Enhancing Developer Comprehension of Error Notifications through Visual Aid

Stefano Casafranca
Austin Community College
Austin, TX, USA
scasafrancal01@gmail.com

ABSTRACT

This paper investigates how software developers understand compiler error messages and proposes visual annotations to support their comprehension. The research addresses the limitations of current Integrated Development Environments (IDEs) in facilitating developers' self-explanation processes. A user study with 28 software engineering students demonstrates that the proposed annotations improve developers' ability to interpret error notifications, leading to more accurate mental models.

Author Keywords

Compiler errors; Visualization; Self-explanation; User-centered design; IDE feedback

ACM Classification Keywords

D.2.6 [Software Engineering]: Programming Environments—Integrated environments D.2.5 [Software Engineering]: Testing and Debugging—Error handling and recovery H.5.2 [Information Interfaces and Presentation (e.g., HCI)]: User Interfaces—Evaluation/methodology; Graphical user interfaces (GUI)

General Terms

Human Factors; Design; Experimentation

DOI: 10.37789/icusi.2025.6

1 INTRODUCTION

Integrated Development Environments (IDEs) such as Eclipse, IntelliJ IDEA, and Visual Studio have evolved significantly to support software development tasks through a variety of visual aids and interactive features. Among these, one of the most frequently encountered and crucial functionalities is the handling and presentation of compiler error messages. When a developer introduces an error into the code, modern IDEs typically highlight the corresponding location in the source text with red underlines, margin icons, or other visual cues. Additionally, a detailed textual error message is displayed, often in a console pane or error window, providing further context regarding the issue.

ICUSI 2025, September 18–19, 2025, Bucharest, Romania

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.37789/icusi.2025.1.1.XX

Despite these visual aids, compiler errors continue to be a source of frustration for developers, especially for novices or those unfamiliar with the particular compiler's diagnostic style. Prior research [18] has documented that many developers find these error messages cryptic, ambiguous, and difficult to interpret. One contributing factor to this confusion is that compilers generally do not make transparent the logical steps or internal reasoning that lead to the generation of the error message. From a user's standpoint, the compiler behaves like a black box that merely presents an end result without revealing how that conclusion was reached.

In typical development workflows, the burden is placed entirely on the developer to understand and fix the error. This requires a mental process that involves reverse-engineering the compiler's diagnostic pathway — a form of self-explanation [16]. Starting from the error output, developers must infer the nature of the problem, identify related parts of the codebase, trace back dependencies or misused constructs, and hypothesize plausible fixes. This backward reasoning can be cognitively demanding, time-consuming, and prone to misinterpretation, especially in complex codebases or for subtle errors. Moreover, human reasoning is often bounded by limited domain knowledge, working memory, and attention [11], which makes such tasks error-prone.

It is important to note that much of the information developers attempt to infer during this self-explanation process is already known to the compiler at the time the error is produced. The compiler, during the different stages of lexical analysis, syntax checking, semantic analysis, and code generation, accumulates a rich set of intermediate representations and internal states. These internal diagnostics form a reasoning trail that is typically discarded or hidden from the user, even though it could be harnessed to improve the user's understanding of what went wrong.

Current IDE visualizations, such as red squiggly lines and sidebar icons, are predicated on the assumption that the compiler is an opaque entity. These visual cues serve more as alerts than explanations, providing minimal support for understanding the underlying cause of errors. As a result, they are limited in their ability to guide developers through the logical chain of events that led to the error.

In this paper, we challenge this paradigm and advocate for a shift from opaque to transparent compiler diagnostics. We propose that making compiler reasoning accessible and visually explorable can significantly enhance developers' comprehension of errors. By exposing internal compiler diagnostics to the IDE, we can generate what we term *explanatory visualizations*—context-aware, structured visual cues that mirror the inferential steps developers would otherwise perform manually.

The main contributions of this work are as follows:

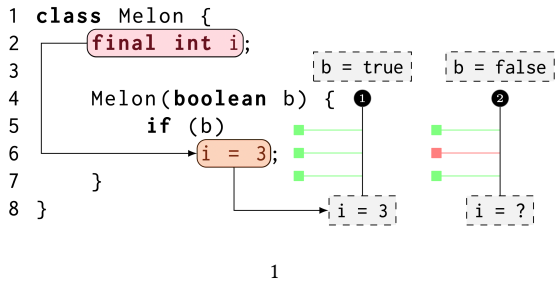

Figure 1: Comparison of uninitialized variable error shown via (a) baseline, (b) explanatory visualizations, and (c) textual message.

- We present a foundational framework of composable visual annotation types that reflect the structure of compiler reasoning. These annotations are designed to work together to tell a coherent story about the origin and nature of an error.
- We conduct a controlled *explanation task* study using paper prototypes, comparing our proposed explanatory visualizations with standard IDE visualizations. Our results show that participants using explanatory visualizations are significantly more accurate in producing correct self-explanations.
- We further evaluate the impact of explanatory visualizations on program comprehension through a *recall task*, in which participants intentionally write erroneous code within a simplified development environment. Our findings demonstrate that the additional insight afforded by explanatory visualizations enables developers to form more accurate and lasting mental models of compiler errors.

