How Does Experience Influence Developer Perceptions of Atoms of Confusion?

Guoshuai (Albert) Shi, Farshad Kazemi, Shane McIntosh, Michael W. Godfrey
University of Waterloo
Waterloo, Ontario, Canada
{albert.shi,f2kazemi,shane.mcintosh,migod}@uwaterloo.ca

Abstract

Atoms of Confusion (AoCs) are small, syntactically valid code patterns that can increase cognitive load during program comprehension. Earlier research suggested that AoCs are common and potentially harmful, but more recent studies have questioned whether their effects generalize beyond less experienced developers. This confirmatory study aims to reexamine whether the presence of AoCs slows comprehension or alters repair preferences. Moreover, we examine whether these effects are moderated by developers' programming experience. We investigate task completion time and the kind of repairs developers prefer when interacting with code containing AoCs. We propose a two-phase study consisting of a pre-screening questionnaire and a controlled experiment. The questionnaire will function as a qualification instrument. In the experiment, participants will complete eight Java comprehension tasks, four with an AoC and four without. For each task, developers are asked to identify a seeded defect and to rank three functionally equivalent repairs differing in AoC inclusion. Task completion time and the top-ranked repair will be analyzed using mixed-effects linear and multinomial regression models, with AoC presence as the manipulated factor and programming experience as a covariate.

CCS Concepts

• **Software and its engineering** → **Consistency**; • **General and reference** → *Experimentation*; Empirical studies.

Keywords

program comprehension, Atoms of Confusion, controlled experiment

1 Introduction

Code that is difficult to understand slows software maintenance [14, 15], increases the likelihood of introducing defects [11, 23], and reduces team productivity [8, 20]. While many readability issues arise from structural or semantic complexity, confusion can also stem from syntactic patterns that are valid yet cognitively demanding. Gopstein *et al.* identified a set of such patterns and termed them *Atoms of Confusion* (AoCs) [10]. Subsequent studies have catalogued AoCs in multiple languages [11, 18] and examined their impact on comprehension and defect proneness [4, 5, 21]. However, findings remain inconsistent as some studies report measurable slowdowns, while others find only negligible effects. These discrepancies may arise from differences in task realism, participant expertise, or the operationalization of comprehension. In particular, recent work has focused primarily on novice populations, leaving open whether AoC effects vary with developer experience.

We investigate this gap through a controlled experiment designed to test whether the presence of AoCs affects developers' task efficiency and the repair preferences of developers, and whether these effects are moderated by their programming experience. Participants complete eight Java comprehension tasks, half containing a target AoC and half without. For each task, they identify a seeded defect and rank three functionally-equivalent repairs that vary in AoC inclusion: one removes the AoC, one retains it, and the third introduces a different AoC. We will record task completion time and repair rankings to capture both comprehension efficiency and stylistic preference. Analyses will use mixed-effects linear and multinomial regression, with AoC presence as the manipulated factor and programming experience (in years) as a covariate.

This design allows direct comparison of comprehension and repair behaviour on code with and without AoCs, while controlling for participant-level variability. Findings aim to clarify when and for whom AoCs pose cognitive challenges, informing future readability tools, review practices, and training approaches that account for programming experience.

2 Background and Related Work

The concept of *Atoms of Confusion* (AoCs), i.e., syntactic patterns that may cause confusion, was introduced by Gopstein *et al.* [10], who identified 15 such patterns in C-like languages. More recent work found these patterns to be both prevalent and defect-prone in open-source C/C++ systems [11]. Listing 1 shows an example from the FreeBSD repository, where the developer confused the order of precedence between the bitwise OR and ternary operators, which introduced a defect. More recent studies have shown that the presence of AoCs in code snippets was not necessarily why the participants incorrectly evaluated the outcome, nor did correct evaluations guarantee understanding of AoCs [9, 25].

```
ulpmc->cmd = htobe32(V_ULPTX_CMD(ULP_TX_MEM_WRITE) |
     is_t4(sc) ? F_ULP_MEMIO_ORDER : F_T5_ULP_MEMIO_IMM);
```

Listing 1: Defective code snippet with Infix Order Precedence AoC

Subsequent research investigated the behavioural and physiological effects of AoCs. De Oliveira *et al.* [5] used eye tracking to show that AoCs slowed comprehension time by over 40% and drew focused visual attention. Da Costa *et al.* [4] conducted a controlled experiment with 32 undergraduate participants classified as novices in the target language, comparing programs containing AoCs with functionally equivalent clarified versions. While their study provides detailed evidence of how AoCs affect novice programmers, it does not examine how these effects vary across different levels of developer experience or model experience as a moderating factor.

