Pre-Filtering Code Suggestions using Developer Behavioral Telemetry to Optimize LLM-Assisted Programming

Mohammad Nour Al Awad
ITMO University
Saint Petersburg, Russia
MohammadNourAlAwad@itmo.ru

Sergey Ivanov
ITMO University
Saint Petersburg, Russia
svivanov@itmo.ru

Olga Tikhonova
ITMO University
Saint Petersburg, Russia
tikhonova_ob@itmo.ru

***Abstract*—Large Language Models (LLMs) are increasingly integrated into code editors to provide AI-powered code suggestions. Yet many of these suggestions are ignored, resulting in wasted computation, increased latency, and unnecessary interruptions. We introduce a lightweight pre-filtering model that predicts the likelihood of suggestion acceptance before invoking the LLM, using only real-time developer telemetry such as typing speed, file navigation, and editing activity. Deployed in a production-grade Visual Studio Code plugin over four months of naturalistic use, our approach nearly doubled acceptance rates (18.4% → 34.2%) while suppressing 35% of low-value LLM calls. These findings demonstrate that behavioral signals alone can meaningfully improve both user experience and system efficiency in LLM-assisted programming, highlighting the value of timing-aware, privacy-preserving adaptation mechanisms. The filter operates solely on pre-invocation editor telemetry and never inspects code or prompts.**

***Index Terms*—Behavioral Modeling, Adaptive Systems, Human-Computer Interaction, Code Completion**

I. Introduction

Large Language Models (LLMs) have rapidly transformed the landscape of software development by enabling intelligent code completions, refactorings, and in-editor conversations. These capabilities are increasingly integrated into modern development environments, particularly through plugins for popular IDEs such as Visual Studio Code. However, despite their power, LLM-driven code suggestions often fail to align with developer intent in real-time, leading to low acceptance rates, disrupted workflows, and wasted computational resources [1].

Recent advances in LLM optimization have focused on improving model quality, response latency, or retrieval augmentation. Yet, much less attention has been given to the human side of the interaction—specifically, whether the developer is cognitively ready, task-wise situated, or contextually interested in receiving a suggestion at a given moment. Treating all editing contexts as equally appropriate for suggestion triggering is both computationally expensive and interactionally naive [2]. At a broader level, Sun et al. [3] have shown how LLM-based assistance can enhance not only coding efficiency but also upstream software design activities and overall developer experience. Large-scale enterprise deployments, such as those studied by Weisz *et al.* [4], confirm that usage context and developer workflows strongly shape the effectiveness and perceived value of AI-assisted coding tools.

In this paper, we propose a lightweight behavioral pre-filtering mechanism designed to operate prior to LLM invocation. Instead of analyzing prompt content or generated code, our system uses real-time behavioral telemetry to predict whether a developer is likely to accept a suggestion. This prediction serves as a gatekeeper: if the model forecasts low likelihood of acceptance, the suggestion request is suppressed, thus saving LLM resources and reducing interface interruptions.

Our approach is grounded in a human-centered view of developer-AI interaction. We model the developer not as a passive recipient of completions, but as an active agent engaged in cognitively demanding tasks—navigating files, writing code, pausing to think, reacting to errors, and using IDE features. These interaction patterns are observable and measurable through standard APIs in the development environment, allowing us to capture a continuous representation of the developer's state without manual labeling or intrusive interventions.

To evaluate our approach, we deployed the pre-filtering model inside a production-grade VS Code plugin used across our internal research center. The model was trained on months of sparsely collected telemetry data and suggestion logs. We compare usage before and after deployment of our trained model, revealing a significant improvement in suggestion acceptance rates—from 18.4% to 34.2%—and a reduction of approximately 35% in LLM suggestion calls. These results suggest that even in noisy and non-scripted environments, behavioral context alone can meaningfully improve the efficiency and usability of AI-powered programming tools.

Our contributions are as follows:

- We introduce a behavior-first framework for pre-filtering code suggestions, operating entirely without prompt or model content.
- We develop and deploy a production-ready system that captures, processes, and aligns behavioral telemetry in real-world VS Code usage.

2151-0849/25/$31.00 ©2025 IEEE
DOI 10.1109/ASEW67777.2025.00032

- We demonstrate substantial improvements in acceptance rates and resource efficiency, showing that behavioral signals can serve as reliable proxies for user readiness and intent.

This work opens new directions for modeling human-AI interaction in coding environments, emphasizing adaptivity, efficiency, and respect for developer context as first-class design goals.