Through this work, we aim to bridge the gap between the internal logic of compilers and the external tools developers use to interact with them. By making compiler diagnostics more transparent and intelligible, we open up new possibilities for improving developer productivity, learning outcomes, and overall software quality.

Paper roadmap. Section 2 details the visual design of the error explanations and their integration into the workflow. Section 3 describes the study design, participants, tasks, and measures. Section 4 reports quantitative and qualitative results, including statistical tests for explanation quality and annotation usage. Section 5 connects the findings to mental-model theory and implications for IDE tooling, and Section 6 concludes with limitations and avenues for future work.

2 MOTIVATING EXAMPLE

To illustrate the challenges developers face when interpreting compiler error messages, consider the hypothetical case of Yoonki, a seasoned C++ programmer who is transitioning to a Java-based project. During development, Yoonki encounters a red wavy underline beneath the declaration `final int i`, as depicted. This visual indicator suggests an error, but does not immediately clarify its cause. Curious, Yoonki checks the error output in the IDE's console, which states that the variable `i` "might not have been initialized."

Initially, Yoonki is skeptical of the message. The referenced line seems harmless — it simply contains a closing curly brace. Given his past experiences in C++, where compiler warnings sometimes pointed to unrelated or misleading locations, he is inclined to dismiss the warning as a false alarm.

Relying on his prior knowledge, Yoonki draws a parallel between Java's `final` and C++'s `const` keyword, assuming that both enforce immutability from the point of declaration. Based on this understanding, he modifies the line to explicitly initialize the variable, changing it to `final int i = 3;`. However, this change introduces a new error further down in the code: `cannot assign a value to final variable i` on Line 6. Yoonki now realizes that he has violated Java's rule that a `final` variable, once assigned, cannot be reassigned.

To resolve the issue, he removes the conditional structure where the variable was being set. Although this allows the program to compile without errors, it alters the intended logic of the program. In essence, Yoonki has made a correction that satisfies the compiler but not the underlying functionality, highlighting a disconnect between the tool's feedback and the developer's mental model.

This misunderstanding arises from a subtle distinction between Java and C++: while both languages enforce a single assignment for constant variables, Java allows the assignment to occur later in the control flow, as long as it happens before the variable is accessed. Yoonki's prior mental model, effective in a C++ context, fails to accommodate this nuance, leading to what we can describe as a *knowledge breakdown* [11] — a situation where a developer's conceptual framework does not align with the behavior of the system.

The core issue is exacerbated by the IDE's limited means of communication. While the red underline and textual message are accurate in identifying a potential problem, they lack the context necessary for a deeper understanding. The IDE does not visualize the dependency between variable initialization and conditional control flow paths. It simply flags the symptom (possible uninitialized variable), without showing the underlying cause (a code path where the variable is never set).

Now imagine Yoonki working in an enhanced development environment, such as the one we propose, which incorporates explanatory visualizations. Instead of a single underline and terse message, the IDE highlights multiple relevant elements in the source code. It visually traces the control paths, showing that in one branch of the conditional (when `b = true`), the variable `i` is correctly assigned a value, but in the alternate path (when `b = false`), no assignment occurs.

This explicit representation of control flow and data dependencies enables Yoonki to recognize that the error stems not from the declaration syntax, but from incomplete conditional logic. Armed with this insight, he implements a correct fix by adding an `else` clause that assigns `i` a value when `b = false`. This change preserves the intended program logic and adheres to Java's constraints on `final` variables.

This example underscores the limitations of current error notification paradigms in IDEs, which often treat compilers as opaque entities that reveal only surface-level symptoms. Developers are left to reverse-engineer the reasoning process through trial, error, and introspection. In contrast, explanatory visualizations can make the

compiler's reasoning explicit, helping developers avoid incorrect assumptions and ultimately leading to more accurate and efficient debugging.

The broader implication is that the typical form of error reporting — consisting of an error message and a single highlighted location — is insufficient for complex reasoning tasks required during debugging. Developers benefit when tools bridge the gap between raw analysis results and actionable understanding. Our approach aims to fill this gap by making the compiler's internal reasoning more transparent and aligned with how humans naturally construct explanations.

3 PILOT STUDY

3.1 Purpose and Research Question

Before developing our visualization-based annotation approach, we carried out a small-scale exploratory study with undergraduate students in a Software Engineering course. This preliminary investigation aimed to understand how students naturally explain compiler errors to their peers and what visual strategies they employ in doing so. We sought to answer the following research question:

RQ0: What kinds of visual annotations do student developers intuitively use when explaining error messages to others?