Researchers have also studied AoCs that can occur in Java programs. Langhout and Aniche [18] proposed 14 types of AoCs for Java, and surveyed 132 students, showing that some patterns hinder comprehension more than others. Mendes *et al.* [21] and Tahsin *et al.* [27] analyzed open-source Java projects and found that while some AoCs types were common, others were rare. Moreover, they found that the prevalence of AoCs correlated with project-level factors, such as the depth of the inheritance tree, the project age, and indicators of maintenance activity.

Prior work has also scrutinized the impact of AoCs. Yeh *et al.* [28] found no significant increase in cognitive load (measured using EEG signals) between code with and without AoCs. Pinheiro *et al.* [22] found that only 4.5% of the studied commits explicitly removed AoCs, and just 11.7% of those removals were the direct cause of fixes or improvements. Bogachenkova *et al.* [2] observed only weak correlations between reviewer confusion and the presence of AoCs, with most *Pull Request*s (PRs) retaining them after review. In prior work [25], we found no consistent link between AoCs and defect-fixing activity in the PRs of six Java projects. In fact, some bug-fixing PRs introduced more AoCs than they removed.

While early findings supported the idea that AoCs negatively affect code quality [11], recent work reports mixed results [2, 22, 25]. This contrast highlights a gap in our understanding and motivates deeper empirical investigation into how developers interpret and respond to AoCs. In particular, prior experimental work has largely focused on novice populations, leaving open whether the impact of AoCs differs systematically with programming experience. We hypothesize that these contradictory findings of past work may in part be explained by differences in developer expertise. The study that we propose will evaluate this hypothesis.

3 Research Questions and Hypotheses

This experiment reinvestigates how the presence of AoCs affects the perception and repair of defective code, and whether programming experience moderates these effects, assessed through self-reported measures. We focus on the time developers take to read and repair defective code with and without AoCs, and their rankings of three similar repairs that differ in terms of the presence of AoCs.

We set out to address two research questions, grounded in the assumption that experienced developers are more tolerant of syntactic complexity and may prioritize maintainability, conciseness, or minimal edit. In contrast, less experienced developers may favour clearer, easier-to-read repairs.

RQ1 Is the AoC effect on end-to-end debugging task time moderated by programming experience?
$H1_A$: Tasks containing an AoC will take longer to complete than tasks without one [9, 10], with less experienced developers exhibiting larger increases in task time.
$H1_\emptyset$: AoC presence does not affect task completion time, regardless of the amount of experience.

RQ2 Is the AoC effect on the type of repair preferred moderated by programming experience?
$H2_A$: Repair preference is affected by AoC presence [9, 10], with programming experience modifying the preference.
$H2_\emptyset$: AoC presence does not influence repair preference, regardless of the amount of experience.

4 Study Design

The study consists of two phases: a **pre-screening questionnaire** and a **controlled experiment**. Both phases will be conducted online using a custom web interface. All anonymized data will be released via Zenodo to support replication.

4.1 Pre-screening Questionnaire

The pre-screening questionnaire collects participant demographics and programming background to ensure that only qualified participants continue to the controlled experiment. There are three categories of questions (Table 1): (a) **Qualifying questions**, which are used to exclude participants who do not meet the inclusion criteria, (b) **Programming experience question**, used to retrieve the key covariate for analysis, and (c) **Follow-up questions**, used to support post-hoc interpretation or exploratory analyses.

Qualifying Questions. Participants must meet specific criteria to reduce variation caused by confounding factors and improve internal validity. These criteria were chosen based on prior empirical studies of code comprehension [5, 18].

To qualify for the experiment, participants must:

- be proficient in English (to avoid misinterpretation due to language processing difficulties);
- have received a formal education in programming; and
- have at least a year of professionalprogramming experience.

These conditions are intended to filter out participants who might be confused by syntax, task framing, or language semantics unrelated to the manipulated presence of AoCs.

Programming Experience Question. The key covariate in our study is participants' self-reported years of experience programming in Java. We operationalize this as a continuous variable, which allows us to investigate experience as a gradient rather than a binary (novice vs. expert) construct.

Follow-up Questions. Although not used in the primary analysis, we collect additional background information such as code review frequency and approach to unfamiliar code. These may inform future exploratory analyses or help explain variability in comprehension strategies.

4.2 Controlled Experiment

In the second phase, participants complete eight tasks, four of which are associated with different types of AoCs, and four of which do not feature AoCs. The four types of AoC used are *Conditional Operator, Infix Operator Precedence, Pre Increment Decrement,* and *Type Conversion. They are selected since they are the most prevalent and are meaningfully modified in more than one project in our study [25], implying that developers encounter and interact with them more often than the other types of AoC.*

For each task, we present a defective Java code snippet (15–25 LOC), each containing a seeded defect that is independent of the AoC, allowing us to isolate the effect of AoC presence on task performance. We ask participants to identify the bug, after which we present three repairs and ask them to provide three rankings of the repairs: their preference, ease to read and understand, and ease to modify and refactor. The last two criteria serve as attention checks and provide exploratory analysis materials. Example tasks can be found in our online appendix [26]. We expect that each

Table 1: Pre-screening questionnaire items and their operationalization.

