Evaluating Human-AI Partnership for LLM-based Code Migration

Ishaani M*
Amazon Web Services
Seattle, WA, USA
ishaani@amazon.com

Behrooz Omidvar-Tehrani
AWS AI Labs
Santa Clara, CA, USA
omidvart@amazon.com

Anmol Anubhai
Amazon Web Services
Seattle, WA, USA
anubhaia@amazon.com

ABSTRACT

The potential of Generative AI, especially Large Language Models (LLMs), to transform software development is remarkable. In this paper, we focus on one area in software development called "code migration". We define code migration as the process of transitioning the language version of a code repository by converting both the source code and its dependencies. Carefully designing an effective human-AI partnership is essential for boosting developer productivity and faster migrations when performing code migrations. Though human-AI partnerships have been generally explored in the literature, their application to code migrations remains largely unexamined. In this work, we leverage an LLM-based code migration tool called Amazon Q Code Transformation to conduct semi-structured interviews with 11 participants undertaking code migrations. We discuss human's role in the human-AI partnership (human as a director and a reviewer) and define a trust framework based on various model outcomes to earn trust with LLMs. The guidelines presented in this paper offer a vital starting point for designing human-AI partnerships that effectively augment and complement human capabilities in software development with Generative AI.

CCS CONCEPTS

• **Human-centered computing** → **Collaborative interaction**; **User studies**; • **Software and its engineering** → **Designing software**.

KEYWORDS

Application Modernization, Code Migration, Human-AI Partnership, Human-in-the-Loop Techniques, Trust Framework

ACM Reference Format:
Ishaani M, Behrooz Omidvar-Tehrani, and Anmol Anubhai. 2024. Evaluating Human-AI Partnership for LLM-based Code Migration. In *Extended Abstracts of the CHI Conference on Human Factors in Computing Systems (CHI EA '24), May 11–16, 2024, Honolulu, HI, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3613905.3650896

*Ishaani M and Behrooz Omidvar-Tehrani contributed equally to this research.

CHI EA '24, May 11–16, 2024, Honolulu, HI, USA

ACM ISBN 979-8-4007-0331-7/24/05.
https://doi.org/10.1145/3613905.3650896

1 INTRODUCTION

Generative AI, especially Large Language Models (LLMs), are revolutionizing software development across various realms of application modernization, such as code refactoring [26], documentation generation [10], bug detection [1], and fault localization [15]. This technology is not only transforming the way developers approach complex coding challenges, but also paving the way for a future of more agile, robust, and innovative software development practices.

In this paper, we focus on *code migration*, an essential application modernization technique that automates the process of transitioning the language version of a code repository by converting both the source code and its dependencies. An example of code migration is to migrate a code repository developed in Java 8 specification to be compatible and functional with Java 17. Many organizations have critical business systems running on outdated codebases like Java 8. Migrating these legacy codebases to modern versions like Java 17 is essential to leverage new language features, avoid exposure to security vulnerabilities, and avoid maintenance issues from outdated libraries and frameworks [1]. However, manually upgrading millions of lines of legacy code is tedious, risky, time consuming and infeasible beyond a certain point. Generative AI and LLMs in particular can automate much of this code migration process which greatly accelerates modernization of legacy systems and allows faster innovation.

While Generative AI shows promise for automating code migration, there is still uncertainty around how developers will collaborate with such automated tools: *"how much control do developers want on the output?"* and *"do they trust the output produced?"* In [34], Weiss et al. come close to discuss such a collaboration but they focus on translating one code file at a time. This is limiting since code migrations typically happen on an entire codebase spanning thousands of files, with interlinked dependencies between them. Our work expands on their work and focuses on understanding human-AI partnership when AI has been given the task of making code changes on entire codebase at a time.

Our research consists of an in-depth study focused on developer interaction when utilizing LLMs for application-level code migration. The study involves observing users as they interact with the output of the AI-generated code modifications, and then evaluate cases of both successful and unsuccessful migrations. We observe user workflows, trust levels, and collaboration strategies, when presented with AI-generated code modifications across multiple files in a complete codebase. This study leverages Amazon Q Code Transformation, an AWS Generative AI service for application modernization including code migration, to answer the following two research questions:

[1] *The Importance of Modernizing Legacy Code: https://leeorengel.com/modernizing-legacy-code/*

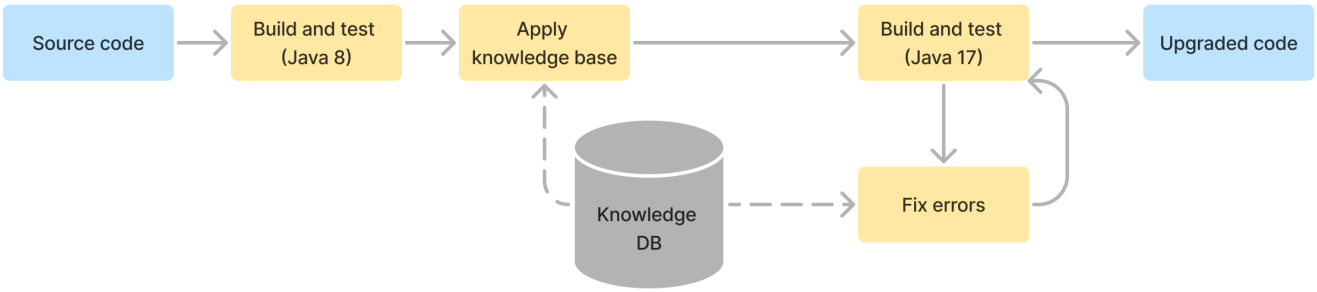

Figure 1: Overall structure of Amazon Q Code Transformation.