We posited that if developers demonstrate a preference for certain types of annotations when engaged in peer-to-peer explanations, then integrating these forms of visual cues into programming environments—such as IDEs—could improve comprehension and error resolution during coding.

3.2 Study Design and Methodology

The pilot activity was carried out in a controlled classroom setting as part of regularly scheduled lab sessions. Each student participant was provided with a printed snippet of source code along with a corresponding compiler error. These snippets were intentionally stripped of any visual indicators or annotations that might bias the students' interpretive strategies.

Students worked in pairs, rotating through two roles: *explainer* and *listener*. The explainer was asked to verbally interpret the meaning of the compiler error for the listener while simultaneously marking up the printed source code with any visual elements they deemed helpful—such as arrows, text, or symbols. Importantly, they were not permitted to consult any external resources such as documentation or internet searches. After two minutes, the roles were reversed and a second, distinct example was used for the next round.

Each participant was randomly assigned one of four example tasks (T1, T2, T3, or T6), selected from real-world compiler error cases extracted from the OpenJDK 7 unit test suite for Java diagnostics. Tasks T4 and T5 were excluded from this pilot as they had not yet been curated. In total, we gathered 73 annotated samples: 17 from task T1 (23%), 12 from T2 (16%), 20 from T3 (27%), and 24 from T6 (33%).

3.3 Data Analysis

To interpret the collected annotations, we employed a two-phase qualitative coding process. First, we examined all annotated artifacts and derived a classification scheme—or taxonomy—of annotation types based on recurring patterns. This taxonomy was inductively formed from the data. Second, we applied this taxonomy to each student's annotated sheet to quantify how often each type of annotation appeared.

Table 1: Summary of Visual Annotation Types and Usage Frequency in the Pilot Study

Annotation Type	Frequency	Description
Token Highlighting	49	Marking individual code elements such as keywords, variables, or operators using circles, underlines, or boxes to draw attention to specific syntax.
Explanatory Text	45	Use of handwritten text to clarify errors, such as writing "variable not initialized" or "invalid syntax" near the relevant code.
Visual Connections	33	Drawing lines (with or without arrows) to link related parts of the code, such as a declaration and a usage, illustrating dependencies or control flow.
Symbolic Markers	20	Insertion of visual icons like question marks, crosses, or numbered markers to signal uncertainty, errors, or sequential order.
Illustrative Code Fragments	14	Writing example code or pseudo-code alongside the original code to illustrate a correction or clarify logic. These were not always syntactically correct Java, but were understandable approximations.
Strikethroughs	5	Crossing out incorrect or irrelevant code to indicate parts that should be removed or ignored. This differs from highlighting as it implies deletion.
Color Usage (Unavailable)	N/A	Use of color as a communicative aid—such as orange/red for errors and green for correct segments—was noted as a desirable but unavailable feature during the study.

3.4 Insights and Design Implications

The results from this pilot activity offer key insights into how developers instinctively visualize and externalize their understanding of program errors. Without any prior training or instruction, students gravitated toward a consistent set of visual strategies—such as highlighting tokens, drawing lines between related code, and adding short explanatory comments.

These findings suggest that such visual conventions are both intuitive and potentially useful for broader adoption in developer tooling. The prevalence of these annotations also supports the idea that embedding similar forms of visual feedback within an IDE could help developers better understand and resolve compiler errors on their own. Consequently, the taxonomy developed through this study served as a blueprint for designing the annotation types later incorporated into our visual explanation system.

Table 2: Legend for Visual Annotations

Symbol	Description
code	Marks the initial site of the compiler error.
code	Highlights related elements contributing to the error.
	Directed arrows trace logical connections or cause-effect links between code fragments.
❶	Used to sequentially number important annotations, especially those positioned away from the original source line.
?	Represents missing associations; an expected link to another code element is absent.
	Denotes conflicting or contradictory code elements.
code	Refers to synthesized code blocks introduced by the compiler or for illustration. Not part of the user's original code.
	Represents code coverage: green for executed lines, red for unexecuted or failed code paths.

4 VISUAL EXPLANATIONS FOR COMPILER ERROR MESSAGES

To improve developer comprehension and debugging efficiency, we introduce a set of eight graphical annotations to visually enhance the presentation of compiler error messages. Table 2 summarizes these annotations. We illustrate their application with a motivating.

The initial reference point in the source code is visually encoded using a —a red rectangle identifying the precise location where the error originates. This location often coincides with the site marked by standard IDEs such as IntelliJ. In our running example, the declaration `final int i` is flagged this way.

Additional related elements—such as a reassignment or associated declaration—are marked in orange using , highlighting them as contributing to the issue. The logical link between these two points is indicated with a directional arrow , suggesting an inferred control or data flow.

To further contextualize the problem, we introduce explanatory blocks, rendered as code, that replicate or reformulate the code in a simplified or pedagogical format. These do not exist in the original codebase and are designed purely to aid understanding.