Question	Type	Operationalization
Category A. Qualifying Questions		
Q1. What is your level of proficiency in English?	Nominal	Native, Very Proficient, Somewhat proficient, or not proficient.
Q2. Have you received a formal education in programming?	Binary	Yes or No.
Q3. How many years have you been programming professionally?	Ordinal (numeric)	An integer between 0 and 100, inclusive.
Category B. Programming Experience Questions		
Q4. How many years have you been programming in Java?	Ordinal (numeric)	An integer between 0 to 100, inclusive.
Category C. Follow-up Questions		
Q5. On a scale from 1 to 10, how do you estimate your programming experience?	Ordinal (numeric)	1: very inexperienced to 10: very experienced.
Q6. Which of the following best describes your current role in software development?	Nominal	Software developer, QA engineer, DevOps engineer, researcher, or other.
Q7. How many hours per week do you work on OSS projects on average?	Ordinal	A number between 0 and 168, inclusive.
Q8. What programming languages have you used for more than one year?	Multiple choice	Java, Kotlin, Python, C, C++, Rust, ..., and others.
Q9. How many years have you been programming in [each programming language selected in Q9]?	Ordinal (numeric)	An integer between 0 and 100, inclusive.
Q10. How often do you perform code reviews?	Ordinal	Daily, Weekly, Monthly, Rarely, Never.
Q11. Do you follow an explicit code style guide in some projects?	Binary	Yes or No.
Q12. How do you usually approach understanding unfamiliar code?	Nominal	Refer to documentation, Test/debug the code, Discuss with colleagues, Read the code, or other.
Q13. How much time do you typically spend reading code versus writing it?	Ordinal	Mostly reading, Balanced, or Mostly writing.

task would take approximately 3–5 minutes for a developer with at least one year of Java experience. The tasks are domain-neutral and designed around general-purpose programming constructs (e.g., loops, conditionals, expressions), allowing us to isolate control- and data-flow effects independent of domain knowledge.

We record the order in which each task is presented, allowing us to model potential fatigue or learning effects over the course of the session. Task order will be included as a covariate in our statistical models to control for systematic variation due to position and to explore whether comprehension behaviour shifts over time.

Each of the three suggested repairs corrects the defect, but each has a unique property. *Repair X* removes the original AoC by rewriting the expression using more verbose logic. *Repair Y* retains the AoC, correcting only the functional error. *Repair Z* introduces different AoCs, yielding a shorter version compared to *X* and *Y*.

The set of candidate repairs reflects real-world refactoring trade-offs, where some developers may favour brevity while others prioritize clarity and readability. To prevent arbitrary selection, participants are required to describe the defect and rank the correct solutions according to preference and readability.

Participants will perform tasks using a *custom-built web interface* that randomizes the order of tasks and repair options to mitigate order effects. The interface logs two primary behavioural outcomes:

- The time taken to interpret and respond to each task (measured from snippet display to repair ranking); and
- The repair ranked first by the participant.

No hard time limits are enforced; participants may proceed at their own pace. To prevent idle periods from inflating task times, we will flag and exclude implausibly long durations (e.g., times exceeding several standard deviations above the task median). Results where the participants incorrectly identify the bug will not be excluded from analysis. Each task is self-contained and includes only the minimal code required to observe the effect of the manipulated AoC. After completing all of the tasks, participants are asked to justify their repair ranking. This will provide insight into whether developers prioritize clarity or brevity when repairing code.

Each task contributes one observation. Since the same participant completes multiple tasks, those observations are more similar to each other than to observations from other participants. Likewise, different participants complete the same tasks, making observations from the same task more similar to each other than to observations from other tasks. To account for this non-independence, we include random intercepts for participants and tasks. We collect multiple observations per participant while modelling AoC effects individually rather than aggregating across tasks, reflecting the distinct syntactic pattern and comprehension challenge each AoC presents.

4.3 Pilot Study

Before full deployment, we will conduct a pilot study using convenience sampling (e.g., labmates or colleagues within our research group), aiming to recruit at least ten participants. The pilot will help assess the clarity of task instructions, web interface functionality, logging accuracy, task timing, and the comprehensibility of code snippets. We will check for unintentional cues in the code that could bias participant rankings, especially given the syntactic subtlety of AoCs. If pilot results reveal unclear instructions, overly difficult tasks, or biased code presentation, we will revise or replace the affected items. Insights from the pilot will guide minor usability improvements, which will be documented transparently. No pilot data will be included in the final analysis.

