TU Delft

Delft University of Technology

Document Version
Final published version

Licence
CC BY

Citation (APA)
Koohestani, R., Sergeyuk, A., Gros, D., Spiess, C., Titov, S., Devanbu, P., & Izadi, M. (2026). Does In-IDE Calibration of Large Language Models work at Scale? In S. H. Tan, & F. Khomh (Eds.), *FSE Companion 2026 - Proceedings of the 34th ACM International Conference on the Foundations of Software Engineering* (pp. 609-619). Association for Computing Machinery (ACM). https://doi.org/10.1145/3803437.3805234

Important note
To cite this publication, please use the final published version (if applicable).
Please check the document version above.

Copyright
In case the licence states “Dutch Copyright Act (Article 25fa)”, this publication was made available Green Open Access via the TU Delft Institutional Repository pursuant to Dutch Copyright Act (Article 25fa, the Taverne amendment). This provision does not affect copyright ownership.
Unless copyright is transferred by contract or statute, it remains with the copyright holder.

Sharing and reuse
Other than for strictly personal use, it is not permitted to download, forward or distribute the text or part of it, without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license such as Creative Commons.

Takedown policy
Please contact us and provide details if you believe this document breaches copyrights.
We will remove access to the work immediately and investigate your claim.

This work is downloaded from Delft University of Technology.

Does In-IDE Calibration of Large Language Models work at Scale?

Roham Koohestani[*†]
rkoohestani@tudelft.nl
Delft University of Technology
Delft, The Netherlands

Agnia Sergeyuk[*]
agnia.sergeyuk@jetbrains.com
JetBrains Research
Belgrade, Serbia

David Gros
dgros@ucdavis.edu
University of California, Davis
Davis, California, USA

Claudio Spiess
cvspiess@ucdavis.edu
University of California, Davis
Davis, California, USA

Sergey Titov
sergey.titov@jetbrains.com
JetBrains Research
Amsterdam, The Netherlands

Premkumar Devanbu
ptdevanbu@ucdavis.edu
University of California, Davis
Davis, California, USA

Maliheh Izadi
m.izadi@tudelft.nl
Delft University of Technology
Delft, The Netherlands

Abstract

Code assistants powered by large language models are now embedded in integrated development environments, yet developers lack reliable signals for when to trust generated code. Model confidence could serve as a signal, but only if it accurately reflects the likelihood of acceptance. Post-hoc calibration aims to achieve this alignment, though its efficacy in production settings remains understudied. We investigate in-IDE confidence calibration from two perspectives: (1) scalable methods for calibrating confidence signals and (2) interface design for communicating reliability to developers. We introduce a **flexible calibration framework** for open-source models and evaluate calibration against developer acceptance behavior using over **24 million real-world IDE interactions** across multiple languages. We find that a general Platt-scaling calibrator *does not*, consistently improve the usefulness of confidence as a reliability signal, while personalized calibration can help when sufficient user interaction data is available. Complementing this, a multi-phase design study with expert designers and **153 professional developers** indicates a preference for non-numerical, color-coded reliability indicators embedded in the in-editor generation workflow.

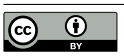
CCS Concepts

• **Software and its engineering** → **Integrated and visual development environments**; • **Human-centered computing** → *Human computer interaction (HCI).*

Keywords

LLM calibration, code completion, confidence measure, human-AI interaction, empirical study

[*]Both authors contributed equally to this research.
[†]Roham Koohestani is also affiliated with the JetBrains Research.



This work is licensed under a Creative Commons Attribution 4.0 International License.
FSE Companion '26, Montreal, QC, Canada
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2636-1/2026/07
https://doi.org/10.1145/3803437.3805234

ACM Reference Format:
Roham Koohestani, Agnia Sergeyuk, David Gros, Claudio Spiess, Sergey Titov, Premkumar Devanbu, and Maliheh Izadi. 2026. Does In-IDE Calibration of Large Language Models work at Scale?. In *34th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE Companion '26), July 05–09, 2026, Montreal, QC, Canada.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3803437.3805234

1 Introduction

Modern software development increasingly relies on code assistants powered by Large Language Models (LLMs). Tools like GitHub Copilot [3], JetBrains Junie [11], and others have become increasingly sophisticated assistants, capable of generating everything from single lines of code to entire codebases and developer workflows [26]. While studies confirm that these tools have increased developers' productivity [6, 31] and have thereby shortened development cycles, they also introduce several new concerns. These models, despite their impressive capabilities, are fallible and can generate unreliable code that is subtly incorrect, insecure, or inefficient [7, 19]. Developers, therefore, face a dilemma. They benefit from assistance from Artificial Intelligence (AI)-based tools, but they cannot blindly trust every suggestion.

To navigate this dilemma, developers need a reliable signal of when to trust generated code. Model confidence could serve this purpose. In our work, *model confidence* refers to the internal scores that a model assigns to its predictions, typically derived from token probabilities. If confidence accurately reflected the likelihood of correctness, developers could use it to guide their acceptance decisions. Unfortunately, LLMs are poorly calibrated out of the box: they often express high confidence in erroneous generations [28, 33]. This miscalibration is particularly acute in code generation, where studies show that generative code models require post-hoc adjustment to achieve reasonable calibration [25].

Post-hoc calibration offers a potential remedy. *Calibrated confidence* refers to model confidence that has been adjusted so that expressed confidence corresponds to actual correctness likelihood. A model is considered well-calibrated when a prediction assigned 80% confidence is correct approximately 80% of the time. If calibration succeeds, confidence becomes a useful reliability signal. If

it fails, developers will be left without guidance. Despite this importance, evidence of calibration effectiveness in production-level integrated development environments remains limited.

We argue that calibration is not only a statistical alignment problem, but a core requirement for making AI assistants useful and trustworthy. Therefore, this paper examines the calibration of code LLMs' confidence in production IDE settings by combining technical and human–AI user-experience perspectives. We study calibration in real-world IDEs owned by JetBrains[1] from two linked angles: (1) the *technical* approaches for implementing confidence calibration and enhancing model reliability, and (2) the *human-centered design* strategies for conveying reliability cues to developers. Based on this, we formulate three Research Questions (RQs).

- **RQ1.** Does calibrating confidence estimates better correlation (than raw confidence) with behavioral measures derived from telemetry in the wild?
- **RQ2.** Does person-specific or project-specific model confidence calibration yield higher absolute correlation with behavioral measures?
- **RQ3.** How do developers expect a reliability signal to be presented inside the IDE?

To address RQ1 and RQ2, we conduct a large-scale analysis of code completion interactions inside the production-level IDEs. We analyze over **24 million** interaction records from more than **750,000 unique devices** across a **six-month** period. We additionally develop **Calibrate-CC**, a scalable and reusable framework that can be used to obtain calibration weights for any open-source model using any dataset.

Our results show that post-hoc calibration generally *does not* improve the reliability of model confidence scores (RQ1). Furthermore, while personalized calibration for specific users or projects *can* offer modest additional gains, we find that its effectiveness is highly dependent on the volume of interaction data, primarily benefiting the most active users (RQ2).

Given that calibration alone does not reliably improve confidence signals, effective communication of whatever reliability information exists becomes essential. To address RQ3, we conduct a multi-phase design study combining scenario-based design, semi-structured interviews, and survey validation with 3 expert designers and **153 professional developers**. We find that developers prefer reliability signals communicated through non-numerical, color-coded indicators integrated directly into the in-editor code generation workflow.

In summary, the main contributions of our paper are:

- a wide-scale in-the-wild evaluation of calibration frameworks for LLMs in the context of code completion.
- A scalable and flexible framework for model calibration based on commonly used libraries.
- A comprehensive user study on the needs and design of reliability indicators in a software development environment.