In more complex cases, these blocks are integrated into composite annotations that include control-flow cues and coverage data. For instance, colored arrows like (green) and (red) illustrate whether particular branches were executed or bypassed during a program run. To differentiate between such branches, we use numeric markers like ❶ and ❷.

Through these compositional visuals, we can assert facts like: `i = 3` and all statements in branch 1 will execute if `b = true`. This allows users to identify and understand counterexamples, such as when `i` remains uninitialized due to branch 2.

While such facts could be verbally explained (e.g., `i` is uninitialized when `b = false`), we hypothesize that visualizing each step of this logic fosters deeper comprehension and improves recall for developers.

Two additional visual indicators, not used in our main example, are included here for completeness. The first is the red cross symbol;

```
class Apple {
    public public String toString() {
        return "Red";
    }
}
```

Figure 2: Red cross-out marker used for highlighting errors.

```
catch(IOException ex) { }
```

Figure 3: Visual marker representing uncertainty or a missing value.

the second is a circle with a question mark. Together, these annotations constitute a rich visual language for conveying compiler diagnostics with clarity, traceability, and pedagogical value.

The initial reference point in the source code is visually encoded using a red rectangle, identifying the precise location where the error originates. Additional related elements—such as a reassignment or associated declaration—are marked in red using code annotations, highlighting them as contributing to the issue.

5 METHODOLOGY

5.1 Research Questions

We conducted a between-subjects study with 28 participants split into control (red wavy underline) and treatment (explanatory visuals) groups. We investigated:

RQ1 Do explanatory visuals improve developer explanations?
RQ2 Do developers adopt annotation styles from visuals?
RQ3 What features differentiate explanatory from baseline visuals?
RQ4 Do better explanations reflect improved mental models?

5.2 Participants

28 third-year Software Engineering students (82% male, mean age 22, avg. 9 months industry experience) participated for course credit. Most used Eclipse and rated as Intermediate/Advanced Java developers.

5.3 Error Messages

Six representative Java compiler errors from OpenJDK tests were chosen (Table 2), focusing on clarity and potential for visual explanation.

5.4 Mockups

12 paper mockups (6 per group) simulated IDE outputs; treatment versions included explanatory annotations based on prior study and compiler knowledge. No IDE tooltips were included.

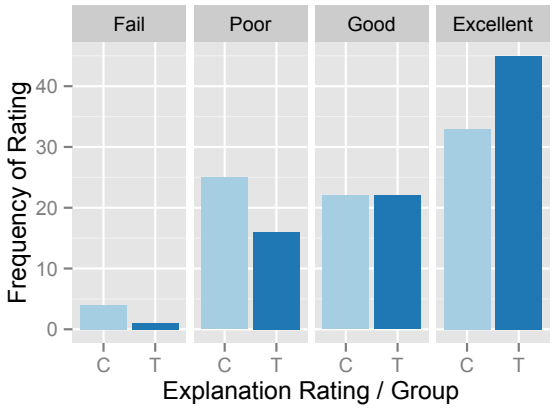

Figure 4: Ratings of explanation quality by group. Treatment > Control (Mann–Whitney $U = \underline{XXX}$, $p = \underline{0.XXX}$); {Good/Excellent} proportion differs ($\chi^2 = \underline{CCC}$, $p = \underline{0.XXX}$).

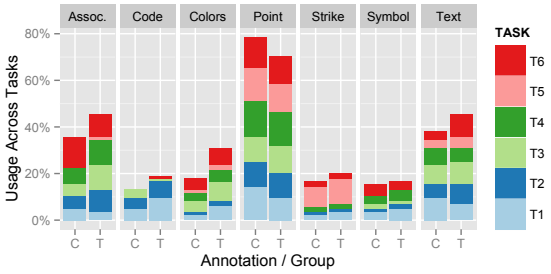

Figure 5: Annotation usage. Type distribution not different ($\chi^2 = \underline{CCC}$, $p = \underline{0.XXX}$); Treatment used more annotations per participant (Mann–Whitney $U = \underline{XXX}$, $p = \underline{0.XXX}$).

5.5 Procedure

Participants explained errors (Phase 1) and later recalled details (Phase 2). Sessions were recorded; random group assignment ensured balance.

5.6 Example Error

[style=JavaError] Zebra.java:8: error: cannot infer type arguments for BlackStripe<>; Stripe<String> sf1 = new BlackStripe<>("Marty"); $^{r}eason : inferredtypedoesnotconformtodeclaredbound(s)inferred : Stringbound(s) : Number1error$

5.7 Statistical Analysis

Unless otherwise noted, two-sided nonparametric tests were used due to the ordinal and overdispersed nature of the data. For explanation *ratings* (ordinal scale), we used the Mann–Whitney U test to compare Control vs. Treatment, reporting U, z, exact p, and effect size (r and Cliff's δ). For binary or categorical outcomes (e.g., counts of rating categories or annotation-type distributions), we used chi-square tests of independence (or Fisher's exact test when expected cell counts < 5), and report χ^2, df, p, and Cramér's V. For per-participant *annotation counts*, we used Mann–Whitney U with effect sizes as above. When multiple tests were run across annotation categories, we controlled the false discovery rate with Benjamini–Hochberg.