5 Participants

The target population consists of experienced Java developers. Participants will be recruited through professional networks, mailing lists, academic contacts, developer communities, and crowdsourcing platforms. We will also leverage our industrial connections to reach a wide audience. To be eligible, participants must have at least one year of programming experience and consider themselves to be proficient in Java. They must also complete the pre-screening questionnaire. Participants who fail embedded attention checks, submit incomplete responses, or indicate no exposure to Java will be excluded from the analysis.

Participants are expected to complete all eight tasks in a single session, although no time limit is imposed. Incentives will be offered to participants who complete all tasks. For each task, we collect two primary behavioural outcomes: (1) the time taken to solve the task, and (2) the top-ranked repair from the three available options.

The primary manipulated variable is the presence or absence of an AoC in each task. Programming experience is included as a continuous covariate (in years) to investigate potential moderation effects on task completion time and repair preference.

Sample Size

We estimated the required sample size using an approximate power analysis for a within-subject design [7, 19]. Each participant completes $k = 8$ tasks, with half containing an AoC. We follow recent findings that suggest a 10 to 15 percent increase in comprehension time due to atoms of confusion [4, 5]. This corresponds to an expected log time effect of $\beta_1 = \log(1.10 \text{ to } 1.15) \approx 0.10$ to 0.14. Since we do not have pilot data, we assume a residual standard deviation of $\sigma = 0.40$ and a within-subject correlation of $\rho = 0.5$.

Under these assumptions, the effective number of independent observations per participant is

$$n_{\text{eff}} = \frac{k}{1 + (k-1)\rho} = \frac{8}{1 + 7(0.5)} \approx 1.78.$$

The standardized effect is $d = \beta_1/\sigma \approx 0.25$. Detecting an effect of this magnitude with power $1 - \beta = 0.8$ and $\alpha = 0.05$ requires

$$N_{\text{eff}} = \left(\frac{z_{1-\alpha/2} + z_{1-\beta}}{d}\right)^2 \approx 125, \quad N_{\text{participants}} = \frac{N_{\text{eff}}}{n_{\text{eff}}} \approx 70.$$

Allowing a slight margin for the interaction term, potential model complexity, and unexpected attrition, we target a sample size of **75 participants**.

For the multinomial choice analysis [6, 16], we assume a baseline probability of choosing the non-AoC repair of $p_1 = \frac{1}{3}$. Treating as minimally relevant an odds ratio of OR = 1.6, the implied alternative probability is $p_2 = \frac{4}{9} \approx 0.444$ ($\Delta p \approx 0.11$). Using the Hsieh et al. approximation for logistic regression [16] with a two-sided $\alpha = 0.05$ and power $1 - \beta = 0.80$ gives

$$N \approx \frac{(1.96 + 0.84)^2}{[p_1(1-p_1) + p_2(1-p_2)]\,(\ln \text{OR})^2} \approx 76.$$

supporting a similar target of $\mathbf{N \approx 75}$ participants (≈ 600 observations total, power ≈ 0.8–0.85).

Alternative Considerations and Final Decision. We considered expanding our models to include additional covariates (e.g., code review frequency or comprehension style). However, modelling multiple covariates or their interactions (e.g., 3 experience levels × 3 behavioural categories) would substantially increase the required sample size. Preliminary estimates suggest such designs would require **about 250 participants**, which we consider to be infeasible.

We also considered treating programming experience as a categorical variable (*novice*, *intermediate*, *expert*) to simplify interpretation and relax linearity assumptions. For task completion time, an ANOVA power analysis [17] with large effect size ($f = 0.4$), $\alpha = 0.05$, and 80% power requires about **66 participants**. This is close to our target sample size, so if time permits, we will conduct an exploratory analysis using categorial programming experience.

Given these trade-offs, we retain a **continuous formulation** of programming experience as a covariate. This preserves statistical power under realistic sample sizes and allows flexible modelling (e.g., splines or robust regression) to capture potential non-linear effects without inflating sample size requirements.

We evaluated three realistic sampling scenarios:

- **Baseline power target:** 75 participants (χ^2 approximation)
- **Fully powered MLR scenario:** 187 participants (multinomial regression with a continuous covariate)
- **Pragmatic compromise:** 80 participants (sufficient for task-level regressions and marginal MLR modelling)

We therefore target a final sample size of **75 participants**, balancing empirical rigour with recruitment feasibility. This sample size exceeds the requirements for linear mixed modelling under relaxed assumptions and approaches adequacy for repair preference modelling with a continuous covariate.

If resources allow, data collection may continue up to 187 participants. If the final sample falls short, we will document this limitation and apply sensitivity analyses or bootstrapped confidence intervals to assess the robustness of our findings.

6 Analysis Plan

This section outlines our statistical analysis for the two dependent variables: task completion time (DV1) and repair preference (DV2). We test our hypotheses using regression models appropriate to each outcome, incorporating participant-level random effects to account for repeated measures.

