching without the need for creating file changes or authenticating to establish remote connections. Similarly, collaboration techniques, such as pull requests, should be learned without

time constraints to support creating local commits, performing authentication, establishing connections, and pushing changes to remote repositories.

Ultimately, this strategy seeks to provide students with a well-rounded learning experience by combining theoretical understanding with hands-on practice, fostering collaboration and engagement, and preparing them for real-world scenarios.

5.3 Practical Strategy 3: Challenge Students to Think Across Areas Based on Hypothetical Scenarios and an Array of Transfer Questions

The development of transfer knowledge requires a setting in which students can explore an array of questions involving concepts drawn from different Git areas. This involves diverse and relatively unconnected problem sets that are hard to combine in a coherent case exercise. Against this background, our third practical strategy proposes the use of hypothetical scenarios and an array of transfer questions.

The ordering of Git areas in our proposed method suggests proceeding from transfer questions first related to *branching and committing* to those related to *branching and collaborating* and then those related to *committing and collaborating* (as shown in Figure 1). Table 4 provides example questions for each transfer block, along with a rationale for arranging the questions in the order shown. After providing students with a scenario and transfer question, the solution may involve intuition based on pre-existing knowledge of Git concepts, consultation of the official Git manuals, a Web search, or the testing of outcomes in an example repository. Given the challenging nature of these transfer questions, we provide students with a detailed solution that can be revisited after the session.

Overall, encouraging students to think in a more connected way enables them to gain a more holistic understanding of Git and its applications beyond software development. This approach allows them to address complex issues from a more comprehensive perspective and ultimately become more proficient in using Git.

6. STUDENT FEEDBACK AND EVIDENCE

We offered the Git tutorial three times and, after each session, we updated and extended the materials based on the feedback received. We drew inspiration from prior pedagogical work calling for holistic, 360-degree feedback approaches (Tee & Ahmed, 2014) and involved different stakeholders in assessing the teaching materials and providing constructive feedback. After the sessions, students were asked to provide feedback on their learning experience and rate their perceived learning outcomes (the Appendix contains the survey items). Teaching and research assistants observed the sessions, noting potential improvements and administering short surveys at the conclusion. We also consulted with colleagues to evaluate our approach, clarify presentation, and identify potential improvements.

Overall, student responses confirmed the appropriateness of the new approach, which was associated with improved learning outcomes. The new approach and presentation of contents were well-received, especially by students who did not have a technical background. In particular, students who had previously taken similar courses (following the traditional approach) indicated that the overall approach and illustrations made understanding the underlying Git model much easier. Moreover, students rated their proficiency with Git as high (3.7 out of 5 on average) after completing the session. They found it helpful to start with Git branching (4 out of 5 on average), with students who attended other Git courses agreeing that "*Compared to previous Git introductions, the materials are less technical, and more intuitive*" (student comment).

In addition, the materials were commended for clarity of illustration. The materials were found to be complete, providing a good overview of the main elements of Git (3.7 out of 5 on average). No contents were found to be missing (5 out of 5 on average), suggesting a comprehensive learning experience.

The tutorial raised the interest of the students, who indicated potential areas where the materials could be extended. Demands for additional practice materials, references (including those for Git commands and manuals), and discussion were noted. Regular discussions were considered helpful to address specific questions related to, for example, best practices in the context of remote collaboration and rebasing. With regard to the interactive *Learn Git Branching* tutorial, students appreciated additional examples. These cases were deemed particularly helpful because they allowed students to reproduce the commands that led to the example Git graph. Students also expressed a desire to continue their practice and improve their confidence with Git based on specific use case examples, such as open-source projects.

We also noted a few aspects that were associated with improved ease of learning. In particular, students expressed appreciation for the variety in modes of teaching delivery, including the dynamic illustration of concepts, highlighting that "*The usage of a whiteboard and moveable arrows is better than static slide*" (student comment).

The tutorial also featured a mix of conceptual and interactive vs. applied learning to make the material more engaging. Additionally, gamification and open-source collaboration were found to enhance the learning experience.