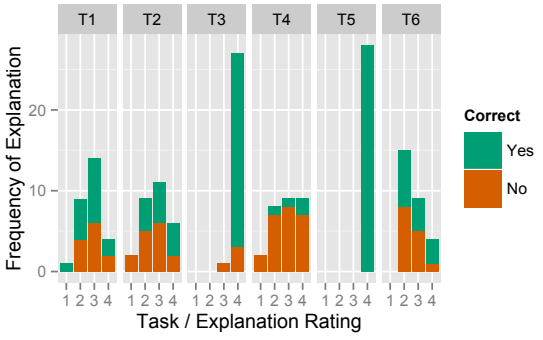

Figure 6: Explanation ratings for six tasks correlated with recall correctness, showing higher ratings lead to better recall.

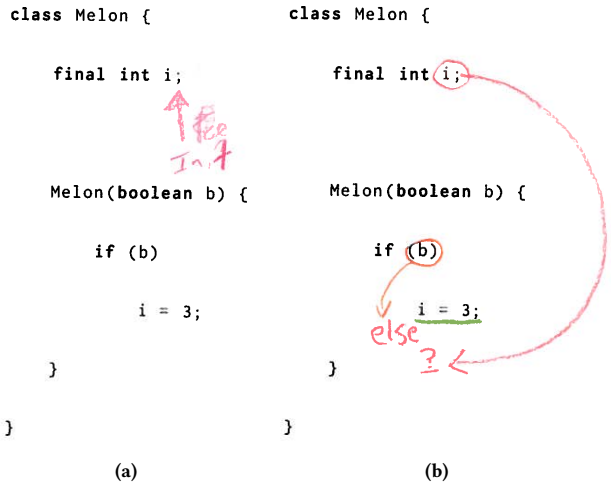

Figure 7: A contrast between visual explanations offered by (a) control group participant with explanation rating of Fail, and (b) treatment group participant with explanation rating of Excellent.

6 RESULTS

We report findings from our between-subjects user study, focusing on participants' error explanations, use of visual annotations, and recall accuracy. Results are organized according to our research questions (RQ1–RQ4).

Explanation quality Treatment explanations were rated higher than Control on the ordinal scale (Mann–Whitney $U = \underline{XXX}$, $z =$

$\underline{YYY}$, $p = \underline{0.XXX}$; effect sizes $r = \underline{RRR}$, Cliff's $\delta = \underline{DDD}$). Aggregating into {Good/Excellent} vs. {Fail/Poor}, we also observed a difference in proportions (Control $\underline{29}$% vs. Treatment $\underline{68}$%; $\chi^2(\underline{1}) = \underline{CCC}$, $p = \underline{0.XXX}$; Cramér's $V = \underline{VVV}$).

6.1 RQ1: Do Explanatory Visuals Improve Developer Explanations?

As shown in Figure 4, participants in the treatment group (explanatory visuals) produced significantly higher-rated explanations than those in the control group (baseline visuals). Specifically, 68% of the treatment group responses were rated "Good" or "Excellent," compared to only 29% in the control group. This indicates that explanatory annotations helped participants more effectively understand and communicate the causes of compiler errors.

6.2 RQ2: Do Developers Adopt Annotation Styles from Visuals?

The Figure illustrates annotation usage across tasks. While both groups used a variety of annotation types, the treatment group employed annotations more frequently across nearly all categories, including associations, explanatory text, and symbolic markers. This suggests that exposure to explanatory visualisations encouraged participants to mirror similar visual strategies in their own explanations. **Annotation usage** The distribution of annotation *types* did not differ between groups ($\chi^2(\underline{df}) = \underline{CCC}$, $p = \underline{0.XXX}$), but Treatment used *more annotations per participant* than Control (Mann–Whitney $U = \underline{XXX}$, $z = \underline{YYY}$, $p = \underline{0.XXX}$; $r = \underline{RRR}$, Cliff's $\delta = \underline{DDD}$). Across individual annotation categories (e.g., highlights, callouts), no test survived FDR correction ($q < .05$).

6.3 RQ3: What Features Differentiate Explanatory from Baseline Visuals?

Cognitive Dimensions Questionnaire results revealed a statistically significant difference in the *Hidden Dependencies* dimension (median = 4 for treatment, 3 for control, $p = 0.008$). This suggests that explanatory visualizations enhanced participants' ability to identify and reason about underlying relationships in code. Notably, participants exposed to color-enhanced explanatory visualizations showed observable improvements in understanding hidden code relationships and mental operations. While statistical significance was strongest in the Hidden Dependencies dimension, revised visual cues, such as clearer color differentiation, contributed to improved reasoning in more complex tasks.