The remainder of this paper is organized as follows: Section II reviews related work on adaptive suggestion systems and developer modeling. Section III outlines our telemetry design, behavioral framing, and modeling rationale. Section IV describes the deployment methodology and evaluation setup. Section V presents our findings and discusses the implications. Section VI concludes with reflections and future directions.

II. Related Work

A. AI-Powered Code Suggestions and Developer Acceptance

LLM-based coding assistants such as GitHub Copilot, Amazon CodeWhisperer, and Cursor have reshaped the way developers interact with code editors, enabling real-time suggestions for code completion, refactoring, and documentation. Despite their growing popularity, multiple empirical studies report relatively modest acceptance rates [5] in real-world usage. For example, field deployment at ZoomInfo found that only around one-third of Copilot suggestions were accepted by developers during daily work [6], [7]. Broader ecosystem analyses, such as Song *et al.*'s study of GitHub Copilot adoption in open-source projects [8], further emphasize that the impact of LLM-based assistance is shaped by collaborative and project-level factors. Recent work has also combined prompt-based models with multi-retrieval augmentation to increase completion accuracy and contextual relevance [9]. Other work has shown that the way suggestions are presented influences acceptance: Oertel et al. demonstrated that encouraging developers to explore more than the first candidate improves the quality of adopted completions without harming productivity [10].

While these studies focus on optimizing presentation or tuning suggestion ranking, our work takes a complementary direction by addressing the triggering decision itself—predicting whether any suggestion should be generated in the first place, based solely on real-time behavioral signals.

B. Telemetry-Driven Observability in IDEs

The increasing integration of AI tools into developer environments has sparked efforts to incorporate observability and telemetry as first-class entities. Koc et al. introduced the Model Context Protocol (MCP) [11], advocating for embedding metrics and runtime context into LLM-enhanced IDEs to better understand prompt behavior and system load [11]. Nam et al. analyzed fine-grained usage data from Google's code editor, revealing struggles with prompt-driven editing and proposing corrective workflows [12]. These works underscore the value of instrumentation in LLM systems, yet typically treat telemetry as a post-hoc debugging signal. In contrast, we use behavioral telemetry as a real-time signal to modulate LLM invocation itself.

C. Modeling Developer Behavior

A long-standing thread in software engineering has studied developer interactions to model cognitive states and predict future behavior. Earlier works used IDE command logs to predict developer intent [13], or topic models to infer upcoming tasks from code navigation patterns [14], [15]. Others have connected temporal features such as typing bursts or navigation frequency to self-reported productivity [15]. Similarly, Overwatch [16] leverages patterns in sequential code edits to predict future actions, underscoring the predictive power of temporal interaction traces. Our approach builds on this tradition by using observable developer activity—typing, pauses, corrections, and navigation—as a real-time representation of engagement and task context, to predict suggestion receptiveness. However, unlike prior work focused on retrospective prediction, our prefiltering method operates online and is embedded within a deployed LLM-powered assistant.

D. Interruptibility and Context-Aware Assistance

A parallel line of research in human-computer interaction explores the timing of assistance and its impact on cognitive load. Fogarty et al. pioneered the use of sensors to infer human interruptibility and modulate desktop notifications accordingly [17]. Subsequent work confirmed that interruptions during software development—especially those unrelated to the immediate task—can fragment attention and reduce perceived productivity [15]. Also misalignment between a developer's mental model of the AI assistant and its actual behavior can reduce trust and uptake; elicitation studies by Desolda *et al.* [18] highlight these cognitive factors as critical in designing timing-aware suggestion systems. Our system can be interpreted as a software-level analog to these models: it withholds suggestions during behavioral patterns suggestive of low interruptibility (e.g., high error load, frequent corrections, or rapid typing), thereby aligning system behavior with developer readiness.

E. Positioning and Novelty

In summary, prior work has emphasized three key themes: (i) measuring and improving LLM suggestion acceptance, (ii) instrumenting developer environments with observability and telemetry, and (iii) modeling developer behavior using IDE logs. This work bridges these strands by introducing a proactive behavioral gatekeeper that uses real-time telemetry to determine whether a suggestion should be shown. To our knowledge, it is the first deployed system to use solely developer behavioral state—rather than prompt content or model confidence—as the basis for suppressing or triggering LLM completions. Recent work by de Moor *et al.* has explored a complementary approach, using a transformer-based model to predict the optimal moments for invoking code completion based on IDE telemetry, highlighting the value of timing-aware suggestion systems [19]. Our findings show that such a filter can substantially reduce computational overhead while

improving the alignment between suggestions and developer intent.