7. CONCLUDING REMARKS

With this teaching tip, we rethink how Git can be taught effectively by proposing a novel, macro-level ordering of contents. Our approach may appear surprising to instructors who traditionally teach Git with technical dependencies in mind. Specifically, we recommend starting with branching instead of committing, as this allows students to understand the most fundamental principles of the Git model before proceeding with commands that will be used less frequently and are more technical in nature. Evidence from our ongoing teaching activities and student feedback indicates that the approach is effective. Particularly encouraging was seeing the many cases in which students who did not have a technical background apply the key concepts of Git with confidence.

Our suggestions for teaching Git add a novel approach to IS pedagogy theory, as they challenge the traditional assumption that technical complexity requires educators to introduce contents in a particular order. Instead, we envision more effective educational strategies that are engaging and allow students to better absorb technical material. A potential implication is the possibility of expanding this approach to other areas within IS, including facets of DevOps, open-source development and rapid prototyping. By doing so, follow-up work may reinforce efforts to ground teaching of Git and other

software engineering topics in pedagogical theory to foster deeper understanding and better learning outcomes. Teaching resources in the IS discipline could even contribute to enriching educational materials and textbooks in other disciplines. In particular, such work could inform teaching efforts in social science disciplines in which students are increasingly expected to be familiar with technical topics such as Git, programming, and computational analyses.

To facilitate the practical application of our teaching recommendations, we provide complementary online resources and invite educators to draw on our work when teaching Git. As the ability to use Git competently is becoming a common requirement, it is time to adopt effective teaching approaches, which facilitate students' acquiring a deep understanding of Git and enable them to apply the basic commands or quickly identify appropriate options.

8. ACKNOWLEDGMENTS

We thank Julian Prester for his feedback on the paper, Rida Arain for her help with the illustrations, and May Graybeal for proofreading the paper.

9. ENDNOTES

Complementary online resources are available at https://digital-work-lab.github.io/rethink-git-teaching/

10. REFERENCES

Aasheim, C., Shropshire, J., Li, L., & Kadlec, C. (2012). Knowledge and Skill Requirements for Entry-Level IT Workers: A Longitudinal Study. *Journal of Information Systems Education*, 23(2), 193-204.

Anderson, R. C. (2018). Role of the Reader's Schema in Comprehension, Learning, and Memory. In *Theoretical Models and Processes of Literacy* (pp. 136-145). Routledge. https://doi.org/10.4324/9781315110592-9

Baddeley, A. D., & Hitch, G. J. (1974). Working Memory (Vol. 8). New York: GA Bower (Ed.), *Recent Advances in Learning and Motivation*. https://doi.org/10.1016/S0079-7421(08)60452-1

Bou Ghantous, G., & Gill, A. (2017). DevOps: Concepts, Practices, Tools, Benefits and Challenges. *Proceedings of the Pacific Asia Conference on Information Systems*.

Brooks, N. G., Greer, T. H., & Morris, S. A. (2018). Information Systems Security Job Advertisement Analysis: Skills Review and Implications for Information Systems Curriculum. *Journal of Education for Business*, 93(5), 213-221. https://doi.org/10.1080/08832323.2018.1446893

Chaiklin, S. (2003). The Zone of Proximal Development in Vygotsky's Analysis of Learning and Instruction. In A. Kozulin, B. Gindis, V. S. Ageyev, & S. M. E. Miller (Eds.), *Vygotsky's Educational Theory in Cultural Context* (pp. 39-64). Cambridge University Press. https://doi.org/10.1017/CBO9780511840975.004

Chandler, P., & Sweller, J. (1992). The Split-Attention Effect as a Factor in the Design of Instruction. *British Journal of Educational Psychology*, 62(2), 233-246. https://doi.org/10.1111/j.2044-8279.1992.tb01017.x

Connolly, A. J., Mutchler, L. A., & Rush, D. E. (2022). Teaching Tip: Socio-Cultural Learning to Increase Student Engagement in Introduction to MIS. *Journal of Information Systems Education*, 33(2), 113-126.