RQ1: What role do developers expect LLMs to play in code migration? What is the ideal level of human involvement and oversight at different stages of the LLM action?

RQ2: How can developers build trust with the outputs of the AI-generated code modifications for code migration?

Our findings provide HCI design implications for future AI-based application modernization tools that promote seamless human-AI collaboration. We aim to guide user experience decisions that enable transparent and productive human-AI collaboration for complex code migrations. By understanding workflows and challenges around trusting AI suggestions made on complete code repositories, we expand the HCI community's knowledge for designing guidelines for the next generation of AI-based application modernization tools. In summary, our key findings are as follows:

Finding 1: Developers desire involvement not only during code review, but also in directing the AI to produce the desired output using their own knowledge and the AI's prior experience (elaborating on RQ1).

Finding 2: When performing code reviews, developers hold the AI-generated code to the same standards as code written by human teammates (elaborating on RQ1).

Finding 3: Varying outcomes from different LLM models necessitate tailored approaches to help developers establish trust in the AI output (elaborating on RQ2).

The remainder of this paper is structured as follows. Section 2 reviews relevant literature on human-AI partnership specifically in the context of code migration. Section 3 introduces the task of code migration for application modernization and presents Amazon Q Code Transformation, the Generative AI tool used for code migrations in our experiments. Our study design is detailed in Section 4. Section 5 presents the results of our study, including key observations and lessons learned. Last, we conclude in Section 6.

2 RELATED WORK

Recent years have seen revolutionary advances in using AI techniques like transformer models for application modernization and code migration, starting with the pioneering work by Devlin et al. [8] on understanding code semantics. Critical contributions were made by Chen et al. [4] demonstrating GPT-3 for code tasks, Tufano et al. [30] on neural approaches for code refactoring and transformation, and Lachaux et al. [18] on unsupervised learning for code translation. However, despite this progress, significant challenges remain, as shown by the low accuracy (4.8%) of resolving GitHub issues in the SWE-Bench benchmark [14] using Claude V2. This complexity of automated code modernization illustrates the need for human-AI collaboration to improve outcomes. AI and humans working together collaboratively can be more effective than either alone. As Weiss et al. [34] posit, the question is how to design effective interactions for humans and AI to work together.

A body of research focuses on designing effective human-AI interactions by examining the core aspects of the partnership. This includes understanding task delegation to AI systems [25], establishing guidelines for designing interactions with AI [2, 6, 33, 36], and cultivating appropriate levels of trust [11, 17, 20, 27]. However, this research often lacks domain specificity. Prior studies show practitioners struggle to directly apply such general frameworks and recommendations [7, 22, 29]. However, LLMs are most effective when they are adapted for specific domain [19, 24] and thus the interactions need to adapt to those domains for an effective human-AI partnership. Therefore, AI practitioners need the next body of work to define the right experiences for their domain.

Another distinct area of research focuses on human-AI partnerships in specific use cases. These partnerships are often categorized into three types: (*i*) *co-creation*, where humans and AI work on the same canvas generating an output together [3, 12, 16, 21, 23], (*ii*) *complementary work*, where humans and AI produce similar outputs in parallel [13, 31], and (*iii*) *hand-offs*, where humans assign unique tasks to AI (e.g., translating single file code) [32, 34]. Our research examines the human-AI partnership in the context of code migration spanning across multiple files, an example of the hand-off model. Specifically, we aim to understand (*i*) the roles of humans versus AI when an AI system is tasked with high-level codebase modifications, and (*ii*) how users can build trust in the outcomes generated by AI.

3 AMAZON Q CODE TRANSFORMATION FOR CODE MIGRATION

Amazon Q Code Transformation is a new Generative AI tool for application modernization and particularly code migration [28]. Amazon Q Code Transformation automates common language upgrade tasks like updating code, conducting unit tests, and verifying

```
<dependency>
  <groupId>com.fasterxml.jackson.core</groupId>
  <artifactId>jackson-core</artifactId>
  <version>2.9.4</version>
  <version>2.12.5</version>
</dependency>
```

```
Error: [...] package org.junit.runner does not exist.
Fix: Add dependency junit.

<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <scope>test</scope>
</dependency>
```

Figure 2: Example of AI-led code modifications by Amazon Q Code Transformation.

deployment readiness. It saves days' worth of the undifferentiated work involved in moving off older language versions. While there exist already a few commercially available AI-assisted code generation tools (GitHub Copilot and ChatGPT, to name a few [35]), at the time of writing, Amazon Q Code Transformation represents one of the existing technologies for performing LLM-based code migration. The overall architecture of Amazon Q Code Transformation is illustrated in Figure 1.