III. Behavioral Metrics and Modeling

A. System Architecture

Our pre-filtering model is integrated into a production-grade, LLM-powered coding assistant delivered via a Visual Studio Code plugin. The system operates in real time and comprises several modular components. The plugin continuously captures behavioral telemetry from the developer's interaction with the IDE, applies heuristic constraints (e.g., max token length), and uses an adaptive mechanism to modulate suggestion triggering timing.

To this architecture, we added a lightweight behavioral pre-filtering model that predicts whether the developer is likely to accept a suggestion—before invoking the LLM. If the model forecasts low acceptance likelihood, the suggestion request is suppressed entirely, thereby reducing unnecessary GPU usage and latency. If the request passes the filter, it proceeds to a generation pipeline that includes a retrieval-augmented generation (RAG) component and Qwen LLMs served via vLLM for low-latency inference. Post-generation, suggestions are optionally trimmed or reformatted to improve UX.

While the full assistant includes additional modules—such as personalized code indexing, dynamic suggestion shaping, and real-time feedback logging—this paper focuses exclusively on the behavioral pre-filtering component and its impact on system efficiency and suggestion quality.

B. Data Collection Procedure

To understand the behavioral context surrounding code suggestion events, we instrumented a Visual Studio Code extension to log two primary types of telemetry: (i) event-level metrics associated with each suggestion request, and (ii) continuous behavioral summaries aggregated at a one-minute resolution. All data collection was conducted anonymously and in compliance with institutional consent procedures, with no raw code or prompt content retained.

a) Per-Suggestion Logging.: For every code suggestion request made through the plugin, we captured interaction-level metadata such as the length of the prompt, number of characters in the generated suggestion, time to accept or reject, and whether the suggestion was eventually accepted. A suggestion was marked as *accepted* if the developer inserted it into their code without requesting another suggestion. Conversely, if the developer requested a new suggestion without accepting the previous one, we labeled it as a *rejection* with passive rejection after 30 s of inactivity. This framing aligns with natural developer behavior in IDEs, where the act of bypassing a suggestion implies contextual irrelevance or dissatisfaction.

b) Continuous Behavioral Metrics.: In parallel, we recorded a rolling set of behavioral features aggregated every minute, providing a snapshot of the developer's activity independent of suggestion events. These include typing speed, frequency of pauses, file navigation events, command palette usage (e.g., copy, paste, quick fix), and code compilation outcomes such as warnings or errors. This behavioral stream forms the temporal context in which each suggestion request occurs and serves as the primary input to our pre-filtering model.

c) Estimating Task Complexity.: To contextualize developer effort, we compute a per-file scalar *Task Complexity* at the time of each autocomplete request. Using Tree-sitter, the surrounding code is parsed and summarized into a single score that combines cyclomatic paths, Halstead effort (operator/operand counts), and a maintainability index. The method supports all our supported languages, and falls back to heuristics based on decision keywords and LOC when a grammar is unavailable. Only aggregate metrics are retained—no code or identifiers—making the feature language-aware but content-agnostic.

C. Behavioral Categories and Feature Design

Rather than relying on code content, prompt formulation, or model internals, our system is driven by real-time telemetry reflecting how developers engage with the IDE environment. The classifier consumes a rich stream of behavioral signals captured prior to suggestion triggering—offering a proactive, privacy-preserving proxy for developer receptiveness.

We group the engineered features into five conceptual categories that reflect different dimensions of interaction: fluency (e.g., typing rhythm), editing scope, command usage, code state, and session context. Table I summarizes representative features across these categories.

Together, these metrics form a compact yet expressive representation of developer behavior. Since all features are computed before LLM invocation, the model operates in real time without inspecting prompt or code content, i.e. all predictors are purely behavioral and were chosen for interpretability and cross-language applicability.