6.4 RQ4: Do Better Explanations Reflect Improved Mental Models?

Participants who produced higher-quality explanations, particularly those rated as "Good" or "Excellent," demonstrated notably better performance in the recall tasks. In contrast, those with lower-rated explanations, such as "Fail," showed considerably weaker recall accuracy. These observations reinforce the idea that strong self-explanatory abilities are linked to the development of more accurate mental models of compiler behavior.

7 THREATS TO VALIDITY

Our study, like any empirical investigation, is subject to several potential threats to validity. These threats arise from limitations in the experimental design, the scope of the tasks, participant characteristics, and the generalizability of our findings. We discuss these concerns in detail below to contextualize our results and guide future research.

7.1 External Validity

One major concern pertains to the external validity, or the extent to which our findings generalize to real-world programming scenarios. In practical software development, developers often encounter error messages within large and complex codebases that span multiple files and contain a mixture of both functional and erroneous code. By contrast, our experimental tasks were designed to isolate the specific lines of code directly responsible for generating an error, and all code was contained within a single source file. This simplified setting may not fully capture the cognitive challenges developers face when diagnosing errors across distributed codebases or in the presence of intricate dependencies.

Furthermore, our study applied explanatory visualizations to only six carefully selected programming tasks that fit entirely on a single screen. These tasks were restricted to Java and represented a narrow subset of possible error scenarios. It remains an open question whether similar visualization techniques can be effectively scaled to a broader and more diverse set of error messages, especially in other programming languages or in larger, more complex codebases. Thus, while our results are promising, we caution against overgeneralizing them beyond the specific contexts studied.

7.2 Construct Validity

Construct validity concerns whether the measurements and instruments used in the study truly capture the constructs of interest. We observed an interesting divergence in our data: participants in the treatment group, who received explanatory visualizations, rated their ability to explain the errors significantly higher than those in the control group. However, this increased confidence did not consistently translate into improved correctness during recall tasks.

We hypothesize that this discrepancy may be due to partial understanding; participants could formulate plausible explanations without fully internalizing the underlying compiler reasoning or error semantics. This incomplete mental model might have limited their ability to accurately recall specific details or resolve related problems. Additionally, several participants struggled with syntax-related issues, occasionally introducing secondary compiler errors that were unrelated to the primary recall task. These distractions likely affected performance and highlight the complexity of disentangling understanding from practical coding ability.

Another construct validity threat stems from the experimental protocol itself. The use of a think-aloud procedure, while essential for capturing participants' explanations and reasoning processes, may have inadvertently enhanced self-explanation abilities across both groups. This effect could have diminished the observable differences between the treatment and control conditions, particularly in recall accuracy.

Recent refinements in the visual annotation system, including the use of distinct and intuitive color schemes, have reduced the cognitive overhead associated with unfamiliar representations. As a result, participants were able to engage more readily with the visualizations and derive meaningful insights, particularly in reasoning about complex code paths and error propagation.

7.3 Ecological Validity

Ecological validity refers to how closely the experimental setting and materials approximate real-world conditions. Although we strove to simulate realistic debugging tasks, the controlled laboratory environment inherently differs from natural programming contexts, where developers have access to extensive documentation, debugging tools, version control history, and collaborative support.

Participants completed tasks in isolation without the usual interruptions, resource consultations, or social interactions characteristic of professional software development. Additionally, the constrained time limits and task framing may have influenced participants' cognitive strategies in ways that differ from everyday practices.

Future studies would benefit from deploying explanatory visualizations in authentic development environments over extended periods to better understand their practical utility and long-term effects on developer cognition and workflow.

7.4 Summary

While these threats do not invalidate the positive trends observed in our study, they underscore the need for cautious interpretation and further research. Addressing these limitations through more ecologically valid experimental designs, broader task sets, longitudinal studies, and integration with real-world development tools will be essential to fully establish the efficacy and generalizability of compiler visualization techniques for error comprehension.

8 RELATED WORK

The concept of self-explanation as a powerful cognitive strategy to enhance learning and comprehension has been extensively validated in the context of human-computer interaction and educational psychology. Lim et al. [12] provided foundational insights by empirically demonstrating that when users articulate reasons behind a system's behaviors or outputs, their understanding deepens and their trust in the system increases significantly. This effect stems from the active engagement required during self-explanation, which helps users internalize complex system logic and fosters more accurate mental models. Building on this, Ainsworth and Loizou [1] highlighted the critical role of visual representations, showing that diagrams and graphical depictions not only facilitate but also amplify the benefits of self-explanation compared to purely textual explanations. Their findings underscore the cognitive advantages of externalizing abstract information visually, enabling users to form clearer and more organized conceptual structures.