Amazon Q Code Transformation input. The starting point for Amazon Q Code Transformation is to upload the code repository into a secure build environment in the cloud. Amazon Q Code Transformation then analyzes the code and performs code modifications and dependency upgrades using a knowledge base of necessary code changes. An example is shown in Figure 2 left where Amazon Q Code Transformation modifies the version of a dependency called `jackson-core` for improved compatibility and security within Java 17 framework. Then the modified code will be built and Amazon Q Code Transformation will iterate over the errors to improve the code, akin to the process that a human developer follows. An example is shown in Figure 2 right where building the code repository results in an error about the dependency `junit` (dependency for simplifying the process of writing and running tests in Java applications), and Amazon Q Code Transformation suggests to add this dependency to the repository.

Amazon Q Code Transformation output. Amazon Q Code Transformation's output is a diff file highlighting code changes in each file. An example of a code diff in JetBrains IntelliJ IDE[2] is shown in Figure 3 below where the highlighted code on the right pane shows the modified code. Here we observe that some annotations for `WebMvcTest` are modified to be compatible with Java 17, given that `secure` parameter is not supported anymore in newer versions. Also the keyword `public` is removed for a method to provide more control over inheritance using sealed classes. The developer can then decide to approve or disapprove each single code change before being pushed to the code repository.

In our experiments, we leverage Amazon Q Code Transformation for code migration from Java 8 to Java 17. On Amazon's reported internal benchmark, it is shown that Amazon Q Code Transformation upgraded 1000 Java 8 applications to Java 17 in two days, with each upgrade taking 10 minutes on average, and the longest upgrade taking less than one hour compared to the 2+ days these upgrades previously required [28].

4 STUDY DESIGN

To understand how developers interact with an LLM-based code migration tool, we designed two experiments, `E1` and `E2`, and conducted a between-subject study with 11 participants. In both experiments, the input is a code repository developed in Java 8. In `E1`, the participants were provided with a successful migration scenario in which the output was built successfully using Java 17. In `E2`, an unsuccessful migration example was provided, where code changes made by the AI were unable to successfully build the application on Java 17. The experiments were designed to address RQ1: identifying the role of LLMs within developers' expectations in the migration process. They also provide direct insight into RQ2 by evaluating how developers interact with the LLM output, assessing their interaction patterns whether the output is successful or unsuccessful. In both experiments, participants were asked to review the code generated by Amazon Q Code Transformation in the form of code diffs. The experiments were conducted on open source Java code repositories so as to control for familiarity of participants with the codebase used for migration. We sampled Java code repositories for the study based on a benchmark of 500 open source code repositories collected from Github[3]. Each repository in our benchmark manages its dependencies with Maven[4] and includes a commit ID that successfully builds in Java 8.

4.1 Participants of the Study

We recruited 11 software developer participants by circulating a screening questionnaire. A necessary condition to participate in our study was to have expertise in Java. The participants had varying levels of experience: four had over 15 years, three had 5-10 years, and four had less than 5 years. Moreover, 8 participants had previous Java migration experience and 3 did not. All participants received a digital swag as compensation for their time dedication. As part of recruitment, we required participants to have IntelliJ, a commonly used Java IDE, installed prior to the study sessions to ensure a consistent environment.

[2] *https://www.jetbrains.com/idea.*

[3] *GitHub is a web-based platform which provides hosting for software development and version control using Git: https://github.com.*

[4] *Apache Maven is a software project management and comprehension tool for Java projects: https://maven.apache.org.*