TABLE I
Behavioral categories and representative features used by the acceptance classifier

Category	Purpose	Example Features
Interaction Fluency	Typing pace and continuity	Typing speed, Total characters typed, Pause count, Typing efficiency
Code Editing and Scope	Engagement with code structure and files	Lines of code added, File size, Edit density, Number of open files
IDE Command Usage	Use of assistive and tooling features	Undo frequency, Quick fix usage, Terminal toggling, Command palette actions
Code State	Stability and complexity of current code	Number of warnings/errors, Breakpoint count, Estimated code complexity
Session Context	Cumulative indicators across the session	Suggestions accepted/rejected, Acceptance ratio, Total typing duration

Engineered Behavioral Ratios: To capture dynamic aspects of developer interaction, we derive several engineered ratios from raw telemetry. These metrics normalize key behavioral signals relative to contextual baselines (e.g., time, file size), enabling consistent modeling across users and sessions.

$$\text{Typing Efficiency} = \frac{C_{\text{typed}}}{T_{\text{typing}} + \epsilon}$$

$$\text{Pause Frequency} = \frac{N_{\text{pauses}}}{T_{\text{typing}} + \epsilon}$$

$$\text{Acceptance Ratio} = \frac{N_{\text{accepted}}}{N_{\text{accepted}} + N_{\text{rejected}} + \epsilon}$$

$$\text{Edit Density} = \frac{L_{\text{added}}}{L_{\text{file}} + \epsilon}$$

Where:

- C_{typed}: Total characters typed during the aggregation window.
- T_{typing}: Total typing duration in seconds.
- N_{pauses}: Number of pauses exceeding a threshold (e.g., 2s).
- N_{accepted}, N_{rejected}: Count of accepted and rejected suggestions in the current session.
- L_{added}: Number of lines of code added during the window.
- L_{file}: Total number of lines in the current active file.
- ϵ: A small constant (e.g., 10^{-6}) to prevent division by zero.

D. Model Training

To operationalize the pre-filtering mechanism, we trained a lightweight binary classifier that predicts whether a suggestion is likely to be accepted by the developer based solely on real-time behavioral features. The dataset used for training comprised 2,318 unique suggestion events, collected from user sessions with informed consent. Out of these, 426 suggestions were accepted by developers, while the remaining 1,892 were either explicitly rejected or bypassed by requesting a new suggestion. This significant class imbalance (approximately 1:4.4) motivated the use of balanced learning techniques.

To ensure robust evaluation and threshold selection, the dataset was split into training, validation, and test sets using stratified sampling. Specifically, 64% of the data was used for training, 16% for validation, and 20% for final testing. Models were trained using randomized hyperparameter search over several lightweight classifiers, including XGBoost, LightGBM, CatBoost, and Balanced Random Forest. While all models achieved comparable performance, the CatBoost model was selected for deployment due to its higher trade-off between precision and recall at low thresholds, along with very small model size, which aligns with our goal of confidently suppressing unproductive suggestions.

Given a training dataset $\{(x_i, y_i)\}_{i=1}^{n}$, where each $x_i \in \mathbb{R}^d$ is a behavioral feature vector and $y_i \in \{0, 1\}$ denotes whether the suggestion was accepted, we train the classifier by minimizing a weighted binary cross-entropy loss:

$$\mathcal{L}(\theta) = -\sum_{i=1}^{n} w_{y_i} \cdot [y_i \log f_\theta(x_i) + (1 - y_i) \log(1 - f_\theta(x_i))]$$

Here, $f_\theta(x_i) \in [0, 1]$ is the model's predicted probability of acceptance, and w_{y_i} is a class-specific weight used to address the class imbalance. This formulation allows the model to prioritize recall on accepted suggestions while reducing the frequency of low-value completions.

Importantly, we did not seek to optimize for overall accuracy or F1-score. Rather, our primary objective was to reduce the number of low-value suggestion requests—true negatives—without significantly harming the number of accepted suggestions (true positives).

E. Justification for Behavioral-Only Filtering

By excluding prompt content, suggestion content, and developer intent labels from our model inputs, we adopt a deliberately minimalist and privacy-preserving design. This decision was informed by several practical and theoretical considerations:

- Real-time Operability: Behavioral signals are available before the suggestion is generated, making them ideal for proactive gating.
- Generalizability: Since behavioral patterns are modality-independent, the approach can generalize across languages and codebases without fine-tuning for domain-specific syntax.
- Privacy and Transparency: Avoiding the use of raw code or prompts mitigates concerns around data sensitivity and developer surveillance, aligning the system with ethical guidelines for responsible telemetry.
- Model Simplicity and Interpretability: Behavioral features are inherently interpretable (e.g., typing speed or undo frequency), allowing system designers to reason about model behavior and thresholds.

IV. Experimental Design and Deployment

A. Participants

Our adaptive suggestion system was deployed in a naturalistic academic research setting. Rather than recruiting a fixed participant group for a scripted study, we made the tool available to a pool of 25 researchers and developers via our internal communication channels. Lab members could freely install and use the plugin during their daily coding tasks. Usage was entirely voluntary and varied organically: some developers tried the system once or twice, while others used it regularly for extended periods.