6.1 Task Completion Time (DV1)

We model log-transformed completion time using a crossed random-effects linear mixed model [3, 24]:

$$\log t_{ij} = \beta_0 + \beta_1 A_{ij} + \beta_2 x_i + \beta_3 (A_{ij} \times x_i) + \zeta_{ij}.$$

Here, t_{ij} is the completion time for participant i on task j. $A_{ij} \in \{0, 1\}$ indicates AoC presence (counterbalanced), x_i is mean-centred experience, and ζ_{ij} denotes random components defined in Section 6.3. Residuals will be inspected for normality and homoscedasticity. If substantial deviations are detected (e.g., strong skew or non-linearity), we will apply appropriate alternative modelling strategies such as variance-stabilizing transformations or generalized linear mixed models. These adjustments will follow standard recommendations for handling non-normal data in behavioural experiments.

6.2 Repair Preference (DV2)

The repair that is ranked first ($y_{ij} \in \{1, 2, 3\}$) is modelled using a mixed-effects multinomial logit [12, 13]:

$$\Pr(y_{ij}=k) = \frac{\exp(\eta_{ijk})}{\sum_{m=1}^{3} \exp(\eta_{ijm})}, \tag{1}$$

$$\eta_{ijk} = \beta_{0k} + \beta_{1k} A_{ij} + \beta_{2k} x_i + \beta_{3k} (A_{ij} \times x_i) + \zeta_{ij}. \tag{2}$$

Category k=1 (no-AoC repair) serves as the reference. ζ_{ij} follows the random structure that we explain in Section 6.3. Fixed effects will be reported as odds ratios with 95% confidence intervals and marginal predicted probabilities by A_{ij} and x_i. Model fit will be evaluated using residual diagnostics and likelihood-ratio tests. If convergence issues occur, penalized likelihood (ridge or Firth) will be applied. If the full multinomial mixed model is infeasible, three one-vs-rest logistic regression models with cluster-robust standard errors and task fixed effects will be fit to check for robustness.

6.3 Random Effects and Error Structure

Both models include crossed random effects [1, 24] for participants and tasks:

$$\zeta_{ij} = u_i + u_{1i} A_{ij} + v_j + \varepsilon_{ij}.$$

Here, $u_i \sim \mathcal{N}(0, \sigma_u^2)$ is a participant intercept describing how participant i differs from the overall average, $u_{1i} \sim \mathcal{N}(0, \sigma_{u1}^2)$ is a participant-specific AoC slope describing how the AoC effect differs for participant i, $v_j \sim \mathcal{N}(0, \sigma_v^2)$ is a task intercept describing baseline differences between tasks, and $\varepsilon_{ij} \sim \mathcal{N}(0, \sigma^2)$ is the residual error capturing remaining trial-level variability. An unstructured covariance allows correlation between participant intercepts and slopes. If convergence fails, the random slope term will be removed, yielding (1|participant) + (1|task).

6.4 Covariates and Task Order

We record task order as a covariate to account for fatigue or learning effects that may influence task time. For exploratory purposes, we may pool responses across tasks to model aggregate trends in repair preference, although our primary analysis remains task-level.

6.5 Exploratory Analyses

We plan to explore whether self-reported code-reading habits, code review frequency, or preferred comprehension strategy correlate with task time or repair preference. These analyses will be labelled as *exploratory* and will not be used to draw confirmatory conclusions.

- **Q5-7:** We may substitute self-estimation of programming experience, current role, or work time on OSS for years of experience, or include them as additional predictors when modelling task time or repair preference.
- **Q8-9:** Considering the participants' experiences in other languages may offer insight into how diversity in programming language shapes comfort with different coding styles or syntactic conventions, particularly with AoCs.
- **Q10:** Considering the frequency of code reviews may help inform post hoc exploration of whether review frequency relates to attention to stylistic clarity.
- **Q11:** Knowing whether the participants follow explicit code style guides may help contextualize whether coding style aligns with certain repair preferences, such as preferring clearer or more formally structured code.
- **Q12:** We may explore whether preferred comprehension strategies for understanding unfamiliar code correspond to task behaviour, such as time spent or interaction with different repair types.
- **Q13:** The relative proportion of time spent reading versus writing code may clarify how reading-heavy workflows relate to sensitivity to syntactic features like AoCs.

We will also report descriptive statistics for each task (e.g., average time, repair preference proportions) to contextualize differences across AoCs.

Any groupings derived post hoc (e.g., binning participants into experience tertiles) will be labelled as exploratory. These analyses will not support confirmatory claims and are intended solely for hypothesis generation or interpretation.

7 Expected Outcomes and Interpretation

7.1 Task Completion Time (RQ1)

We expect that introducing AoCs into defective code will increase task completion time for most developers. However, this increase is expected to be smaller for more experienced developers.