In the sections to follow, the background is presented in section 2, followed by problem formulation in section 3. Research questions are addressed in section 4, section 5, and section 6. We discuss implications and limitations in section 7 and conclude in section 8.

[1]JetBrains: https://www.jetbrains.com/; a major software vendor specializing in the creation of intelligent development tools.

The replication package [13], along with the CalibrateCC code, can be found in https://doi.org/10.5281/zenodo.18231197.

2 Background & Related Work

LLMs are increasingly used in IDEs to generate code suggestions and perform other software development tasks (*e.g.*, Cursor [4], JetBrains Junie [11], Windsurf [32]). While these tools promise productivity gains, their impact depends on the quality of suggestions that developers accept. Studies show that programmers working with AI assistants can unintentionally introduce insecure or incorrect code, sometimes with increased confidence in their outputs [8, 21]. This risk highlights the need for reliable signals that guide developers' acceptance and review decisions.

Uncertainty in Code Generation. When using generative models for code completion, the model's confidence, computed as token probability, is often taken to be a proxy for likely correctness and is used when generating suggestions. However in code generation, low probability may reflect multiple valid alternatives rather than incorrectness, and high probability can still coincide with errors. Vasconcelos et al. [30] show that highlighting row low-probability tokens does not improve outcomes, while edit-based confidence models better align with developer needs. Other work similarly reports that raw model confidence is often miscalibrated, requiring improvement before it can serve as a useful guide [14, 25].

Calibration Techniques. Calibration methods aim to align predicted probabilities with observed correctness [25], and metrics such as expected calibration error and calibration curves are used to evaluate this alignment [27]. Calibrated confidence scores provide developers with more reliable information for deciding when to accept or review suggestions. Complementary research also proposes lightweight intrinsic measures such as perplexity and entropy as indicators of correctness or hallucination in generated code [14].

UI for AI Reliability. Equally important is how reliability information is presented in the IDE. Human–AI interaction studies show that interface choices strongly influence trust and reliance [1, 15, 18]. Developers request confidence indicators that are granular but not overwhelming, such as highlights or compact visual cues [29, 30]. Explanations, sources, and consistency cues can further affect reliance: explanations often increase reliance on both correct and incorrect outputs, sources help reduce overreliance, and inconsistencies can serve as warnings [12]. Interactive approaches such as exposing local alternatives at points of model uncertainty have been shown to reduce vulnerabilities and give developers more control [8]. Finally, confidence signals interact with human judgment. Recent work shows that developers' own self-confidence can align with AI confidence, sometimes persisting beyond the interaction [16]. Without careful design, this alignment can lead to miscalibration on the human side. These findings emphasize that confidence is not only a model property but also a feature of the joint human–AI system in the IDE.

In summary, prior work shows that raw token probabilities are not well-correlated with test-passing correctness, that calibrated confidence measures provide better signals, and that interface design shapes how such signals affect usage. What is missing are

empirical studies that connect calibrated confidence to actual acceptance behavior in real-world IDE workflows and that compare alternative user interface designs for presenting these signals. Our study addresses this gap.

3 Defining the Problem

To analyze the calibration of code completion models, we must establish a formal framework that defines the core components of the developer-AI interaction. This framework provides the foundation for our research questions and subsequent analysis.

Let the state of the IDE at the moment a code completion is triggered be the **context**, denoted as C. A code generation model, M, takes C and produces a **suggestion**, S, which is a sequence of tokens $(t_1, t_2, \ldots, t_k)$ intended to complete the developer's work.

Internally, the model M computes a score for the suggestion S, which we refer to as the raw **model confidence**, $conf(S, C)$. While this function can be implemented in various ways, in this work, we define it as the average log-probability of the generated tokens, which is equivalent to the geometric mean of their probabilities. This is formally expressed as:

$$conf(S, C) = exp(\frac{1}{k} \sum_{i=1}^{k} \log P(t_i | C, t_1, \ldots, t_{i-1}))$$

The goal is to assess whether the model's confidence score reflects the empirical reliability of its suggestion. We define an **outcome variable** $Y \in [0, 1]$, which quantifies the realized quality or correctness of a suggestion. Depending on the evaluation setting, Y may be binary (e.g, accepted/rejected) or continuous (e.g., normalized similarity of suggested code to code as finally used)

A model achieves **perfect calibration** when, for each confidence value $p \in [0, 1]$ the expected result is equal to the reported confidence. Mathematically, $E[Y \mid conf(S, C) = p] = p$

A model is said to be **miscalibrated** when this condition is not met, that is, when its expressed confidence is a poor estimator of the expected outcome. Modern LLMs are notoriously miscalibrated and often exhibit systematic overconfidence ($E[Y] < p$) or underconfidence ($E[Y] > p$); this was found to be the case when Y indicates "test-passing correctness" [25].

One of our goals here is to learn **calibration model** (or calibration function), denoted as f. This model transforms the raw, uncalibrated model confidence into a new, calibrated probability, $\hat{p} = f(conf(S, C))$.

The function f is trained on a dataset of suggestions and their observed outcomes (Y) so that the resulting $\hat{p}$ provides a more accurate estimate of the true utility. In other words, the objective is for $\hat{p}$ to be as close to a perfectly calibrated estimate as possible, such that $E[Y|\hat{p} = p] \approx p$.

This formal framework allows us to articulate our research goals:

- **RQ1** investigates whether the calibrated probability $\hat{p}$ increased the reliability of confidence signals by having a stronger correlation with the observed developer behavior (our proxy for Y) than the raw confidence $conf(S, C)$.
- **RQ2** explores whether the calibration model f becomes more effective when it is specialized, that is, trained on data partitioned by a specific person or project.
- **RQ3** examines how the reliability signal in the form of calibrated probability $\hat{p}$ can be most effectively communicated to developers within the IDE.

4 RQ 1.

In our first research question, we investigate the feasibility of using post-hoc calibration on code completion models. Specifically, our objective is to determine (1) if a calibrated confidence score provides a more reliable signal of suggestion utility than the model's raw confidence, and (2) whether specializing the calibration model to a specific usage context yields further improvements.

4.1 Method

To systematically address this question, we developed **CalibraTeCC**, a scalable and extensible Python framework for evaluating and calibrating code completion models. Our framework, developed with the Hugging Face *transformers* and *datasets* libraries, is intentionally designed to be model- and data-independent. This decision ensures that it can integrate with any open source decoder-based language model and a wide range of code datasets. It supports configurable masking policies, generation parameters, and evaluation strategies, enabling reproducible research into model calibration.

4.1.1 Data Generation for Calibration. The first step in our methodology is to generate a dataset suitable for training a calibration model. This process involves creating pairs of model confidence scores and their corresponding ground-truth utility labels $(conf(S, C), Y)$. Our framework automates this pipeline as follows:

(1) **Dataset Loading:** We gather code examples from an internal dataset specifically tailored for code completion. This dataset is decontaminated with respect to the model's training data. Although this dataset includes examples from multiple programming languages, our research emphasizes the subsets related to the **Java**, **Python**, and **Kotlin** languages. The selection of these languages is motivated due to their wide usage, their distinct programming paradigms (dynamically versus statically-typed), as well as an organizational interest in their results. As these languages cover a significant portion of current industry usage in modern development, our findings can have broader generalizability across different workflows. Each sample in the dataset can be seen as a 4 tuple (*prefix*, *suffix*, *middle*, *multifileContext*) where the objective is to predict the middle based on the prefix, suffix, and multi-file context[2]. As per the definition in section 3, these fields are combined into a unified piece of context $ctx \in C$ and a suggestion $s \in S$ is generated by the model M. Table 1 presents details about the data used.
(2) **Masking and Prompting:** For each code example, a masking policy is applied to create a realistic code completion scenario. For instance, the *RandomLineSpanPolicy* selects a span of lines to be the *ground truth* completion and splits the remaining source code into a prefix (C_{prefix}) and a suffix (C_{suffix}). This pair forms the context C for a fill-in-the-middle (FIM) prompt.
(3) **Generation and Confidence Extraction:** The context C is passed to the base model M to generate a suggestion S. During

[2]This is a set consisting of three additional files, each containing code related to the file where the current generation is being developed.