Prior to the deployment, we surveyed 44 developers across multiple university research environments to identify those who regularly use Visual Studio Code as their primary IDE. From these, 25 were invited to participate and provided with instructions and technical support. Ultimately, 9 experienced developers actively used the assistant for a sufficient period, each contributing at least 50 recorded code suggestions. All

active participants had at least 3 years of programming experience (with 78% reporting over 5 years) and coded for more than 4 hours daily in most cases. Python was the dominant language among them, complemented by JavaScript, TypeScript, C++, and C. Notably, while all participants had prior experience with AI chat assistants (e.g., ChatGPT), most had not used AI-powered IDE plugins before, highlighting the novelty and practical relevance of the adaptive timing mechanism in real workflows.

All usage logs were anonymized and collected with informed consent in accordance with institutional policies.

B. Metric Suite and Threshold Selection

While standard classification metrics provide insight into model discrimination, interactive evaluation frameworks such as the RealHumanEval benchmark [20] demonstrate the importance of measuring assistance quality in realistic, human-in-the-loop coding scenarios. We evaluate our model in alignment with its primary role as a gatekeeper—deciding *whether* to trigger a suggestion—we adopt a metric suite that emphasizes both *ranking quality* and *class-specific utility* under imbalance, rather than global accuracy.

Specifically, we report:

- *ROC–AUC*: Measures general discriminative ability under balanced priors.
- *PR–AUC*: Captures precision-recall trade-offs under real class imbalance.
- *Balanced Accuracy*, *Matthews Correlation Coefficient (MCC)*, and *Cohen's* κ: Assess calibration against random or uninformative classifiers.
- *Brier Score*: Evaluates the quality of predicted probabilities.

Given the asymmetric costs of false positives and false negatives, we selected a deliberately low operating threshold ($\tau = 0.1$) on the validation set. This threshold maximizes recall for `accepted` suggestions (to avoid suppressing useful completions) while preserving high precision for the more frequent `rejected` class.

The decision rule for triggering the LLM can thus be formalized as:

$$\text{Trigger LLM} \iff \hat{P}_{\text{accept}}(x_{\text{behavioral}}) > \tau$$

where $\hat{P}_{\text{accept}}$ is the predicted acceptance probability based solely on behavioral features $x_{\text{behavioral}}$.

V. Results and Discussion

We evaluate the impact of our pre-filtering system using two complementary approaches: *(i)* offline analysis of model performance on a held-out test set, and *(ii)* live deployment analysis in real developer workflows.

A. Offline analysis of Classifier performance

1) Evaluation Metrics: Evaluation on a held-out test set (464 suggestions) shows:

These results confirm that despite moderate headline scores, the model provides valuable *ranking utility* under class imbalance.

TABLE II
Classifier Evaluation Metrics on Held-out Test Set

Metric	Value	Std. Dev.
ROC-AUC	0.726	$\pm$ 0.05
PR-AUC	0.350	$\pm$ 0.09
Balanced Accuracy	0.691	–
MCC	0.310	–
Cohen κ	0.191	–
Brier Score	0.139	–

2) Threshold Selection and Gatekeeping Behavior: Operating at a tuned threshold of $\tau = 0.10$:

- **True Negative Rate (Rejected Class)**: 41.6% of all rejections filtered before LLM invocation (precision = 0.981)
- **False Negative Rate (Accepted Class)**: Only 3.5% of accepted suggestions mistakenly filtered (recall = 0.965)

This asymmetry is desirable: filtering unproductive suggestions is cheap, while suppressing helpful ones risks user trust.

B. Understanding Behavioral Drivers

To better understand which behavioral factors contributed most to the model's decision-making, we conducted two complementary analyses of feature influence: permutation-based importance and SHAP (SHapley Additive exPlanations) value interpretation. Both techniques consistently identified several developer behaviors as key predictors of suggestion receptiveness.

As shown in Figure 1, recent acceptance ratio and suggestion history emerged as dominant predictors, with typing rhythm and help-feature usage playing secondary roles. We expand on these behavioral drivers below.