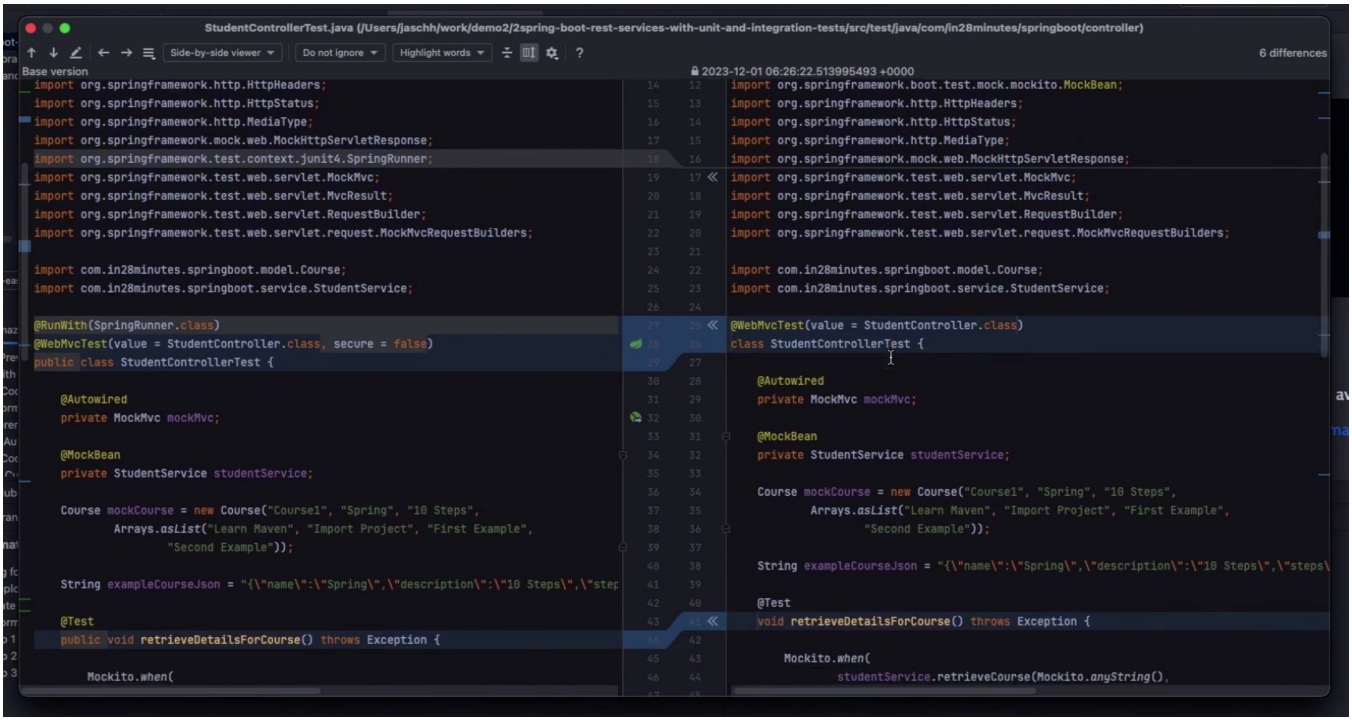

Figure 3: Code diffs in Amazon Q Code Transformation.

4.2 Protocol of the Study

Each participant engaged in a 90-minute remote screen-sharing session[5] with one or two co-authors of this paper, interacting with Amazon Q Code Transformation within the IntelliJ IDE. Appendix A presents the welcome script and sample steps that we provided to the participants. The objective defined for the participant was to review the AI-code modifications and ensure that the migration to Java 17 was successful. The definition of "successful migration" was kept vague on purpose to understand user expectations. Participants were allowed to use internet for help. We recorded the audio and screen captured user interactions with the consent of each participant. Throughout the session, participants were asked to talk out loud so as to capture the reasons behind their actions captured through screen record.

Each participant was given one Java code repository with two representative commit IDs: `C1` and `C2`. `C1` is the commit with build success in Java 8. `C2` is a commit made by Amazon Q Code Transformation which contains a version of the code migrated to Java 17. In the commit tree, `C2` is a descendant of `C1`. Also note that `C1` always provides participants with successfully building code, while `C2` contains code that either builds successfully (in the experiment `E1`) or fails to build (in `E2`). Participants were assigned to each experiment at random.

Each session followed a strict timing protocol. For the first 20 minutes, participants familiarized themselves with the tooling in IntelliJ and interacted with the version control system and the source code in Java 8. They then spent 20-40 minutes reviewing, debugging, and verifying the code by comparing the commits `C1` and `C2` and analyzing AI-led code modifications. A document outlining Amazon Q Code Transformation's changes was provided as reference (see Appendix A). The last 20 minutes involved discussion about the participant's experience and specific actions taken during the session. Participants could abandon the review session at any point if desired.

To analyze the results, the first author of this paper performed open-coding on participants' responses in a post-interview process and identified themes. The first author then discussed with the co-authors to refine these themes over multiple sessions. These themes are presented in Section 5 to explain the results of our research in a granular way.

[5] *The participant and interviewer utilized a video conferencing platform. During the session, the participant shared their screen while engaging with the output.*

5 RESULTS

In this section, we discuss elements of a successful human-AI partnership in code migration obtained from our study with 11 software development engineers. We discuss themes of human-in-the-loop techniques for code migrations, including enabling users to refine AI-led code updates and designing guidelines to earn trust with LLMs. To conduct a comprehensive analysis of both successful and unsuccessful cases, we integrate the outcomes of `E1` and `E2`. We refer to participant quotes using the labels `P1` through `P11`.

5.1 Roles of Human in Human-AI Partnership

Our study showed that developers consider AI to be a teammate and hold the code produced to the same standards as that of a teammate. This analogy effectively illustrates users' mental models and expectations of Generative AI code migration systems. Users expect Generative AI code migration systems to understand historical decisions, team constraints, and context – much like they would expect of a human teammate – and to apply team preferences when making coding decisions. Similarly, when developers review the code of their teammates, they expect a certain level of rigor and thought put into it. However, they do understand that their teammates may require some hand holding in the beginning. Developers are willing to provide this handholding to teammates, with the unspoken expectation that these teammates will learn over time and improve. When designing human-AI partnership experiences for code migration, considering this mental model could help define interactions that incentivize engagement with AI-generated output.

Through our research, we discovered two salient roles of humans in the human-AI partnership at different stages of the migration: (*i*) human as director and (*ii*) human as reviewer. When humans wear the hat of a director, they can help guide the AI in the beginning of code migration to generate the right output. Toward the end, humans can assume the role of a reviewer, reviewing, editing, and correcting the AI-generated output.