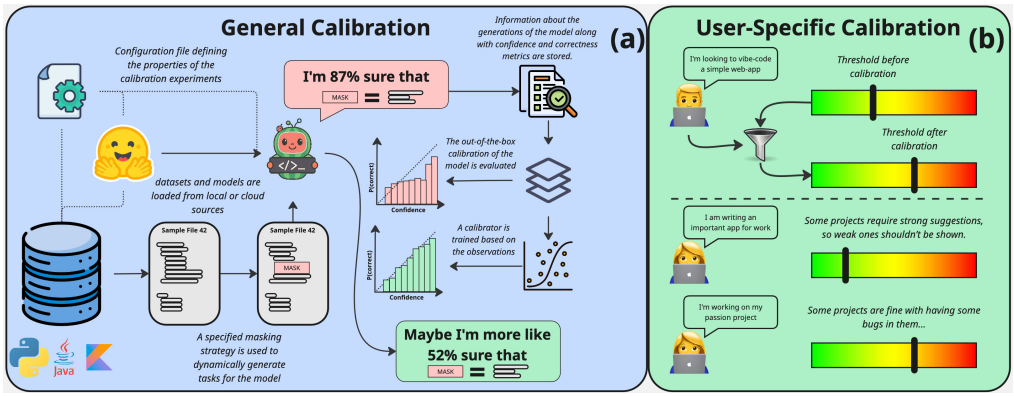


Figure 1: (a) In general calibration, a calibrator is trained on the model's confidence and correctness to get more reliable estimates. (b) In the user-specific stage, the calibrator is adapted to match user- and project-specific needs.

Table 1: Dataset statistics (tokens using the Mellum4B tokenizer; mean ± std.).

Language	Metric	Value
Java	Unique examples	6151
	Context length/example	6788.53 ± 101.59
	Masked span length	64.54 ± 8.48
Python	Unique examples	6900
	Context length/example	12901.29 ± 192.26
	Masked span length	61.02 ± 8.93
Kotlin	Unique examples	6988
	Context length/example	5874.25 ± 84.76
	Masked span length	69.97 ± 9.12
Total	Unique examples	20039
	Context length/example	8574.50 ± 151.81
	Masked span length	65.22 ± 8.87

this process, we record the sequence of token log-probabilities. As formally defined in section 3, we calculate the raw model confidence, $conf(S, C)$, as the average log-probability of the generated tokens; equivalent to the geometric mean of the token probabilities in linear space. To minimize variability introduced by stochastic decoding strategies, we utilized greedy decoding (temperature=0) without sampling. We set the maximum new tokens to 64, consistent with prior literature on single-line completion [5], and defined stop sequences using sentinel tokens from the training data and standard end-of-line delimiters.

(4) **Evaluation and Labeling:** The generated suggestion S is evaluated against the ground truth to produce the outcome variable Y. This evaluation is our ground-truth labels.

4.1.2 Defining Ground Truth Labels (Y). When correlating with real-world developer interactions in our main analysis, we define Y as the **preserved ratio (PR)** of the suggestion. This continuous metric, valued in $[0, 1]$, measures the ratio of similarity between the provided completion and the code eventually included in the code by the developer. Assume that the expected completion is denoted by $middle$, and the generated completion by $\widehat{middle}$. The PR is

$$PR = 1 - \frac{L(middle, \widehat{middle})}{max(|\ middle\ |, |\ \widehat{middle}\ |)}$$

where $L(s_1, s_2)$ denotes the Levenshtein distance. The underlying assumption in choosing this metric instead of **Exact Match** is that a developer's likelihood of accepting a suggestion increases with the degree to which the suggestion resembles the intended completion.

4.1.3 Training Calibration Models. With a generated dataset of $(conf(S, C), Y)$ pairs, we can train the calibration model f, as formalized in section 3. For this study, similar to previous work by Spiess et al. [25], we use Platt Scaling [22], a parametric method suited for correcting miscalibrations by fitting a logistic regression model.

While more complex non-parametric methods exist (e.g., Isotonic Regression or Dirichlet calibration), we prioritized Platt Scaling for three reasons: (1) Scalability: Our analysis involves over 24 million interaction records, which requires a lightweight solution that is computationally efficient at scale; (2) Continuous Targets: Methods like Dirichlet calibration are designed for multiclass classification and are ill-suited for our continuous "preserved ratio" target variable; and (3) Robustness: Given that our data exhibits both over- and under-confidence across different contexts, simple parametric updates provide a stable global correction without the high variance or computational cost associated with instance-wise methods.

We train two different types of calibrators to address RQ1:

- **General Calibrator ($f_{general}$):** This model is trained on the entire dataset, $\mathcal{D}_{all}$, which aggregates all observations across all programming languages. The goal is to learn a single, universal calibration function that improves confidence estimates regardless of the language context.
- **Language-Specific Calibrators (f_{lang}):** To test the hypothesis that calibration benefits from specialization, we train separate models for each language. For a given language L (e.g., Python), we first create a dataset $\mathcal{D}_L = \{(conf_i, Y_i) \in \mathcal{D}_{all} | \text{language}_i = L\}$. We then train a dedicated calibrator, f_L, on this subset.

4.1.4 Evaluation of Calibration. To evaluate the performance of our calibration models, we use standard metrics designed to quantify the alignment between the predicted calibrated probabilities ($\hat{p}$) and the observed outcomes (Y) on a held-out set of N samples. Similar to previous work [9, 20, 25], we have selected these metrics to provide a holistic assessment of the performance, as well to evaluate both the trustworthiness of the probability estimates and their practical

utility. We use **Expected Calibration Error (ECE)** as the primary standard statistic for measuring the alignment between confidence and correctness, alongside **Maximum Calibration Error (MCE)** to capture worst-case miscalibration risks that might be hidden by averages. Beyond alignment, we use the **Brier Score (BS)** and **Brier Skill Score (BSS)** to measure refinement and predictive power. We specifically prioritize BSS as it allows us to contextualize the model's performance against a naive baseline (the average acceptance rate), thereby quantifying the actual "skill" the calibration adds over a simple guess. More details of the metrics are presented below.

- **Expected Calibration Error (ECE):** This metric approximates the difference between expected confidence and accuracy. The predictions are partitioned into M bins $(B_1, \ldots, B_m)$ based on their confidence scores. The ECE is the weighted average of the absolute difference between the mean predicted confidence and the mean true outcome in each bin. A lower ECE is better.
- **Maximum Calibration Error (MCE):** Similar to ECE, MCE considers the binned differences but reports the maximum deviation, i.e., the worst-case miscalibration.

$$\text{MCE} = \max_{m \in \{1,\ldots,M\}} \left| \left(\frac{1}{|B_m|} \sum_{i \in B_m} Y_i \right) - \left(\frac{1}{|B_m|} \sum_{i \in B_m} \hat{p}_i \right) \right|$$

- **Brier Score (BS):** The Brier score measures the mean squared error between the predicted probability $\hat{p}_i$ and the outcome Y_i. It is a measure of both calibration and resolution.

$$\text{BS} = \frac{1}{N} \sum_{i=1}^{N} (\hat{p}_i - Y_i)^2$$

  Lower values are better.
- **The Brier Skill Score (BSS):** quantifies the relative improvement in accuracy of one probabilistic forecast over another. Formally, it compares the BS of two forecasts, x and y, as $BSS = 1 - (BS_x / BS_y)$ where BS_x and BS_y denote the Brier Scores of forecasts x and y, respectively. A positive BSS indicates that forecast x produces lower error (and is therefore more skillful) than forecast y, while a negative BSS indicates the opposite. In general:
  - $BSS > 0$: forecast x is more skillful than forecast y;
  - $BSS = 0$: forecast x has no additional skill over forecast y;
  - $BSS < 0$: forecast x is less skillful than forecast y.