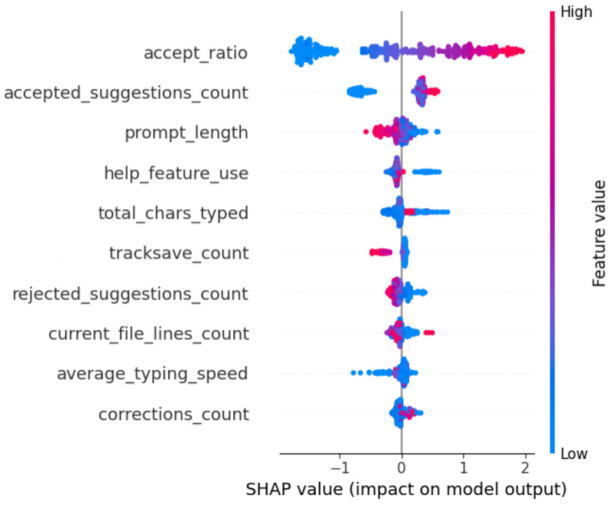

Fig. 1. SHAP value analysis for the CatBoost acceptance classifier.

a) Recent Acceptance Trends.: The most influential predictor was the developer's recent history of accepting or rejecting suggestions. A high frequency of accepted suggestions

typically indicated continued openness to assistance, whereas a streak of rejections often signaled disengagement. This highlights a momentum effect in developer behavior: prior acceptance is a strong cue for future receptiveness.

$$\text{Acceptance Ratio} = \frac{N_{\text{accepted}}}{N_{\text{accepted}} + N_{\text{rejected}} + \epsilon}$$

b) IDE Tool Usage and Problem-Solving Signals.: Frequent engagement with built-in help functions or automated quick fixes was associated with a lower likelihood of accepting suggestions. These behaviors may reflect task difficulty, uncertainty, or cognitive load—contexts in which developers are less receptive to interruptions or unsolicited input. This finding resonates with Tang *et al.* [21], who observed that developers engage in extensive validation and repair activities when presented with AI-generated code, especially during periods of high problem-solving activity.

c) Typing Rhythm and Flow.: Typing speed and the presence or absence of pauses provided insight into the developer's cognitive state. Sustained high-speed typing with minimal pauses suggested focus or flow, a state in which the developer is less likely to welcome external suggestions. In contrast, slower or fragmented typing often coincided with a higher likelihood of suggestion acceptance.

d) Cumulative Activity Context.: Longer-term interaction patterns—such as the overall volume of code written or the cumulative number of previously accepted completions—also proved informative. These signals helped the model contextualize short-term decisions within the broader scope of a coding session.

Taken together, these results validate the central premise of our approach: that real-time developer behavior offers rich, interpretable signals for deciding when AI assistance is likely to be welcome. The model's reliance on recent interactions, typing dynamics, and help-seeking patterns reflects a nuanced behavioral framing of suggestion timing—one that respects user attention and reduces unnecessary cognitive load.

C. Live Deployment Analysis

1) Improved Suggestion Acceptance and Call Reduction: After deployment of the pre-filtering mechanism, we observed two complementary gains: a higher acceptance rate on the suggestions that were issued, and a substantial drop in total LLM invocation.

First, of the 2,319 suggestion events recorded in the "before" period, all resulted in an LLM call and yielded an 18.4% acceptance rate. During the "after" period, the filter flagged 768 out of 2,190 incoming suggestion requests (35.1%), suppressing their upstream LLM calls. The remaining 1,422 requests proceeded to the LLM, and of those the developers accepted 34.2%. Thus:

- **Absolute acceptance increase:** +15.8 pp (18.4% $\rightarrow$ 34.2%)
- **Relative acceptance improvement:** +86%
- **LLM-call reduction:** 768 calls suppressed (35.1% of requests)

TABLE III
COMPARISON OF SUGGESTION DELIVERY AND ACCEPTANCE BEFORE VS. AFTER FILTERING

Metric	Before	After
Total suggestion events logged	2,319	2,190
LLM calls actually issued	2,319	1,422
Filtered out suggestions	0	768
Acceptance rate	18.4%	34.2%

By suppressing 768 low-value requests, we reduced GPU usage, network traffic, and backend queuing by nearly 35%, while simultaneously nearly doubling the yield of accepted completions per LLM call. This dual benefit—fewer interruptions and more relevant suggestions—highlights the value of a lightweight behavioral gate before triggering expensive inference.

2) Qualitative Impact on Latency and Attention: Suppressing unhelpful suggestions carries three downstream advantages:

- **Reduced GPU and network load:** At scale, a 35% reduction in calls translates directly to lower compute and hosting costs without any model modifications.
- **Fewer unwanted interruptions:** Developers reported fewer unsolicited suggestions during deep-focus tasks, aligning delivery with natural pauses and increasing perceived relevance.