If the mixed-effects linear model reveals a consistent negative association between programming experience and the AoC-induced time increase across tasks, this will support $\boldsymbol{H1_A}$. Conversely, inconsistent or null effects will support the null hypothesis ($\boldsymbol{H1_\emptyset}$).

7.2 Repair Preference (RQ2)

We expect repair preferences to vary with programming experience:

- More experienced developers may prefer concise repairs, even when they retain additional AoCs.
- Less experienced developers may favour longer, clearer repairs that remove potentially confusing constructs.

If the mixed-effects multinomial regression shows that the probability of selecting a repair containing more AoCs increases with experience (relative to the no-AoC repair), this will support $\boldsymbol{H2_A}$. If no significant trend emerges, or the no-AoC repair is equally preferred across experience levels, the null hypothesis will be retained.

Alternative Interpretations and Null Results

If neither dependent variable is significantly associated with experience, several interpretations are possible:

- Developers may not perceive AoCs as sufficiently confusing to affect task completion time or repair preference.
- Differences between repair options may be too subtle to elicit consistent behavioural changes.
- Other factors, such as individual reasoning strategies or task-specific context, may play a stronger role than experience.

Null results will be interpreted cautiously. Exploratory follow-up analyses will be conducted to generate hypotheses for future work. Our goal is not to generalize across all potentially confusing code patterns, but to clarify the role of programming experience in how confusion is recognized and resolved during code comprehension.

8 Threats to Validity

We outline potential threats that may influence the interpretation of our findings and describe the steps taken to mitigate them.

8.1 Internal Validity

Fatigue and learning effects. Participants will complete eight tasks in a single session, which may induce fatigue or learning effects. We mitigate these by randomizing task order and allowing participants to proceed at their own pace. Task order will be recorded and included as a covariate, and implausibly fast or slow responses will be examined and excluded if necessary.

Instrumentation bias. All participants will use the same web-based interface, which controls code presentation and logs interaction times consistently. Timing is captured via client-side JavaScript events to ensure accuracy across participants.

Use of generative AI tools. We will instruct participants not to use artificial intelligence tools during the tasks; we will emphasize that the study is intended to measure individual performance. In addition, we will (1) require participants to close external tools, (2) disable copy–paste in the task interface to discourage external queries, and (3) monitor completion-time distributions and response patterns for anomalies. We acknowledge that perfect enforcement is not possible, but these measures reduce the likelihood of AI assistance and allow us to detect suspicious cases post hoc.

8.2 Construct Validity

Programming experience measurement. Years of programming in Java is used as a proxy for experience. Although common in empirical software engineering, this measure may not capture depth or quality of experience. Eligibility filters and exploratory covariates, such as code review frequency and reading habits, help mitigate this limitation.

Task realism. Tasks are short and synthetic to isolate specific AoCs, which improves experimental control but may limit ecological realism. Similar task designs have been successfully applied in prior controlled studies [5, 10, 18].

Repeated measures bias. Each participant completes multiple tasks, which may introduce intra-individual dependencies. While each task targets a distinct AoC and is modelled independently in primary analyses, we will also conduct robustness checks using mixed-effects models with participant-level random intercepts.

False positives in non-defective code. All tasks in this study contain a defect that participants are expected to identify. Thus, the design does not measure whether AoCs may also induce false positives in non-defective code (e.g., suspecting bugs where none exist or proposing unnecessary changes). Examining this effect would require adding non-defective code snippets and substantially increasing the number of tasks to maintain balance, which would lengthen the study and alter feasibility and statistical power relative to the existing design. We therefore do not include this dimension here and leave it to follow-up work using a complementary design.

8.3 External Validity

Generalizability. Inclusion criteria may bias the sample toward educated, Java-proficient developers. We do not claim that results generalize to developers with different backgrounds, languages, or experience levels. Future replications can expand eligibility to evaluate trends across broader populations.

Ecological validity. Real-world comprehension occurs in richer contexts (IDE, version control, documentation, and LLM-based agencies). This experiment captures isolated comprehension episodes, which is appropriate for detecting fine-grained effects of syntactic patterns like AoCs.

8.4 Conclusion Validity

We use appropriate statistical models for each dependent variable and justify our sample size with formal power analyses and realistic constraints. Model assumptions will be verified, and alternative methods applied as needed. Confirmatory analyses are clearly distinguished from exploratory analyses.

9 Conclusion

This registered report describes a two-part confirmatory study examining how programming experience influences the comprehension and repair evaluation of code containing *Atoms of Confusion* (AoCs). We combine a pre-screening questionnaire with a within-subject task-based experiment that measures task completion time and repair preference.