  In this work, we adopt a *naive base rate model* as the reference forecast y, which always predicts the historical average acceptance rate. Our reported BSS values represent the relative improvement of each calibrated model over this naive baseline.

When we refer to metrics such as the Brier score for calibration, we are effectively working with a mean squared error–like quantity. In our setup, the positive label (1) corresponds to a suggestion being (partially) accepted, represented as a value in [0, 1]. Because partial acceptances are extremely rare (0.059% of the data), we threshold these values: $x \geq 0.5$ is mapped to 1 and all others to 0. This binarization justifies our use of the Brier score terminology. We use a 26% average acceptance rate for the general model, and language-specific average acceptance rates for individual languages.

Table 2: Statistical overview and language distribution of the user interaction dataset.

Language	Metric	Value
Java	Total records	11,938,231
	Share (%)	48.20%
	Acceptance rate (%)	28.09%
Python*	Total records	9,460,897
	Share (%)	38.20%
	Acceptance rate (%)	24.29%
Kotlin	Total records	3,368,833
	Share (%)	13.60%
	Acceptance rate (%)	23.44%
Total	Total records	**24,767,961**
	Share (%)	**100.00%**
	Acceptance rate (%)	26.00%

* Formed by 8,894,629 `python` records and 566,268 `jupyterpython` records.

4.2 RQ1 Results

To evaluate the efficacy and methodology of our approach, we use a large-scale, real-world dataset of anonymized user interactions. This data is a subset of user data that was collected over a six-month period, from January 1, 2025, to July 1, 2025. The subset contains **24,767,961** records, obtained from **759,983 unique devices** across **3,963,331 distinct sessions**. This data was collected with consent from all the involved developers and does not contain any personal identifiable information. While we cannot freely share the entire dataset used for this study due to regulations, we have included a snippet of approximately 500 rows of user-interaction data spanning 10 internal JetBrains employees in the replication package [13]. [3] This subset of the data contains interaction data across the programming languages of interest. More specifically, Java, Python, and Kotlin. For the purpose of our analysis, we merge interactions from Jupyter notebooks (*jupyterpython*) with standard Python (*python*) files into a single, unified Python category. This decision is motivated by the semantic similarity of the code and the desire to create a comprehensive view of Python development activity. The distribution of records across these languages is presented in Table 2. Java constitutes the largest portion of the dataset (48.20%), followed by the Python category (38.20%) and Kotlin (13.60%).

This portion of the data is derived from a selection of suggestion interactions that we can confidently confirm have been presented to the user at least once and have either been dismissed or (partially)[4] accepted by them. In this analysis, we specifically focus on the interaction data with the generation model [5], for which we have complete access to confidence metrics like token-level probability.

We present a detailed overview of the results from RQ1 in Table 3. The results are shown for both our internal training dataset and a real-world logs dataset (which in ML terminology would equate to the train and test set). With this data we are able to assess performance on both in-distribution and out-of-distribution data.

Observation 1. The primary observation is that the uncalibrated model consistently exhibits poor calibration as evident by high

[3] The full dataset can be obtained upon reasonable request by contacting either of the corresponding authors and signing a non-disclosure agreement.
[4] Partial acceptance involves accepting some tokens from a completion. Unlike tab completion, users can select a token subset as a partial prefix from the suggestion.
[5] Mellum-4b Model https://huggingface.co/JetBrains/Mellum-4b-base

Table 3: Calibration results comparing the uncalibrated model confidence, a general calibrator, and language-specific calibrators. The evaluation is performed on both the Training Dataset and a real-world Logs Dataset. Arrows indicate the preferred direction for each metric (↓: lower is better, ↑: higher is better).

		Training Dataset				Logs Dataset			
Language	**Model**	**ECE ↓**	**BS ↓**	**BSS ↑**	**MCE ↓**	**ECE ↓**	**BS ↓**	**BSS ↑**	**MCE ↓**
Java	Uncalibrated	0.345	0.345	-1.728	0.477	0.300	0.308	-0.524	0.488
	General ($f_{general}$)	0.034	0.225	-0.777	0.300	0.082	0.203	-0.006	0.444
	Language-Specific (f_{Java})	0.031	0.225	-0.783	0.270	0.100	0.206	-0.022	0.443
	Baseline (Avg)	-	0.126	-	-	-	0.202	-	-
Python	Uncalibrated	0.408	0.392	-1.644	0.487	0.298	0.288	-0.567	0.491
	General ($f_{general}$)	0.070	0.230	-0.551	0.489	0.096	0.187	-0.018	0.469
	Language-Specific (f_{Python})	0.053	0.228	-0.536	0.500	0.083	0.185	-0.005	0.478
	Baseline (Avg)	-	0.148	-	-	-	0.184	-	-
Kotlin	Uncalibrated	0.380	0.377	-2.030	0.457	0.317	0.304	-0.694	0.535
	General ($f_{general}$)	0.041	0.230	-0.848	0.332	0.112	0.190	-0.059	0.572
	Language-Specific (f_{Kotlin})	0.044	0.230	-0.852	0.346	0.116	0.191	-0.065	0.570
	Baseline (Avg)	-	0.124	-	-	-	0.179	-	-
Overall	Uncalibrated	0.379	0.372	-1.768	0.463	0.301	0.300	-0.558	0.495
	General ($f_{general}$)	0.041	0.228	-0.698	0.368	0.092	0.195	-0.015	0.465
	Baseline (Avg)	-	0.134	-	-	-	0.192	-	-

expected calibration errors (ECE), Brier scores (BS), maximum calibration errors (MCE) and strongly negative Brier Skill Scores (BSS). After applying the general calibrator ($f_{general}$), we see a uniform improvement across all metrics for Java, Python, Kotlin, and the overall aggregation. For example, in the logs data set, the overall ECE drops from 0.301 to 0.092, and the Brier Skill Score increases from −0.558 to −0.015; this indicates that the calibrated predictions perform nearly on par with or slightly worse than a naive mean probability baseline. This confirms that $f_{general}$ effectively reduces overconfidence and improves probabilistic accuracy, however, is still not improving on the naive baseline.

Observation 2. The second observation concerns the limited benefit of specialization. Comparing the general calibrator ($f_{general}$) to the language-specific variants (e.g., f_{Java}, f_{Python}, f_{Kotlin}) reveals no consistent advantage on the Logs Dataset. While in isolated cases, such as for Python, the language-specific model slightly improves metrics all-round, these marginal gains are accompanied by minor drops in the same metrics for another language-specific model (see BSS f_{Java}). This suggests that, although specialization can slightly adapt to language-level idiosyncrasies, the broader statistical exposure of the general model yields more stable calibration performance under distribution shift, which is a likely advantage when facing the heterogeneous and noisy real-world user data.