Human as director of AI to produce right output. Humans don't come into the process of code migration with a blank slate. The tool should allow developers to bring their existing knowledge and augment the LLM to produce the right output. Developers rely on their previous knowledge to make decisions on approving or disapproving code, and fixing build errors. For instance, P1 said the following.

> P1: *"Okay I have seen this before during my previous upgrade so which step is going wrong?"*

Humans would benefit from having more control over AI-generated output. This could be achieved by allowing humans to set preferences to guide the AI. In our study sessions, we observed divided preferences among participants regarding the level of upgrades. Some wanted to do only minimal updates to get their systems working on the latest versions (e.g., P10). Others expressed wanting to get "all the goodness" of new releases by getting more substantial upgrades to gain full access to new features (e.g., P8). This indicates a need for the AI to adapt to user preferences and incorporate those preferences while generating code modifications. Accounting for developer preferences also increases the likelihood that programmers will accept code changes proposed by the AI system.

> P10: *"I'd wanna make as minimal of a change as possible. I don't generally just like try to upgrade to the latest, especially in a case like this where I just wanna get a minimal build working."*

> P8: *"I know between Java 8 and Java 17, there's a lot of like interface differences, they added some functions here and some optimizations there. It would be great to see those implemented by the tool."*

However, simply offering users the ability to utilize their prior knowledge may be insufficient, as users cannot reasonably be expected to recall every detail that could be relevant in the current context. For instance as seen in P1's quote above, the participant had seen an error before, it was hard for them to recall how to fix the issue and they faced difficulty fixing the error. The Generative AI code migration system must consider how it can leverage techniques related to "recognition rather than recall" [9] to help users fix errors they have previously seen.

Another way in which AI can produce the right output is to learn from past upgrade attempts which already have human input. This could be achieved by maintaining history and context of how the upgrades were performed earlier or allowing users to specify their experiences and learnings to the AI.

Human as reviewer of code generated by AI. Four participants (P3, P4, P8, and P9) indicated that they viewed themselves as reviewers of the code generated by AI.

> P9: *"Initially I expect AI to generate code at the same level as a junior developer or someone who maybe new to the team."*

> P3: *"We view an AI upgrade bot as a member of the group that can provide feedback to the group based on iterations or versions of the group's work."*

These comparisons provide insights into two key user expectations for AI: (*i*) AI will make mistakes, and (*ii*) AI will learn. The expectation that AI may be inaccurate presents an opportunity to design human-AI partnership systems for code migration, as inaccuracies can act as springboards for complementary human input that allows the AI to learn and users to gain confidence in the system [5]. Further, just as code reviews help junior developers improve, constructive user critiques of the AI system enable it to better align with expectations and continuously amend its understanding like a programmer assimilating feedback. Both these user expectations work in favor of designers of human-AI partnership systems as AI systems rely on feedback from the field to become better. Rather than obscuring imperfections, designers of human-AI partnership systems should transparently reveal limitations in the AI's output that align with user expectations for appropriate interaction.

5.2 Approaches for Earning Developers' Trust in AI Capabilities

While AI promises improved efficiency, maintaining human oversight and due diligence remains vital. Enabling users to comprehend an AI's reasoning and troubleshoot errors would likely improve trust and promote more productive human-AI collaboration.

Through our research, we discovered two different dimensions of establishing trust toward the AI-generated code modification. First, we discuss a trust framework which defines design goals to establish trust in various outputs of the system. Second, we discuss robust testing as another important indicator to establish trust.

5.2.1 Trusting Various Model Outcomes. Users didn't feel comfortable accepting the code when a Java dependency was modified without verifying that it was the right version. We observed that verifying the correct version for a dependency upgrade is not

straightforward, as users must conduct multiple web searches to find the answer.

> P9: *"I feel like I am being gaslighted here. The first article says one version and second article says another version. Which one is it?"*

This participant, like 5 others, gave up trying to review the code generated by AI and started working on the upgrade from scratch. Interestingly, later in the session, P9 came to the conclusion that the code suggested by AI (even though incomplete) was actually correct. We also observed another participant P7 who received a hallucinated output but trusted the output for the whole session, trying to debug the errors they received but to no avail. P7 didn't question the output till was probed by the interviewer on what made them feel so confident about the output produced was correct.

These various experiences of the participants form the crux of our trust framework which defines design goals to establish trust in various outputs of the system (hallucinated output, correct but incomplete output, and correct output) so as to reduce time and user frustration to validate AI output.

Outcome 1: Model hallucinates and generates wrong output. Programmers understand the technical code complexities so they understand that the model may hallucinate. However, unmanaged hallucinated results can lead to complete abandonment of the AI system. What's further concerning is that in the case of failure, a user may end up diagnosing failure to compilation error instead of hallucinated code, as in the case of P7. The goal of human-AI partnership systems should be to help users understand that the model hallucinated so that users don't spend time on the wrong thing. The AI system must allow for easy debugging so that users don't end up spending a lot of time interacting with model hallucinations and get to the conclusion quickly.

> P11: *"Since it did not work the first time, so that would hamper on the trust a little bit. I think I would be less inclined to use it again since it didn't get it right. I'm still having to kind of debug manually, um, without, without much guidance."*

Further research should be done here to help users reach the conclusion about model hallucination quickly so that users can spend their time on the right thing.