Our hypotheses test whether more experienced developers complete tasks faster and are more tolerant of AoCs, whereas less experienced developers may prioritize readability over brevity. Task-level linear mixed-effects regression and mixed-effects multinomial logistic regression will be used to evaluate these hypotheses.

By focusing on syntactically isolated yet semantically realistic code examples drawn from patterns commonly seen in software repositories, this study aims to clarify how AoCs interact with programming experience. The results may strengthen repository-based readability assessments and support tools and practices that identify or refactor potentially confusing code with greater awareness of developer background.

Ethics Statement

We will apply for approval from the University of Waterloo Research Ethics Committee after we finalize the experiment design.

References

[1] R Harald Baayen, Douglas J Davidson, and Douglas M Bates. 2008. Mixed-effects modeling with crossed random effects for subjects and items. *Journal of memory and language* 59, 4 (2008), 390–412. doi:10.1016/j.jml.2007.12.005

[2] Victoria Bogachenkova, Linh Nguyen, Felipe Ebert, Alexander Serebrenik, and Fernando Castor. 2022. Evaluating Atoms of Confusion in the Context of Code Reviews. In *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, New York, NY, USA, 404–408. doi:10.1109/ICSME55016.2022.00048

[3] Avital Cnaan, Nan M Laird, and Peter Slasor. 1997. Using the general linear mixed model to analyse unbalanced repeated measures and longitudinal data. *Statistics in medicine* 16, 20 (1997), 2349–2380. doi:10.1002/(sici)1097-0258(19971030)16:20<2349::aid-sim667>3.0.co;2-e

[4] José Aldo Silva da Costa, Rohit Gheyi, Fernando Castor, Pablo Roberto Fernandes de Oliveira, Márcio Ribeiro, and Baldoino Fonseca. 2023. Seeing confusion through a new lens: on the impact of atoms of confusion on novices' code comprehension. *Empirical Software Engineering* 28, 4 (18 May 2023), 81. doi:10.1007/s10664-023-10311-0

[5] Benedito de Oliveira, Márcio Ribeiro, José Aldo Silva da Costa, Rohit Gheyi, Guilherme Amaral, Rafael de Mello, Anderson Oliveira, Alessandro Garcia, Rodrigo Bonifácio, and Baldoino Fonseca. 2020. Atoms of Confusion: The Eyes Do Not Lie. In *Proceedings of the XXXIV Brazilian Symposium on Software Engineering* (Natal, Brazil) *(SBES '20)*. Association for Computing Machinery, New York, NY, USA, 243–252. doi:10.1145/3422392.3422437

[6] Eugene Demidenko. 2007. Sample size determination for logistic regression revisited. *Statistics in medicine* 26, 18 (2007), 3385–3397. doi:10.1002/sim.2771

[7] Peter J Diggle, Patrick J Heagerty, Kung-yee. Liang, and Scott L Zeger. 2002. *Analysis of Longitudinal Data*. Oxford University Press. doi:10.1093/oso/9780198524847.001.0001

[8] Sarah Fakhoury, Yuzhan Ma, Venera Arnaoudova, and Olusola Adesope. 2018. The Effect of Poor Source Code Lexicon and Readability on Developers' Cognitive Load. In *Proceedings of the 26th Conference on Program Comprehension* (Gothenburg, Sweden) *(ICPC '18)*. Association for Computing Machinery, New York, NY, USA, 286–296. doi:10.1145/3196321.3196347

[9] Dan Gopstein, Anne-Laure Fayard, Sven Apel, and Justin Cappos. 2020. Thinking aloud about confusing code: a qualitative investigation of program comprehension and atoms of confusion. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 605–616. doi:10.1145/3368089.3409714

[10] Dan Gopstein, Jake Iannacone, Yu Yan, Lois DeLong, Yanyan Zhuang, Martin K.-C. Yeh, and Justin Cappos. 2017. Understanding Misunderstandings in Source Code. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 129–139. doi:10.1145/3106237.3106264

[11] Dan Gopstein, Hongwei Henry Zhou, Phyllis Frankl, and Justin Cappos. 2018. Prevalence of Confusing Code in Software Projects: Atoms of Confusion in the Wild. In *Proceedings of the 15th International Conference on Mining Software Repositories* (Gothenburg, Sweden) *(MSR '18)*. Association for Computing Machinery, New York, NY, USA, 281–291. doi:10.1145/3196398.3196432

[12] Jonathan Hartzel, Alan Agresti, and Brian Caffo. 2001. Multinomial logit random effects models. *Statistical Modelling* 1, 2 (2001), 81–102. doi:10.1177/1471082X0100100201

[13] Donald Hedeker. 2003. A mixed-effects multinomial logistic regression model. *Statistics in medicine* 22, 9 (2003), 1433–1446. doi:10.1002/sim.1522