Summary of RQ1. Calibration increases confidence reliability across languages and datasets. The general calibrator ($f_{general}$) beats the uncalibrated model but only matches or slightly exceeds a simple baseline. Language-specific variants offer small, inconsistent gains, underscoring the robustness and practicality of a single general calibrator.

5 RQ 2.

While RQ1 explores static calibration models, RQ2 investigates the hypothesis that the optimal calibration of a code model is not universal but is instead a function of the individual developer and the specific project context. Developers have unique coding styles, acceptance thresholds (e.g., during *vibe coding* compared to when programming for work), and expertise, which may change depending on the project they are working on. This motivates a shift from static, one-size-fits-all calibrators to dynamic models that adapt over time to these specific contexts.

5.1 Method

To address this, we introduce an online learning framework where the models are continuously updated based on a stream of developer interactions. This allows us to create and compare several adaptive models, each of which captures a different level of context.

5.1.1 Online Learning Framework. In an online learning setting, the data arrives as a sequential stream [6] of observations. At each time step t, we observe a new interaction consisting of the model's raw confidence for a given suggestion, $conf_t(S, C)$, and the developer's anonymized feedback, used as the outcome Y_t (e.g, acceptance).

Our calibration model, f, evolves over time, denoted as f_t. At any given time t, the current model f_t produces a calibrated prediction $\hat{p}_t = f_t(conf_t(S, C))$. After the true outcome Y_t is observed, this new pair $(conf_t, Y_t)$ is used to update the model to f_{t+1} for subsequent predictions. We apply this framework to two distinct scenarios to disentangle the effects of user and project context:

- **Per-Person Calibration ($f_{user,t}$):** A single adaptive model is maintained for each developer. This model is updated with all interaction data from that user, *regardless of the project.* This model aims to capture a developer's general, project-agnostic interaction style.
- **Per-Person-Per-Project Calibration ($f_{user,project,t}$):** An adaptive model is maintained for each unique *user-project pair.* The model updates only with data from that user's interactions on that project. This tests the hypothesis that a developer's behavior is highly context-dependent (For instance, a user's criteria for Project A are different from those for Project B.).

[6] We define a stream to be a temporally ordered set of AI-completion interaction logs for a given grouping.

5.1.2 Progressive Validation and Model Adaptation. To evaluate the performance of these adaptive models in a temporally-aware manner, we use a progressive validation (or rolling window) methodology. This approach allows us to measure how the calibration performance evolves as the model is exposed to more data. The process is as follows:

(1) **Initialization:** Each adaptive model ($f_{user,t}$ or $f_{user,project,t}$) is initialized, for example, with the general-purpose calibrator, $f_{general}$, from RQ1. Let the initial model be f_0.
(2) **Evaluation Window:** Let W be a sliding window of fixed size k. At time T in a data stream, the window W_T consists of the k most recent observations in that stream: $W_T = \{(conf_t, Y_t)\}_{t=T-k+1}^{T}$
(3) **Evaluate:** We use the initial model to produce calibrated confidences for all k observations in W_T, and then compute standard calibration metrics (ECE, BSS, etc.) over this window. We update the Brier Skill Score (BSS), reference model via Bayesian updates. As in RQ1, we start from the average acceptance probability as a prior and update this reference (the average acceptance estimate) with the observations in each window, yielding more reliable BSS estimates across windows.
(4) **Fine-tune (Adapt):** The observations within the window W_T are used to update the calibration model from f_{T-k} to a new version, f_T. For Platt Scaling, this involves incrementally retraining the logistic regression parameters using a recent history of observations from the relevant stream.
(5) **Slide and Repeat:** The window is moved forward, and the process is repeated. This essentially generates a time series of performance metrics for each adaptive model.

5.1.3 Temporal Evaluation. Our progressive validation primarily yields time series of metric scores (e.g., ECE_T, BSS_T), and we analyze these results with an emphasis on their temporal behavior. We measure the overall correlation between the calibrated probabilities ($\hat{p}$) from each type of adaptive model and the outcomes (Y), comparing them to each other and to the static models from RQ1. We also examine metric score time series to see whether, and how quickly, the adaptive models reach a stable, better-calibrated state. Finally, by comparing the average performance of the $f_{user,t}$ and $f_{user,project,t}$ models, we identify which contextual factors (the person, the project, or their interaction) matter most for accurate calibration. We use a sliding window W of 200 interactions and require a warm-up of 50 data points before applying model updates. We found that this configuration balances reliance on recent data with the noise present in single interactions.

5.2 RQ2 Results

Our analysis of the adaptive calibration framework shows that dynamic, user-aware learning offers moderate but consistent gains over the static general-purpose calibrator from RQ1. We evaluated two adaptive strategies: a **Per-Person** model ($f_{user,t}$) and a **Per-Person-Per-Project** model ($f_{user,project,t}$). Table 4 summarizes the aggregated performance across all users and projects.

Compared to the general calibrator ($f_{general}$), which achieves an average ECE of 0.092, both adaptive models demonstrate lower calibration errors, with ECE values of **0.066** and **0.068**, respectively. Similarly, both models yield reduced Brier Scores and improved Brier Skill Scores (BSS), moving from the negative baseline of −0.015 to positive averages of 0.020 and 0.013. This improvement in BSS, which measures relative performance over a naïve mean-probability predictor, confirms that the adaptive models are not only better calibrated but also more informative in their probabilistic predictions. The comparable performance of the two adaptive strategies suggests that most of the benefit arises from user-level adaptation, with the added project-level granularity contributing marginally at best. This can be attributed to data sparsity: on average, individual users provide longer interaction sequences than user–project pairs, leading to more stable model updates.

To better understand the relationship between activity level and calibration effectiveness, we segmented users and projects into three groups of activity:**Low**, **Average**, and **High**, based on the volume of interactions (Table 5). The results reveal a clear dependency between data availability and adaptation quality.

For the **Per-Person** model, ECE and BS steadily improve with increased activity, and the BSS rises from 0.009 for low-activity users to 0.024 for highly active users, indicating that personalization becomes increasingly effective with richer interaction histories. In contrast, the **Per-Project** model exhibits greater volatility: for low-activity pairs, performance worsens relative to the general model (BSS of −0.040), and reflects unreliable adaptation under data scarcity. However, with sufficient activity, the model recovers and achieves a BSS of 0.012, similar to the per-person model.

Observation 3. We compute the skill score on using model confidence for predicting code retention (acceptance), over a very large amount of actual telemetry data. Given the common practice of using *model confidence* to select good suggestions, we believe this is an important quantity to measure; to our knowledge, this hasn't been done before. Despite Platt scaling, the skill remains rather low, especially when compared to prior reported skill scores, which were for code *test-passing* correctness [10, 25]. This finding suggests that using model confidence to select code completions may have limitations, and other approaches may provide confidence measures that have more reliable associations with retention.

Summary of RQ2. For RQ2, we find that adaptive calibration improves both confidence alignment (ECE, BS) and relative predictive performance (BSS). Yet gains depend strongly on access to user- or project-specific data: personalized calibrators improve as more interactions are observed, while in sparse-data settings the static model remains the most dependable choice.

6 RQ 3.

While RQs 1 and 2 discussed the calibration of model confidence in code suggestions, RQ3 intends to investigate the best way to communicate confidence reliability values to the IDE user.

6.1 Method

To understand how the reliability signal, represented by LLM's confidence, should be integrated into developer workflows, we conducted a multi-phase design user study. It combined scenario-based design [23] with semi-structured interviews [2] and larger-scale surveys to systematically explore and validate implementation approaches for confidence signals in AI-assisted code generation. The

Table 4: Overall performance of adaptive calibration models compared to the static general model. Reported are the mean ± standard deviation across users/projects. Lower is better for ECE, Brier, and MCE; higher is better for BSS.