Outcome 2: Model provides correct but incomplete answers. Use cases such as application modernization require code changes in multiple places. An AI system may only be partially successful in some cases. Those scenarios are still useful to developers as they reduce the code modernization time. However, since users have no visibility into the way the AI works, users can end up creating wrong mental model of the system, further reducing trust. By exposing the reasoning behind why the AI agent recommended a particular code change, providing references through citations and web links, AI practitioners can reduce user frustration and feeling of "bot didn't do anything to help me".

> P2: *"If it's an auto-complete style AI, most of them don't have a deep understanding necessarily. I can certainly understand how it would conclude that the solution is to fix individual dependencies. Much as many developers might, uh, have the same intent it's not fixing it the right way."*

It made P2 conclude that the AI system doesn't have nuanced understanding of complexities of application modernization that may not be necessarily true.

Outcome 3: Model provides correct-looking output with unintended consequences. A surprising outcome that we didn't expect came about to be when a participant mentioned how the code may successfully compile and run tests successfully but may end up introducing insecure versions or vulnerabilities. Security vulnerabilities should be avoided at all costs in automatically generated code because they can enable devastating real-world harm like data theft and service disruption. AI practitioners must make sure that their code suggestions do not introduce any security vulnerabilities and give end users reassurance by providing list of security checks that the suggested code passed.

> P8: *"If there is like an insecure version, like a version that has like specific guidance - It has like these flaws, use another one. I think that's, a good thing to be aware of because it could potentially then be introducing bugs or vulnerabilities into code that didn't have them before."*

Outcome 4: Model produced the correct output. Participants mentioned that even if the the model produced correct output, they would double check everything. The goal here should be to reduce the time it takes for the human to validate the output. Presenting detailed analysis in a clear, understandable format is critical for enabling humans to efficiently evaluate the accuracy of AI output.

> P9: *"I don't know if any AI, I would find extremely trustworthy. I am still going to double check everything. I don't expect it to be malicious, it gave me a valid dependency update but I am still going to double check every everything."*

Presenting detailed analysis in a clear, understandable format is critical for enabling humans to efficiently evaluate the accuracy of AI output.

5.2.2 Trusting Output through Tests. Participants expressed initial skepticism about the efficacy of the Generative AI code migration tool, noting that additional validation is required to confirm full functionality without runtime errors. A tool that provides a compilable code but not a successful test suite reduces the trust users have in the output. For example, participant P9 and P11 mentioned the following.

> P11: *"Organizations kind of have a rule that they're not willing to allow tests to fail as, and in fact, a lot of the continuous integration development tools will fail if the tests fail. Passing tests is important. If the tool gave me a compilable code and said the tests don't pass then I am not going to be super happy with the results."*

> P9: *"If the system didn't show me how it performed on the tests then it is not very useful. For all I know, it could turn off all the tests."*

P1 expressed how running tests and fixing the code should be simple for the tool to do.

> P1: *"I would expect AI to run through test cases and ensure there are no errors - it made sure that the code compiled but not that it was error free/correct. I would expect AI to run the tests and then get this error message followed by doing a simple search on this test message. It should also shape and apply a fix. The fix should be simple."*

Participant P5 went on to say that unit tests are not enough, they want more types of testing to verify end to end operational integrity.

> P5: *"I understood that the code compiled successfully per the AI's steps, but I am unsure about unit tests passed - I am not confident in syntax accuracy. I need more robust testing across integration, deployment, and regression phases to truly verify end-to-end operational integrity."*

These conversations make a case for a robust testing suite results as an important indicator to establish trust in the output of such LLM-based systems. Further research should be done to explore the best way to present such information to participants.

6 CONCLUSION

We explore human-AI partnerships for code migration. Through interviews with developers using an LLM-based code migration tool called Amazon Q Code Transformation, we identified common interaction pattern themes like enabling refinement of AI-led updates and establishing trust guidelines for effectively complementing human capabilities with Generative AI.

While our study focuses on Java code migrations, we believe the identified themes are generalizable for code migrations of other programming languages like Python and Rust. Most projects involving upgrades, dependency changes, and signature changes require similar code modifications (as depicted in Figure 1), regardless of language. It would be interesting to explore how these guidelines apply when migrating code from one language to another, such as Java to Rust, which may necessitate more human involvement due to language proficiency. Additional promising research avenues include architecture overhauls and breaking down monolithic applications, which involve more complex code changes.

A WELCOME SCRIPT AND SAMPLE STEPS

This appendix shows the welcome script and sample steps we provided to the participants. Please note that since the study deployed a "think out loud" session, follow up/clarifying questions were noted and asked to the participants during the session.

A.1 Welcome Script

The welcome for our study participants is as follows.

Hello! Thank you so much for taking the time to participate in this user research study today. My name is {name of the interviewer} and I'll be walking you through this session.