[14] J.D. Herbsleb and A. Mockus. 2003. An empirical study of speed and communication in globally distributed software development. *IEEE Transactions on Software Engineering* 29, 6 (2003), 481–494. doi:10.1109/TSE.2003.1205177

[15] Johannes Hofmeister, Janet Siegmund, and Daniel V. Holt. 2017. Shorter identifier names take longer to comprehend. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, New York, NY, USA, 217–227. doi:10.1109/SANER.2017.7884623

[16] Fushing Y Hsieh, Daniel A Bloch, and Michael D Larsen. 1998. A simple method of sample size calculation for linear and logistic regression. *Statistics in medicine* 17, 14 (1998), 1623–1634. doi:10.1002/(SICI)1097-0258(19980730)17:14<1623::AID-SIM871>3.0.CO;2-S

[17] Tae Kyun Kim. 2017. Understanding one-way ANOVA using conceptual figures. *Korean journal of anesthesiology* 70, 1 (2017), 22. doi:10.4097/kjae.2017.70.1.22

[18] Chris Langhout and Maurício Aniche. 2021. Atoms of Confusion in Java. In *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*. IEEE, New York, NY, USA, 25–35. doi:10.1109/ICPC52881.2021.00012

[19] Guanghan Liu and Kung-Yee Liang. 1997. Sample Size Calculations for Studies with Correlated Observations. *Biometrics* 53, 3 (1997), 937–947. doi:10.2307/2533554

[20] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E. Hassan. 2014. The Impact of Code Review Coverage and Code Review Participation on Software Quality: A Case Study of the Qt, VTK, and ITK Projects. In *Proceedings of the 11th Working Conference on Mining Software Repositories* (Hyderabad, India) *(MSR 2014)*. Association for Computing Machinery, New York, NY, USA, 192–201. doi:10.1145/2597073.2597076

[21] Wendell Mendes, Oton Pinheiro, Emanuele Santos, Lincoln Rocha, and Windson Viana. 2022. Dazed and Confused: Studying the Prevalence of Atoms of Confusion in Long-Lived Java Libraries. In *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, New York, NY, USA, 106–116. doi:10.1109/ICSME55016.2022.00018

[22] Oton Pinheiro, Lincoln Rocha, and Windson Viana. 2023. How They Relate and Leave: Understanding Atoms of Confusion in Open-Source Java Projects. In *2023 IEEE 23rd International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, New York, NY, USA, 119–130. doi:10.1109/SCAM59687.2023.00022

[23] Cristiano Politowski, Foutse Khomh, Simone Romano, Giuseppe Scanniello, Fabio Petrillo, Yann-Gaël Guéhéneuc, and Abdou Maiga. 2020. A large scale empirical study of the impact of Spaghetti Code and Blob anti-patterns on program comprehension. *Information and Software Technology* 122 (2020), 106278. doi:10.1016/j.infsof.2020.106278

[24] Hugo Quené and Huub Van den Bergh. 2008. Examples of mixed-effects modeling with crossed random effects and with binomial data. *Journal of Memory and Language* 59, 4 (2008), 413–425. doi:10.1016/j.jml.2008.02.002

[25] Guoshuai Shi, Farshad Kazemi, Michael W Godfrey, and Shane McIntosh. 2024. Reevaluating the Defect Proneness of Atoms of Confusion in Java Systems. In *Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement* (Barcelona, Spain). Association for Computing Machinery, New York, NY, USA, 154–164. doi:10.1145/3674805.3686677

[26] Guoshuai Shi, Farshad Kazemi, Shane McIntosh, and Michael W. Godfrey. 2025. Appendix for "How Does Experience Influence Developer Perceptions of Atoms of Confusion?". Zenodo. https://zenodo.org/records/18274171?token=eyJhbGciOiJIUzUxMiJ9.eyJpZCI6ImZjOTJjOWRmLWU2MTYtNGU4Ni05ODdjLWJhMWU5NmZjZmJkZCIsImRhdGEiOnt9LCJyYW5kb20iOiI3YTA1YjcxNDdmZDBmN2I0OWYxN2M1NzYwOGFkODU5NyJ9.5RABnVuyzNmCg5QjuWw9EgMWUIboXUk74YN9TBSb-C6DLdtwNBDbkfkROQYmKyqBCshGueWaSRzqengOIMNCug

[27] Noshin Tahsin, Nafis Fuad, and Abdus Satter. 2023. Prevalence of 'Atoms of Confusion' in Open Source Java Systems: An Empirical Study. (Feb. 2023). doi:10.22541/au.167570695.54470176/v1

[28] Martin K-C Yeh, Yu Yan, Yanyan Zhuang, and Lois Anne DeLong. 2022. Identifying program confusion using electroencephalogram measurements. *Behaviour & Information Technology* 41, 12 (2022), 2528–2545. doi:10.1080/0144929X.2021.1933182