Model	Adapted ECE ↓	Adapted Brier ↓	Adapted BSS ↑	Adapted MCE ↓
General (f_{general}, from RQ1)	0.092	0.195	−0.015	0.465
Per-Person ($f_{\text{user},t}$)	0.066 ± 0.038	0.166 ± 0.054	0.020 ± 0.107	0.470 ± 0.253
Per-Person-Project ($f_{\text{user,project},t}$)	0.068 ± 0.038	0.165 ± 0.053	0.013 ± 0.128	0.484 ± 0.256

Table 5: Performance of adaptive models segmented by user activity level. Reported are the mean ± standard deviation across users/projects. Lower is better for ECE, Brier, and MCE; higher is better for BSS.

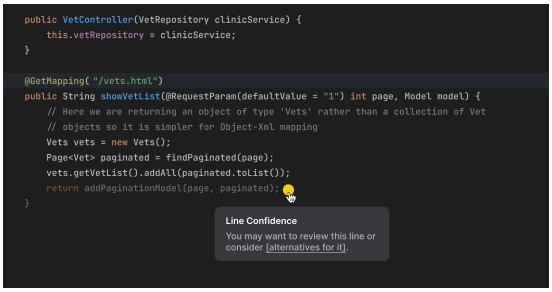
Model	Activity Group	Adapted ECE ↓	Adapted Brier ↓	Adapted BSS ↑	Adapted MCE ↓
Per-Person	Low	0.070 ± 0.039	0.170 ± 0.053	0.009 ± 0.113	0.486 ± 0.255
	Average	0.066 ± 0.036	0.169 ± 0.053	0.016 ± 0.128	0.473 ± 0.253
	High	0.065 ± 0.039	0.164 ± 0.055	0.024 ± 0.096	0.465 ± 0.253
Per-Project	Low	0.095 ± 0.042	0.179 ± 0.048	−0.040 ± 0.201	0.577 ± 0.263
	Average	0.072 ± 0.036	0.169 ± 0.052	0.003 ± 0.138	0.506 ± 0.257
	High	0.065 ± 0.037	0.163 ± 0.054	0.012 ± 0.120	0.473 ± 0.254

study received ethical approval from TU Delft's Human Research Ethics Committee (App Nr: 5682). We share the design session protocol, interview script, and survey instrument in the replication package [13]. Note that the collected user data cannot be made public for privacy reasons, but can be shared upon reasonable request.

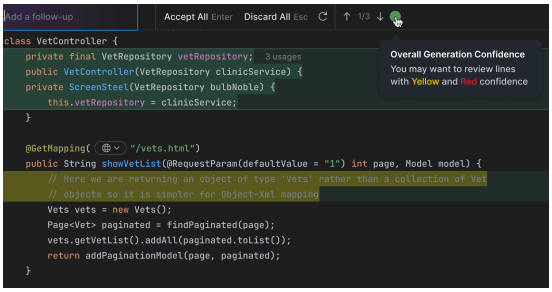
6.1.1 Design Sessions with Professional Designers. We conducted 90-minute design sessions with three expert UI designers (14, 7, and 15 years of experience) specializing in AI interfaces (1, 1.5, and 3 years of experience correspondingly). Participants were recruited from JetBrains, where they were employed as professional designers. The sessions employed collaborative sketching [24] to explore interface concepts for presenting confidence information. The protocol of sessions was reviewed by professional copyeditors employed at JetBrains and refined through pilot testing. The protocol included system introduction, structured design tasks, and reflection on the suggested design of confidence signal integration into the UI of the in-IDE AI assistant. Sessions were conducted remotely, and audio and video were recorded for analysis. The first two authors collaboratively reviewed session transcripts and design artifacts compiled in Figma[7] to identify actionable design principles and interface concepts.

Through repeated discussions, the researchers distilled the results into key interface design dimensions for further investigation with professional developers.

6.1.2 Semi-structured Interviews with Developers. Five professional developers (3-6 years of experience, using AI coding tools daily) were recruited via internal advertisement within JetBrains and participated in 30-minute semi-structured interviews. The interviews explored preferences regarding design dimensions identified in the previous phase. Participants were introduced to the calibrated confidence concept as a reliability signal and asked to identify contexts where they would be useful, appropriate granularity levels for assessment, and potential challenges associated with such signals. Interview recordings were transcribed and collaboratively analyzed by the first two authors to identify preference patterns and integration requirements. Based on this analysis, two candidate interface designs were developed (see Figure 2) for quantitative validation through a survey. We emphasize that the design sessions and interviews were purely formative and exploratory, following an exploratory-sequential design. The goal of this phase is to generate reliable hypotheses for evaluation on the larger sample size in the subsequent *validation* phase.



(a) In-editor autocompletion with reliability signal



(b) In-editor code generation with reliability signal

Figure 2: Candidate interface designs for reliability signal presentation. (a) A tooltip appears when hovering over the yellow signal, prompting review. (b) Yellow highlighting indicates medium-reliability lines in the generation panel.

[7] Figma: Collaborative Design Tool https://www.figma.com

6.1.3 Survey Validation. An online survey was conducted to validate the interface designs with 148 professional developers. Participants were recruited from a JetBrains opt-in research panel and could enter a prize draw for one of five 50 USD Amazon eGift Cards or an equivalent-value JetBrains product as compensation. Participants evaluated both interface approaches (autocompletion-integrated and code generation-integrated) using 5-point Likert scales measuring perceived usefulness, trust impact, workflow disruption potential, and indicated overall preference. The survey included screening questions to confirm professional coding experience and current use of AI coding tools at least once per month. Quantitative responses were analyzed using descriptive statistics. Open-ended responses provided additional context for interpreting quantitative findings.

6.2 RQ3 Results

The designer phase formulated design considerations for in-IDE AI-generated code reliability signals. Designers suggested that signals could be shown at various workflow points with AI: in-editor autocompletion, in-editor code generation from in-line prompts, and AI chat. Designers also proposed that signals could be attributed to different granularity levels: token, line, several lines, or whole code snippet. Multiple signal variations were suggested: traffic-light-like, numerical, and highlight approaches. Additional considerations included what reliability levels are useful to signal and what explanations might be beneficial.

Overall principles for such a feature, advised by participating designers, are:

- Leverage Existing Patterns: Build on established IDE conventions and visual signals;
- Minimalism First: Avoid cognitive overload and visual clutter, show minimal information by default with detailed information on demand;
- Actionability: Reliability signals should guide users toward next steps;
- Contextual Adaptation: Different presentations for different user types, code criticality, and granularity levels.

With distilled questions and design principles established, professional developers showed during interviews a preference for in-editor reliability signals with multi-line granularity. Developers indicated that scores alone provide insufficient actionable information for code acceptance decisions. Participants consistently favored explanatory feedback over numeric confidence indicators and requested reasoning behind code suggestions, echoing the actionability principle from designers. All participants emphasized the critical importance of user-configurable thresholds.

Based on these preferences, we constructed two UI prototypes reflecting the most promising approaches from our design exploration: an autocompletion-integrated signal and an in-editor generation signal (see Figure 2). These prototypes were presented to survey participants for quantitative validation.

The survey results revealed that both interface approaches received identical median ratings across key metrics: usefulness (3/5), trust impact (3/5), and workflow disruption potential (2/5). This convergence suggests that neither approach presents significant usability barriers, yet neither emerged as clearly superior in isolation.