Together, these results demonstrate that behavior-driven pre-filtering can both streamline system resource use and respect developer attention, all without touching the LLM itself.

3) Behavioral Filtering in Action.: Figure 2 illustrates the gatekeeping behavior of the model during a real developer session. Rather than applying a fixed or periodic policy, the system responds dynamically to interaction context. Suggestions are allowed during slower or exploratory phases, and filtered when telemetry indicates high focus or cognitive load. This adaptivity demonstrates the model's alignment with developer rhythms and its capacity to reduce interruptions without sacrificing useful completions.

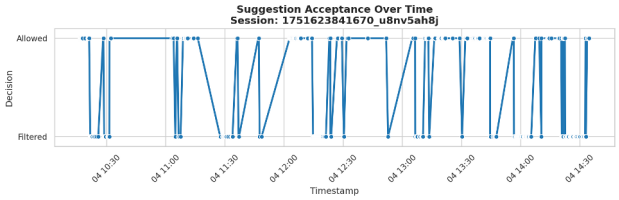

Fig. 2. Suggestion filtering decisions over the course of a real developer session. The model dynamically alternates between allowing and suppressing requests in response to real-time behavior. Extended filtering often coincides with focused activity, while suggestions tend to be allowed during pauses or fragmented interaction.

D. Statistical Significance of Acceptance-Rate Gains

To verify that the observed jump in suggestion–acceptance rates (from 18.4% to 34.2%; cf. Section V-C1) was not

due to sampling variability, we subjected the before–after counts to standard proportion tests. Confidence intervals were computed with the Wilson score method, and both a pooled two-proportion z-test and an exact Fisher test were applied at $\alpha = 0.05$.

TABLE IV
SUGGESTION ACCEPTANCE RATES BEFORE AND AFTER BEHAVIOURAL FILTERING, WITH 95% WILSON CONFIDENCE INTERVALS

Period	Accepted / Total	Rate (%)	95% CI (%)
Before (Baseline)	427 / 2 319	18.4	[16.9, 20.0]
After (Filtered)	486 / 1 422	34.2	[31.8, 36.7]

TABLE V
COMPARATIVE STATISTICS FOR THE TWO INDEPENDENT PROPORTIONS

Metric	Value
Absolute difference (Δ)	+15.8 percentage points
Risk ratio ($p_{\text{after}}/p_{\text{before}}$)	1.86
Odds ratio	2.30
Two-proportion z statistic	10.90
p-value (z-test)	$< 1 \times 10^{-26}$
p-value (Fisher exact)	$< 5 \times 10^{-26}$

Interpretation. The large, highly significant z statistic ($|z| = 10.9$, $p \ll 0.001$) and corroborating Fisher exact test confirm that the acceptance-rate improvement is statistically reliable. An odds ratio of 2.30 indicates that developers were more than twice as likely to accept a suggestion once behavioral filtering was enabled. The tight, non-overlapping Wilson intervals further underscore the practical importance of the effect. These results strengthen our claim that real-time telemetry can be leveraged not only to cut compute costs but also to deliver meaningfully better assistance to developers.

E. Limitations and Threats to Validity

While the results demonstrate a clear benefit from behavioral pre-filtering, several limitations and threats to validity must be acknowledged.

a) Participant Pool and Usage Patterns.: The plugin was deployed in an academic research environment with a relatively small pool of users (n=25), of whom only 9 contributed sustained usage data. Developer engagement was voluntary and often intermittent, leading to uneven distribution of suggestion events. As a result, the data may not fully represent the diversity of workflows seen in large-scale industrial settings. Future work will extend deployment to larger and more diverse industrial cohorts to further validate generalization.

b) Non-Randomized Deployment.: The evaluation design compares two time periods—before and after model deployment—without random assignment or counterbalancing. Although the conditions were held constant as much as possible, external factors such as developer experience, task type, or familiarity with the plugin may have influenced outcomes. This limits causal claims and motivates future controlled studies.

c) Behavioral Feature Scope.: Our model relies exclusively on observable interaction data from the VS Code API. While effective, these features do not capture deeper contextual factors such as semantic intent, code quality, or longer-term developer goals. Consequently, the behavioral signal may fail to distinguish between different types of engagement that lead to similar telemetry patterns. In future, richer contextual metrics could complement behavioral telemetry while preserving privacy.

d) Threshold Sensitivity.: The filtering mechanism depends on a tuned probability threshold to gate LLM requests. Although chosen to balance recall and resource savings, this value may require adjustment in other deployment contexts or user populations. Furthermore, fixed thresholds may not generalize well to developers with distinct working styles. Adaptive or personalized thresholds, potentially learned online, represent a promising path to make filtering more robust across diverse developer styles.