Cummings, J., & Janicki, T. N. (2020). What Skills Do Students Need? A Multi-Year Study of IT/IS Knowledge and Skills in Demand by Employers. *Journal of Information Systems Education*, 31(3), 208-217.

De Rosso, S. P., & Jackson, D. (2013). What's Wrong With Git? A Conceptual Design Analysis. *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, 37-52. https://doi.org/10.1145/2509578.2509584

DeSanto, D., Karnes, V., Flouton, M., Farris, J., & Benson, H. (2023). Product Roles. GitLab Handbook. https://handbook.gitlab.com/job-families/product/

Deshpande, A., Sharp, H., Barroca, L., & Gregory, P. (2016). Remote Working and Collaboration in Agile Teams. *Proceedings of the International Conference on Information Systems.*

Dohmke, T. (2023). 100 Million Developers and Counting. https://github.blog/2023-01-25-100-million-developers-and-counting/

Eraslan, S., Ríos, J. C. C., Kopec-Harding, K., Embury, S. M., Jay, C., Page, C., & Haines, R. (2020). Errors and Poor Practices of Software Engineering Students in Using Git. *Proceedings of the Conference on Computing Education Practice*, 1-4. https://doi.org/10.1145/3372356.3372364

Eysenck, M. W., & Keane, M. T. (2020). *Cognitive Psychology: A Student's Handbook.* Psychology Press. https://doi.org/10.4324/9781351058513

Goldstein, E. B., & Vanhorn, D. (2008). *Cognitive Psychology: Connecting Mind, Research, and Everyday Experience* (Vol. 59). Thomson Wadsworth Belmont, CA.

Haaranen, L., & Lehtinen, T. (2015). Teaching Git on the Side: Version Control System as a Course Platform. *Proceedings of the ACM Conference on Innovation and Technology in Computer Science Education* (pp. 87-92). https://doi.org/10.1145/2729094.2742608

Hassan, N., Rivard, S., Schultze, U., & Willcocks, L. (2023). Products of Theorizing—Towards Native Theories of Emerging Information Technologies. *Journal of Information Technology*, 38(4), 372-381. https://doi.org/10.1177/02683962231217348

Isomöttönen, V., & Cochez, M. (2014). Challenges and Confusions in Learning Version Control With Git. *Proceedings of the International Conference on Information and Communication Technologies in Education, Research, and Industrial Applications* (pp. 178-193). https://doi.org/10.1007/978-3-319-13206-8_9

Jabrayilzade, E., Uyanik, F. S., Sülün, E., & Tüzün, E. (2022). An Interactive Approach to Teaching Git Version Control System. *Proceedings of the Hawaii International Conference on System Sciences*. https://doi.org/10.24251/HICSS.2022.112

Johnson, D. W., & Johnson, R. T. (2009). An Educational Psychology Success Story: Social Interdependence Theory and Cooperative Learning. *Educational Researcher*, 38(5), 365-379. https://doi.org/10.3102/0013189X09339057

Kolb, D. A. (2014). *Experiential Learning: Experience as the Source of Learning and Development*. FT Press.

Latinovic, M., & Pammer-Schindler, V. (2021). Automation and Artificial Intelligence in Software Engineering: Experiences, Challenges, and Opportunities. *Proceedings*

of the Hawaii International Conference on System Sciences (pp. 146-155). https://doi.org/10.24251/HICSS.2021.017

Lippa, D. A. (2016). Get Out of Git Hell: Preventing Common Pitfalls of Git. *Proceedings of the International Workshop on Release Engineering* (p. 22). https://doi.org/10.1145/2993274.3011284

Microsoft News Center. (2018). https://news.microsoft.com/2018/06/04/microsoft-to-acquire-github-for-7-5-billion/

Pathak, A. (2020). Introduction to Git for Beginners. XRDS, 26(4), 54-59. https://doi.org/10.1145/3398459

Plass, J. L., Moreno, R., & Brünken, R. (2010). *Cognitive Load Theory*. Cambridge University Press. https://doi.org/10.1017/CBO9780511844744

Reese, H. W. (2011). The Learning-by-Doing Principle. *Behavioral Development Bulletin*, 17(1), 1-19. https://doi.org/10.1037/H0100597

Smits, M., & Mogos, S. (2013). The Impact of Social Media on Business Performance. *Proceedings of the 21st European Conference on Information Systems*.