Parallel research streams have targeted the problem of improving error notification comprehension, which remains a persistent challenge in programming environments. Jeffery's development of the *Merr* tool [9] illustrates a pragmatic approach to this issue by intercepting and augmenting the error handling process of the LR parser generator within compilers. This enhancement allows for more context-aware and informative syntax error messages, helping developers diagnose problems more effectively. Kantorowitz and Laor [10] similarly proposed modifications at the parser generation level to reduce the ambiguity and improve the clarity of error reports. Despite these advancements, research indicates that even highly detailed and accurate textual error messages may not sufficiently improve programmer comprehension [5, 15]. This suggests that alternative modalities—particularly visual or interactive representations—might be better suited to convey the complex semantics behind compiler errors, addressing limitations inherent to textual feedback.

Additionally, Hartmann et al. [8] proposed a social recommendation system that leverages community knowledge by showcasing how other developers interpret and resolve specific errors. This approach externalizes the troubleshooting process, providing developers with curated examples and strategies derived from collective experience. While this method taps into the social dimension of software development, our approach takes a different stance by advocating for the compiler itself to expose its internal reasoning processes directly through visualizations. By enabling developers to peer "under the hood" of the compiler, our method aims to provide a more fundamental and intrinsic understanding, rather than relying solely on external guidance or heuristics.

Complementing these perspectives are efforts focused on enhancing the precision and reliability of compiler diagnostics to reduce false positives and improve overall error detection [2–4]. These improvements are crucial because the efficacy of any visualization tool hinges on the quality of the underlying data. By combining accurate diagnostics with rich, interactive visualizations, there is potential to create synergistic effects that further empower developers and streamline debugging workflows.

8.1 AI Coding Assistants and Diagnostic Explanations

Modern IDEs increasingly embed conversational assistants for code and diagnostics. GitHub Copilot Chat in Visual Studio can analyze exceptions and errors in context and propose fixes directly within the IDE [13, 14]. Google's Gemini also exposes "explain and fix errors" capabilities (e.g., in Colab and Chrome DevTools) that summarize error causes and suggest remediations [6, 7]. While these systems primarily deliver *textual* rationales, our approach contributes complementary *visual* structure: it externalizes the error's semantics (what failed), scope (where), and dependencies (what else is implicated) at a glance, which can lower cognitive load, support scanning and comparison across attempts, and make hidden dependencies actionable inside the IDE. Visual diagnostics can therefore serve as an intermediate representation that complements LLM narratives by anchoring them to consistent, inspectable visual elements, potentially improving trust and comprehension for time-pressured debugging workflows.

9 FUTURE WORK

Looking ahead, there are multiple avenues for expanding the scope and impact of this research. At the forefront is the technical feasibility of capturing and representing the complex and voluminous

analysis traces that compilers generate during program compilation. While it is well established that modern compilers produce extensive internal data, from syntax trees to semantic analyses and optimization steps, a critical open question remains: which subsets of this data are most salient and beneficial to developers seeking to understand error messages and program behavior? Addressing this requires not only novel algorithms and instrumentation techniques to efficiently extract, filter, and encode compiler state information but also careful human-centered design to ensure that visualizations convey relevant insights without overwhelming users.

One promising research strategy is to prototype visualization techniques within simplified language environments or pedagogical compiler implementations, such as MiniJava [17]. These constrained settings offer controlled complexity, facilitating iterative design and evaluation of visualization concepts before scaling to full-featured compilers. Through such prototypes, researchers can experiment with different visual metaphors, interaction paradigms, and trace representations, gaining valuable feedback from users and refining approaches accordingly.

In parallel, a thorough empirical investigation into the taxonomy of compiler notifications is warranted. Not all notifications serve the same function or possess the same cognitive demands. Some errors may stem from simple syntax mistakes, while others reflect deeper semantic or logical inconsistencies. Developing a systematic categorization of notifications—based on their causes, complexity, frequency, and potential for visualization—could inform tailored visualization strategies that optimize clarity and cognitive load. This taxonomy could also guide the prioritization of visualization efforts toward notifications where the expected benefit is greatest.

Beyond notification categorization, future work should explore the integration of visualization tools into real-world integrated development environments (IDEs) and the impact on developer workflows. User studies involving professional programmers and students could reveal how visual explanations influence debugging efficiency, error resolution accuracy, and knowledge retention over time. Longitudinal studies would help assess how these tools affect developers' mental models and whether they foster deeper, more transferable understanding of compiler behavior.

Additionally, exploring multimodal approaches that combine visualizations with textual explanations, interactive exploration, or even voice-assisted guidance could create more holistic and adaptive support systems. Investigating how these modalities complement each other and how developers prefer to access information during different debugging scenarios would provide actionable insights for tool designers.