*Before we begin, I want to give you a quick overview of what we'll be doing today. The purpose of this study is to understand how you may interact with the output of AI, which in this case is a code package converted from Java 8 to 17. We will be making use of an open-source code package originally in Java 8. This code package is converted using Generative AI. {This code package is fully compilable as per the AI (*E1*) / not fully compilable (*E2*).} Your goal in today's session will be to review the code generated by AI using your native diff browser in the IDE and make a decision in the next 30-45 mins as to whether you feel confident in using this code as is for java upgrade on the repository shared. Keep in mind that AI may hallucinate and may have made wrong changes. Along with the code base, you are also given a list of steps that AI took to fully migrate the code. I request you to talk out loud as you perform the review. We may ask follow up questions based on your actions.*

If at any time you have questions or need a break, just let me know. I want this to be a comfortable experience for you.

I'll be recording our session today so I can review it later and capture your feedback accurately. The recording will only be used internally and will not be shared.

Do you have any questions before we get started? [...] Okay, great! Let's begin.

First, I'd like to ask you a few warm-up questions ...

1. How much software development experience do you have?

2. Have you done java migrations before? If yes, what were the source and target language?

A.2 Sample Companion Guide

Each companion guide came with a description of the repository a participant was given, along with list of steps that the AI took to perform the migration. This same guide was given to participant P2 as part of experiment E1.

Project Objective. The Flash Sales project[6] aims to implement a flash sales system using a technology stack that includes Spring Boot 2.x, MyBatis, Redis, Swagger2, and Lombok. The system provides a flash sales API and employs two types of rate-limiting mechanisms: a counter-based approach and a token bucket (leaky bucket) approach implemented via a Lua script. Once a request passes the rate-limiting checks, a distributed lock is applied to ensure synchronized access to the database. To improve performance, initial inventory data is loaded into a cache, and operations are performed on this cached data. Database write operations are carried out asynchronously by placing the processed data into a queue, which is then consumed by another thread. Alternatively, the data can be written directly to a message queue for more robust handling.

Steps of Migration.

Step 1: The bot switched the java version in `pom.xml` to 17 and executed `mvn clean compile`. The bot observed the following error: `failed to read candidate component class.`

Step 2: The bot solved the issue by adding `<version>2.5.0</version>` to all spring-boot-* dependencies and plugins in `pom.xml`.

Step 3: The bot executed `mvn clean compile` again and observed a new error: `package org.junit.runner does not exist`.

[6] *https://github.com/stonehqs/flashsales*

Step 4: The bot added `junit` dependency.

Step 5: The bot executed `mvn clean compile` again and observed build success making the migration complete.

REFERENCES

[1] Kamel Alrashedy. 2023. Language Models are Better Bug Detector Through Code-Pair Classification. *arXiv preprint arXiv:2311.07957* (2023).

[2] Saleema Amershi, Dan Weld, Mihaela Vorvoreanu, Adam Fourney, Besmira Nushi, Penny Collisson, Jina Suh, Shamsi Iqbal, Paul N. Bennett, Kori Inkpen, Jaime Teevan, Ruth Kikin-Gil, and Eric Horvitz. 2019. Guidelines for Human-AI Interaction *(CHI '19)*.

[3] Shraddha Barke, Michael B James, and Nadia Polikarpova. 2023. Grounded copilot: How programmers interact with code-generating models. *Proceedings of the ACM on Programming Languages* 7, OOPSLA1 (2023), 85–111.

[4] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[5] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128* (2023).

[6] Nazli Cila. 2022. Designing Human-Agent Collaborations: Commitment, Responsiveness, and Support *(CHI '22)*.

[7] Lucas Colusso, Cynthia L Bennett, Gary Hsieh, and Sean A Munson. 2017. Translational resources: Reducing the gap between academic research and HCI practice. In *Proceedings of the 2017 conference on designing interactive systems*. 957–968.

[8] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[9] Erik Du Plessis. 1994. Recognition versus recall. *Journal of Advertising Research* 34, 3 (1994), 75–92.

[10] Shubhang Shekhar Dvivedi, Vyshnav Vijay, Sai Leela Rahul Pujari, Shoumik Lodh, and Dhruv Kumar. 2023. A Comparative Analysis of Large Language Models for Code Documentation Generation. *arXiv preprint arXiv:2312.10349* (2023).

[11] Upol Ehsan, Q. Vera Liao, Michael Muller, Mark O. Riedl, and Justin D. Weisz. 2021. Expanding Explainability: Towards Social Transparency in AI Systems *(CHI '21)*.

[12] Judith E Fan, Monica Dinculescu, and David Ha. 2019. Collabdraw: an environment for collaborative sketching with an artificial agent. In *Proceedings of the 2019 on Creativity and Cognition*. 556–561.

[13] Cheng-Zhi Anna Huang, Curtis Hawthorne, Adam Roberts, Monica Dinculescu, James Wexler, Leon Hong, and Jacob Howcroft. 2019. The bach doodle: Approachable music composition with machine learning at scale. *arXiv preprint arXiv:1907.06637* (2019).

[14] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2023. SWE-bench: Can Language Models Resolve Real-World GitHub Issues? *arXiv preprint arXiv:2310.06770* (2023).

[15] Sungmin Kang, Gabin An, and Shin Yoo. 2023. A Preliminary Evaluation of LLM-Based Fault Localization. *arXiv preprint arXiv:2308.05487* (2023).