However, when forced to choose between approaches, developers demonstrated a clear preference hierarchy. The in-editor generation approach garnered support from 76 participants (51.4%), while the autocompletion-integrated approach attracted 55 participants (37.2%). Notably, 17 participants (11.5%) rejected both options, indicating that reliability signaling may not universally appeal to all developers. This preference for in-editor generation aligns with our interview findings, where developers emphasized the need for actionable information during active code creation. This approach lets developers evaluate reliability without sacrificing control of their code, aligning with the principles defined in our design phase.

Summary of RQ3. Our findings support implementing reliability signals through non-numerical, color-coded indicators integrated into the in-editor generation workflow, as illustrated in Figure 2 (b). This approach balances visibility with minimal cognitive overhead while providing contextual information at the point of decision-making.

7 Discussion

Our mixed-methods approach, combining large-scale telemetry analysis with a user-centered design study, provides a holistic view of confidence calibration for AI coding assistants. Our work provides the first end-to-end treatment of confidence calibration for production-level AI coding assistants, bridging technical calibration methods with human-centered interface design. While prior work has either focused on uncertainty communication strategies [30] or calibration techniques in isolation [25], we demonstrate how these pieces fit together in a production system. We further discuss the implications of our findings, acknowledge the limitations of our study, and outline threats to its validity.

7.1 Implications of Findings

General Calibrator Has Low Predictive Ability (RQ1).

Results in Table 3 demonstrate that the general calibrator ($f_{general}$) consistently improves calibration metrics (such as ECE) over the uncalibrated model across all languages. Notably, it matches or even outperforms language-specific models (f_{lang}) in stability when tested on the logs dataset. For platform providers, this is a significant finding as it simplifies deployment and maintenance as one robust calibrator can be used for all users. This study supports Spiess et al. [25]'s findings on the necessity of post hoc calibration by using extensive real developer interaction data instead of synthetic benchmarks. It also indicates that the statistical traits of miscalibration likely apply across languages.

Our findings also highlight a key limitation: calibration aligns model confidence with outcomes, reducing ECE, but fails to make confidence a reliable predictor of user behavior. As indicated in Table 3, the general calibrator's Brier Skill Score is -0.015, showing that it offers no predictive benefit over a naive average acceptance rate baseline. This suggests a disconnect: model confidence (average token probability) may indicate syntactic or semantic correctness, but does not align with what developers *value* or *accept*. Relying solely on model confidence, even when calibrated, is inadequate for selecting useful code completions.

Calibrator Personalization Requires Substantial Data (RQ2). While the static general model lacks predictive skill, our results for RQ2 indicate that **adaptive, personalized calibration can succeed.** According to Table 4, the per-person model ($f_{\text{user},t}$) achieves an average BSS of 0.020, improving predictive skill over both the static model (BSS -0.015) and the naive baseline. Adapting to individual user acceptance patterns improves the confidence score's significance. However, our findings highlight the challenges in implementing personalization. While the *cold start* problem is evident, it is complex. **User-level adaptation ($f_{\text{user},t}$) is effective.** Even the "Low" activity group in Table 5 sees improvement with a BSS of 0.009 over the general model. However, **project-level adaptation ($f_{\text{user,project},t}$) is risky.** For Low activity user-project pairs, BSS drops to -0.040, indicating that too specific a model without adequate data worsens user experience. Findings suggest a hybrid approach: initially, all users use the system without any type of calibration. Once enough interaction data is gathered, the system can transition to a **per-person model** ($f_{\text{user},t}$) for improved predictions. The detailed *per-person-per-project* model is more viable for users with high interaction within a specific project, where its performance (BSS 0.012) is better.

Reliability Signal as a Supplementary Feature (RQ3). Perhaps our most actionable finding is that the value of a perfectly calibrated model is lost if its reliability is not communicated effectively. Developers are not statisticians, they do not want to interpret raw probability scores. The strong preference for non-numerical, color-coded, and workflow-integrated comprehensive signals provides a clear design guideline: reliability indicators should be informative and intuitive. Furthermore, the relatively modest median scores across all metrics according to our survey suggest that reliability signaling represents a *supplementary feature*. Developers view such signals as potentially useful but not essential, consistent with their expressed preference for user-configurable systems that can be adapted to individual workflows and code criticality levels. Our findings indicate that acceptance depends on human factors beyond mere correctness. Interviewees contrasted low-risk 'vibe coding' with high-risk production engineering, implying that urgency and perceived risk shape the bar for acceptance. Future research should isolate these factors to identify when developers will accept lower-confidence suggestions.

7.2 Threats to Validity

Internal Validity. Our primary measure of suggestion utility for the log analysis was the *preserved ratio*. While this is a common proxy for acceptance, it is imperfect. A developer might discard a correct suggestion because they thought of a better or alternative implementation. Conversely, they might accept an incorrect suggestion, only to fix it later. This potential mismatch between our proxy and the ground truth is a threat to internal validity.

In RQ3, selection bias may affect our findings, as design session and further validation participants were recruited from a single technology company, and survey participants from an opt-in research pool. However, we deliberately employed multi-phase design to reduce dependence on any single participant pool.

External Validity. Our study was conducted using a single, specific large language model (*Mellum-4b*) and on a user base from the JetBrains ecosystem. While large and diverse, these results may not generalize perfectly to other models (e.g., GPT-4, Claude 3) which may have different inherent calibration properties, or to developers who exclusively use other editors and have different workflow habits. Furthermore, our analysis focused on Java, Python, and Kotlin; the findings may not extend to languages with vastly different paradigms, such as C++ or Haskell. The design study participants (RQ3) represented experienced professionals. While findings may not generalize to novice developers or hobbyist programmers, our participant pool aligns with the primary user base for AI coding assistants in professional settings.

Construct Validity. The survey employed static mockups (Figure 2) to represent proposed interfaces. Static representations cannot fully capture the dynamic interplay between reliability signals and actual coding activities, as participants evaluated designs based on imagined workflows rather than lived experience. However, this approach is standard practice in formative design research [17] and appropriate for early-stage exploration before committing resources to full implementation. Our goal was not to validate final designs but to identify promising directions and eliminate clearly problematic approaches. Future work should include field deployment with behavioral logging to validate whether stated preferences translate to actual usage patterns and whether shown reliability signals improve code acceptance decisions in practice.

8 Conclusion

The integration of LLMs into IDEs has introduced a new paradigm for software development, but their fallibility raises challenges for reliability and collaboration. A model's ability to signal when it is uncertain is as important as its ability to generate correct code.

In this paper, we addressed this challenge from both a technical and a human-computer interaction perspective. Through a large-scale analysis of over 24 million real-world interactions, we observe that although post-hoc calibration through Platt-scaling improves the reliability of confidence over the base model, it falls short in predictive power. We found that general models have low skill, and that while personalization can offer further gains, its benefits are conditional on having a high volume of user data. Finally, our user study with 153 professional developers provided design directions for communicating these reliability signals, showing a preference for intuitive, non-numerical indicators embedded within the primary code generation workflow.

Taken together, our results indicate the need to explore other technical approaches (e.g., use of other internal model signals) to develop AI coding assistants that are more transparent, trustworthy, and user-friendly. Furthermore, future work should validate the provided UI designs in live A/B testing environments and explore how reliable confidence signals impact downstream developer behaviors, such as code review effort and bug introduction rates.

Acknowledgments

This work was conducted as part of the AI for Software Engineering (AI4SE) collaboration between JetBrains and Delft University of Technology. The authors gratefully acknowledge the financial support provided by JetBrains, which made this research possible.