Sweller, J. (1988). Cognitive Load During Problem Solving: Effects on Learning. *Cognitive Science*, 12(2), 257-285. https://doi.org/10.1207/s15516709cog1202_4

Tafliovich, A., Estrada, F., & Caswell, T. (2019). Teaching Software Engineering With Free Open Source Software Development: An Experience Report. *Proceedings of the Hawaii International Conference on System Sciences*. https://doi.org/10.24251/HICSS.2019.931

Tee, D. D., & Ahmed, P. K. (2014). 360 Degree Feedback: An Integrative Framework for Learning and Assessment. *Teaching in Higher Education*, 19(6), 579-591. https://doi.org/10.1080/13562517.2014.901961

Thummadi, B., Khapre, V. D., & Ocker, R. (2017). Unpacking Agile Enterprise Architecture Innovation Work Practices: A Qualitative Case Study of a Railroad Company. *Proceedings of the Americas Conference on Information Systems*. https://doi.org/10.5465/AMBPP.2017.16844abstract

Vial, G., & Negoita, B. (2018). Teaching Programming to Non-Programmers—The Case of Python and Jupyter Notebooks. *Proceedings of the International Conference on Information Systems.*

Wagner, G., & Prester, J. (2024). CoLRev: An open-Source Environment for Collaborative Reviews (Version 0.12.3) https://doi.org/10.5281/zenodo.11668338

Wagner, G. (2024). The Open-Source Collaboration Game (0.1.0). Zenodo. https://doi.org/10.5281/zenodo.13323591

Westby, E. J. H. (2015). *Git for Teams: A User-Centered Approach to Creating Efficient Workflows in Git*. O'Reilly Media, Inc.

AUTHOR BIOGRAPHIES

Gerit Wagner is an assistant professor of digital work at Otto-Friedrich Universität Bamberg. His teaching activities cover Python programming, Git, and open-source software development, with many of his students contributing to the CoLRev package for literature reviews. In research, he focuses on the future of work, at the intersection of digital technologies, knowledge work, and crowd work. In addition, he contributes to literature review methods and tool design. His publications have appeared in journals like the *Journal of Strategic Information Systems*, *Information Systems Journal*, and *Journal of Information Technology*.

Laureen Thurner studies information systems at Otto-Friedrich Universität Bamberg and serves as a student assistant at the Professorship for Information Systems (Digital Work). She has industry experience in IT security consulting in Switzerland, where she worked on enhancing cybersecurity measures for various clients. In her role as a student assistant, Laureen focuses on open-source technologies, Git, and employee handbooks in the tech sector, as well as security and data protection. She contributes to creating and maintaining internal documentation and supporting the integration of secure and efficient practices within digital work environments.

APPENDIX

Survey Items

1. Did you attend any Git courses before? (Yes/No, if yes, where?)
2. How does the Git introduction compare to previous courses in terms of structure and difficulty?
3. Rate your proficiency with Git (after the session) - Likert scale, ranging from 1 (Really good) to 5 (None)
4. It was helpful to start with branching - Likert scale, ranging from 1 (Strongly agree) to 5 (Strongly disagree)
5. I have a good overview of the main elements of Git - Likert scale, ranging from 1 (Strongly agree) to 5 (Strongly disagree)
6. I am confident in using Git in projects - Likert scale, ranging from 1 (Strongly agree) to 5 (Strongly disagree)
7. How was the length of this Git introduction?
    1) Too short
    2) Just right
    3) Too long

8. Were there any contents missing? If yes, please elaborate:

	No	Yes	Which contents?
Branching			
Committing			
Collaborating			

9. Comment section (for ideas or changes)