[16] Pegah Karimi, Jeba Rezwana, Safat Siddiqui, Mary Lou Maher, and Nasrin Dehbozorgi. 2020. Creative sketching partner: an analysis of human-AI co-creativity. In *Proceedings of the 25th International Conference on Intelligent User Interfaces*. 221–230.

[17] Sunnie S. Y. Kim, Elizabeth Anne Watkins, Olga Russakovsky, Ruth Fong, and Andrés Monroy-Hernández. 2023. "Help Me Help the AI": Understanding How Explainability Can Support Human-AI Interaction *(CHI '23)*.

[18] Marie-Anne Lachaux, Baptiste Roziere, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised translation of programming languages. *arXiv preprint arXiv:2006.03511* (2020).

[19] Chen Ling, Xujiang Zhao, Jiaying Lu, Chengyuan Deng, Can Zheng, Junxiang Wang, Tanmoy Chowdhury, Yun Li, Hejie Cui, Tianjiao Zhao, et al. 2023. Beyond One-Model-Fits-All: A Survey of Domain Specialization for Large Language Models. *arXiv preprint arXiv:2305.18703* (2023).

[20] Shuai Ma, Ying Lei, Xinru Wang, Chengbo Zheng, Chuhan Shi, Ming Yin, and Xiaojuan Ma. 2023. Who Should I Trust: AI or Myself? Leveraging Human and AI Correctness Likelihood to Promote Appropriate Trust in AI-Assisted Decision-Making *(CHI '23)*.

[21] Andrew M McNutt, Chenglong Wang, Robert A Deline, and Steven M Drucker. 2023. On the design of ai-powered code assistants for notebooks. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. 1–16.

[22] Jane N Mosier and Sidney L Smith. 1986. Application of guidelines for designing user interface software. *Behaviour & information technology* 5, 1 (1986), 39–46.

[23] Changhoon Oh, Jungwoo Song, Jinhan Choi, Seonghyeon Kim, Sungwoo Lee, and Bongwon Suh. 2018. I lead, you help but only with enough details: Understanding user experience of co-creation with artificial intelligence. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. 1–13.

[24] Cheng Peng, Xi Yang, Aokun Chen, Kaleb E Smith, Nima PourNejatian, Anthony B Costa, Cheryl Martin, Mona G Flores, Ying Zhang, Tanja Magoc, et al. 2023. A Study of Generative Large Language Model for Medical Research and Healthcare. *arXiv preprint arXiv:2305.13523* (2023).

[25] Marc Pinski, Martin Adam, and Alexander Benlian. 2023. AI Knowledge: Improving AI Delegation through Human Enablement. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. 1–17.

[26] Atsushi Shirafuji, Yusuke Oda, Jun Suzuki, Makoto Morishita, and Yutaka Watanobe. 2023. Refactoring Programs Using Large Language Models with Few-Shot Examples. *arXiv preprint arXiv:2311.11690* (2023).

[27] Lingyun Sun, Zhuoshu Li, Yuyang Zhang, Yanzhen Liu, Shanghua Lou, and Zhibin Zhou. 2021. Capturing the Trends, Applications, Issues, and Potential Strategies of Designing Transparent AI Agents *(CHI EA '21)*.

[28] AWS AI Team. [n. d.]. Amazon Q Code Transformation (Preview). https://aws.amazon.com/q/aws/code-transformation/. Accessed: 2023-12-30.

[29] Linda Tetzlaff and David R Schwartz. 1991. The use of guidelines in interface design. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 329–333.

[30] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2018. Deep learning similarities from different representations of source code. In *Proceedings of the 15th international conference on mining software repositories*. 542–553.

[31] Fengjie Wang, Xuye Liu, Oujing Liu, Ali Neshati, Tengfei Ma, Min Zhu, and Jian Zhao. 2023. Slide4N: Creating Presentation Slides from Computational Notebooks with Human-AI Collaboration *(CHI '23)*.

[32] Thomas Weber, Heinrich Hußmann, Zhiwei Han, Stefan Matthes, and Yuanting Liu. 2020. Draw with me: Human-in-the-loop for image restoration. In *Proceedings of the 25th International Conference on Intelligent User Interfaces*. 243–253.

[33] Justin D Weisz, Michael Muller, Jessica He, and Stephanie Houde. 2023. Toward general design principles for generative AI applications. *arXiv preprint arXiv:2301.05578* (2023).

[34] Justin D Weisz, Michael Muller, Stephanie Houde, John Richards, Steven I Ross, Fernando Martinez, Mayank Agarwal, and Kartik Talamadupula. 2021. Perfection not required? Human-AI partnerships in code translation. In *26th International Conference on Intelligent User Interfaces*. 402–412.

[35] Burak Yetiştiren, Işık Özsoy, Miray Ayerdem, and Eray Tüzün. 2023. Evaluating the Code Quality of AI-Assisted Code Generation Tools: An Empirical Study on GitHub Copilot, Amazon CodeWhisperer, and ChatGPT. *arXiv preprint arXiv:2304.10778* (2023).

[36] Qiaoning Zhang, Matthew L Lee, and Scott Carter. 2022. You Complete Me: Human-AI Teams and Complementary Expertise *(CHI '22)*.