Finally, as compiler technologies evolve and become more sophisticated, there is an opportunity to leverage emerging techniques in machine learning and artificial intelligence to dynamically tailor visual explanations to individual developer expertise and context. Adaptive visualizations that learn from user interactions and feedback could further enhance the usability and effectiveness of debugging tools, making compiler feedback not only more transparent but also more personalized. Recent advances in AI-assisted debugging tools have begun to explore similar territories, leveraging large language models and deep learning for error localization and fix suggestions [15], [16]. Integrating such approaches with visualization-based reasoning remains an exciting frontier for future research.

10 CONCLUSION

This paper has presented a compelling vision for enhancing developer comprehension and problem-solving through the visualization of opaque compiler reasoning processes. By exposing internal compiler states and decision-making pathways in visually intuitive formats, we facilitate developers' ability to self-explain and internalize the causes of errors, leading to deeper understanding and increased trust in the development environment. Our findings suggest that when visualizations align well with developer expectations and cognitive frameworks, they are more frequently utilized and lead to improved mental models that support more effective debugging and learning.

We posit that the inherently diagrammatic methods developers naturally employ when communicating about programming problems—both in collaboration with peers and in personal cognition—provide a rich source of inspiration for designing next-generation integrated development environments. By embedding these diagrammatic communication principles directly into compiler feedback mechanisms, IDEs can transcend traditional text-heavy error reporting and foster more engaging, interactive, and insightful experiences.

The intersection of compiler internals, cognitive science, and visualization technology represents an exciting frontier for software engineering research. We believe that continued exploration in this space will yield tools that not only reduce the cognitive burden of debugging but also cultivate a deeper synergy between human developers and the compilers that underpin their work. Ultimately, this integration promises to enhance software quality, accelerate development cycles, and empower developers to tackle increasingly complex programming challenges with confidence and clarity.

REFERENCES

[1] S. Ainsworth and A. T. Loizou. 2003. The effects of self-explaining when learning with text or diagrams. *Cognitive Science* 27, 4 (Aug. 2003), 669–681.

[2] N. Boustani and J. Hage. 2011. Improving type error messages for generic Java. *Higher-Order and Symbolic Computation* 24, 1-2 (Jun. 2011), 3–39.

[3] J. C. Campbell, A. Hindle, and J. N. Amaral. 2014. Syntax errors just aren't natural: Improving error reporting with language models. In *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR)*. 252–261.

[4] S. Chen, M. Erwig, and K. Smeltzer. 2014. Let's hear both sides: On combining type-error reporting tools. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 145–152.

[5] P. Denny, A. Luxton-Reilly, and D. Carpenter. 2014. Enhancing syntax error messages appears ineffectual. In *Proceedings of the 2014 Conference on Innovation and Technology in Computer Science Education (ITiCSE)*. 273–278.

[6] Google. 2024. *Explain and Fix Errors in Colab.* https://colab.research.google.com/ Accessed: 2025-08-26.

[7] Google. 2024. *Gemini in Chrome DevTools.* https://developer.chrome.com/docs/devtools/ Accessed: 2025-08-26.

[8] B. Hartmann, D. MacDougall, J. Brandt, and S. R. Klemmer. 2010. What would other programmers do. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI)*. 1019–1028.

[9] C. L. Jeffery. 2003. Generating LR syntax error messages from examples. *ACM Transactions on Programming Languages and Systems* 25, 5 (Sep. 2003), 631–640.

[10] E. Kantorowitz and H. Laor. 1986. Automatic generation of useful syntax error messages. *Software: Practice and Experience* 16, 7 (Jul. 1986), 627–640.

[11] A. J. Ko and B. A. Myers. 2005. A framework and methodology for studying the causes of software errors in programming systems. *Journal of Visual Languages & Computing* 16, 1 (2005), 41–84.

[12] B. Y. Lim, A. K. Dey, and D. Avrahami. 2009. Why and why not explanations improve the intelligibility of context-aware intelligent systems. In *Proceedings of*

the SIGCHI Conference on Human Factors in Computing Systems (CHI). 2119–2129.

[13] Microsoft. 2023. *GitHub Copilot Chat in Visual Studio.* https://learn.microsoft.com/en-us/visualstudio/ide/copilot-chat Accessed: 2025-08-26.

[14] Microsoft. 2023. *Visual Studio IDE Features.* https://visualstudio.microsoft.com/vs/features/ Accessed: 2025-08-26.

[15] M.-H. Nienaltowski, M. Pedroni, and B. Meyer. 2008. Compiler error messages: What can help novices?. In *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education*. 168–172.

[16] C. Parnin. 2011. Subvocalization - Toward hearing the inner thoughts of developers. In *Proceedings of the IEEE International Conference on Program Comprehension (ICPC)*. 197–200.

[17] E. Roberts. 2001. An overview of MiniJava. In *Proceedings of the 32nd SIGCSE Technical Symposium on Computer Science Education*, Vol. 33. 1–5.

[18] V. J. Traver. 2010. On compiler error messages: What they say and what they mean. *Advances in Human-Computer Interaction* 2010 (2010), 1–26.