References

[1] Saleema Amershi, Dan Weld, Mihaela Vorvoreanu, Adam Fourney, Besmira Nushi, Penny Collisson, Jina Suh, Shamsi Iqbal, Paul N Bennett, Kori Inkpen, et al. 2019. Guidelines for human-AI interaction. In *Proceedings of the 2019 chi conference on human factors in computing systems*. 1–13.

[2] Ann E Blandford. 2013. Semi-structured qualitative studies. Interaction Design Foundation.

[3] Git Copilot. 2025. Github Copilot. https://github.com/features/copilot.

[4] Cursor. 2023. Cursor: The AI-first Code Editor. https://cursor.sh.

[5] Yangruibo Ding, Zijian Wang, Wasi Uddin Ahmad, Hantian Ding, Ming Tan, Nihal Jain, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. 2023. CrossCodeEval: A Diverse and Multilingual Benchmark for Cross-File Code Completion. arXiv:2310.11248 [cs.LG] https://arxiv.org/abs/2310.11248

[6] Thomas Dohmke, Marco Iansiti, and Greg Richards. 2023. Sea change in software development: Economic and productivity analysis of the ai-powered developer lifecycle. *arXiv preprint arXiv:2306.15033* (2023).

[7] Shihan Dou, Haoxiang Jia, Shenxi Wu, Huiyuan Zheng, Weikang Zhou, Muling Wu, Mingxu Chai, Jessica Fan, Caishuang Huang, Yunbo Tao, et al. 2024. What's wrong with your code generated by large language models? an extensive study. *arXiv preprint arXiv:2407.06153* (2024).

[8] Emmanuel Anaya González, Raven Rothkopf, Sorin Lerner, and Nadia Polikarpova. 2025. HILDE: Intentional Code Generation via Human-in-the-Loop Decoding. *arXiv preprint arXiv:2505.22906* (2025).

[9] Chuan Guo, Geoff Pleiss, Yu Sun, and Kilian Q Weinberger. 2017. On calibration of modern neural networks. In *International conference on machine learning*. PMLR, 1321–1330.

[10] Yuheng Huang, Jiayang Song, Zhijie Wang, Shengming Zhao, Huaming Chen, Felix Juefei-Xu, and Lei Ma. 2023. Look before you leap: An exploratory study of uncertainty measurement for large language models. *arXiv preprint arXiv:2307.10236* (2023).

[11] JetBrains. 2025. JetBrains AI Assistant. https://www.jetbrains.com/junie/.

[12] Sunnie SY Kim, Jennifer Wortman Vaughan, Q Vera Liao, Tania Lombrozo, and Olga Russakovsky. 2025. Fostering appropriate reliance on large language models: The role of explanations, sources, and inconsistencies. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems*. 1–19.

[13] Roham Koohestani, Agnia Sergeyuk, David Gros, Claudio Spiess, Sergey Titov, Prem Devanbu, and Maliheh Izadi. 2026. *Replication Package for In-IDE Calibration*. doi:10.5281/zenodo.18231197

[14] Zoe Kotti, Konstantina Dritsa, Diomidis Spinellis, and Panos Louridas. 2025. The Fools are Certain; the Wise are Doubtful: Exploring LLM Confidence in Code Completion. *arXiv preprint arXiv:2508.16131* (2025).

[15] John D Lee and Katrina A See. 2004. Trust in automation: Designing for appropriate reliance. *Human factors* 46, 1 (2004), 50–80.

[16] Jingshu Li, Yitian Yang, Q Vera Liao, Junti Zhang, and Yi-Chieh Lee. 2025. As Confidence Aligns: Understanding the Effect of AI Confidence on Human Self-confidence in Human-AI Decision Making. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems*. 1–16.

[17] Youn-Kyung Lim, Erik Stolterman, and Josh Tenenberg. 2008. The anatomy of prototypes: Prototypes as filters, prototypes as manifestations of design ideas. *ACM Transactions on Computer-Human Interaction (TOCHI)* 15, 2 (2008), 1–27.

[18] Siddharth Mehrotra, Chadha Degachi, Oleksandra Vereschak, Catholijn M Jonker, and Myrthe L Tielman. 2024. A systematic review on fostering appropriate trust in Human-AI interaction: Trends, opportunities and challenges. *ACM Journal on Responsible Computing* 1, 4 (2024), 1–45.

[19] Ahmad Mohsin, Helge Janicke, Adrian Wood, Iqbal H Sarker, Leandros Maglaras, and Naeem Janjua. 2024. Can we trust large language models generated code? a framework for in-context learning, security patterns, and code evaluations across diverse llms. *arXiv preprint arXiv:2406.12513* (2024).

[20] Mahdi Pakdaman Naeini, Gregory Cooper, and Milos Hauskrecht. 2015. Obtaining well calibrated probabilities using bayesian binning. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 29.

[21] Neil Perry, Megha Srivastava, Deepak Kumar, and Dan Boneh. 2023. Do users write more insecure code with ai assistants?. In *Proceedings of the 2023 ACM SIGSAC conference on computer and communications security*. 2785–2799.

[22] John Platt et al. 1999. Probabilistic outputs for support vector machines and comparisons to regularized likelihood methods. *Advances in large margin classifiers* 10, 3 (1999), 61–74.

[23] Mary Beth Rosson and John M Carroll. 2007. Scenario-based design. In *The human-computer interaction handbook*. CRC Press, 1067–1086.

[24] Elizabeth B-N Sanders and Pieter Jan Stappers. 2008. Co-creation and the new landscapes of design. *Co-design* 4, 1 (2008), 5–18.

[25] Claudio Spiess, David Gros, Kunal Suresh Pai, Michael Pradel, Md Rafiqul Islam Rabin, Amin Alipour, Susmit Jha, Prem Devanbu, and Toufique Ahmed. 2024. Calibration and Correctness of Language Models for Code. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 495–507. doi:10.1109/ICSE55347.2025.00040

[26] Stack Overflow. 2025. 2025 Stack Overflow Developer Survey. https://survey.stackoverflow.co/2025 Accessed: Sep 10, 2025.

[27] Mark Steyvers and Megan AK Peters. 2025. Metacognition and uncertainty communication in humans and large language models. *arXiv preprint arXiv:2504.14045* (2025).

[28] Fengfei Sun et al. 2025. Large Language Models are overconfident and amplify human bias. *arXiv preprint arXiv:2505.02151* (2025).

[29] Jiao Sun, Q Vera Liao, Michael Muller, Mayank Agarwal, Stephanie Houde, Kartik Talamadupula, and Justin D Weisz. 2022. Investigating explainability of generative AI for code through scenario-based design. In *Proceedings of the 27th international conference on intelligent user interfaces*. 212–228.

[30] Helena Vasconcelos, Gagan Bansal, Adam Fourney, Q Vera Liao, and Jennifer Wortman Vaughan. 2025. Generation probabilities are not enough: Uncertainty highlighting in ai code completions. *ACM Transactions on Computer-Human Interaction* 32, 1 (2025), 1–30.

[31] Justin D Weisz, Shraddha Vijay Kumar, Michael Muller, Karen-Ellen Browne, Arielle Goldberg, Katrin Ellice Heintze, and Shagun Bajpai. 2025. Examining the use and impact of an ai code assistant on developer productivity and experience in the enterprise. In *Proceedings of the Extended Abstracts of the CHI Conference on Human Factors in Computing Systems*. 1–13.

[32] Windsurf. 2023. Windsurf AI Code Editor. https://windsurf.ai.

[33] Miao Xiong, Zhiyuan Hu, Xinyang Lu, Yifei Li, Jie Fu, Junxian He, and Bryan Hooi. 2023. Can LLMs express their uncertainty? An empirical evaluation of confidence elicitation in LLMs. *arXiv preprint arXiv:2306.13063* (2023).