We also note that the deployed model uses a single model trained on pooled data from all participants. While this enables robustness across developers, it does not yet adapt to individual developer profiles. In practice, personalization could be maintained by lightweight per-user models and dynamically fine-tuning thresholds as acceptance/rejection feedback accumulates. We consider this as a promising future direction, and a complement to our current focus on population-level generalization.

Despite these limitations, the system achieved meaningful improvements under realistic conditions, supporting the value of behavioral telemetry as a practical signal for adaptive LLM invocation. Future work can address these concerns by scaling deployment, incorporating user feedback, and exploring more personalized filtering strategies.

VI. CONCLUSION

This work addressed an often-overlooked question in LLM-powered programming assistance: *when* to provide a suggestion. We introduced a lightweight, behavioral-only filtering mechanism that predicts the likelihood of code suggestion acceptance from real-time developer telemetry, without inspecting prompt or code content. Integrated into a production-grade Visual Studio Code plugin, the approach achieved two key outcomes in naturalistic use: (i) the suggestion acceptance rate increased from 18.4% to 34.2%, and (ii) over 35% of low-value LLM calls were suppressed. These gains were achieved without any modification to the LLM itself, demonstrating that optimizing *timing* can substantially improve both system efficiency and user experience.

Future Work

Building on this foundation, several research and engineering directions are promising:

- **Further Personalization and Adaptivity.** Develop per-user or per-session adaptive thresholds, potentially using online learning from ongoing acceptance/rejection feedback to refine responsiveness.

- **Richer Models of Developer State.** Extend beyond surface-level interaction metrics to capture higher-order temporal dynamics and transitions between states of focus, exploration, and problem-solving. This could enable more nuanced timing policies that adapt to the ebb and flow of cognitive engagement.
- **Behavioral-Driven Next Edit Prediction.** Build on this pre-filtering framework to anticipate the developer's likely next editing action, similar to Chen *et al.*'s adaptive next-edit suggestion approach [22]. Such capability could allow the system not only to time suggestions effectively, but also to shape their content in alignment with imminent editing goals—while remaining within a telemetry-only, privacy-preserving scope.
- **Longitudinal Analysis.** Explore multi-session behavioral trends and persistent user profiles to inform more temporally-aware filtering policies.

Closing Remarks

Our findings highlight that meaningful improvements to AI-assisted coding workflows can be achieved not only by enhancing model quality or prompt design, but also by strategically deciding *when* to assist. By treating developer attention as a first-class resource, behavioral pre-filtering offers a scalable, privacy-preserving, and model-agnostic pathway to reducing wasted computation, improving acceptance rates, and aligning assistance with real-time human context.

ACKNOWLEDGMENT

The research was supported by The Russian Science Foundation, agreement №24-11-00272, https://rscf.ru/project/24-11-00272/.

REFERENCES

[1] A. Ziegler, E. Kalliamvakou, X. A. Li, A. Rice, D. Rifkin, S. Simister, G. Sittampalam, and E. Aftandilian, "Measuring github copilot's impact on productivity," *Commun. ACM*, vol. 67, no. 3, p. 54–63, Feb. 2024. [Online]. Available: https://doi.org/10.1145/3633453

[2] H. Mozannar, G. Bansal, A. Fourney, and E. Horvitz, "When to show a suggestion? integrating human feedback in ai-assisted programming," in *Proceedings of the Thirty-Eighth AAAI Conference*, ser. AAAI'24/IAAI'24/EAAI'24. AAAI Press, 2024. [Online]. Available: https://doi.org/10.1609/aaai.v38i9.28878

[3] S. Sun, "Enhancing software design and developer experience via llms," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '24, New York, NY, USA, 2024, p. 2498–2501. [Online]. Available: https://doi.org/10.1145/3691620.3695606

[4] J. D. Weisz, S. Kumar, M. Muller, K.-E. Browne, A. Goldberg, E. Heintze, and S. Bajpai, "Examining the use and impact of an ai code assistant on developer productivity and experience in the enterprise," 2025. [Online]. Available: https://arxiv.org/abs/2412.06603

[5] B. Yetiştiren, I. Özsoy, M. Ayerdem, and E. Tüzün, "Evaluating the code quality of ai-assisted code generation tools: An empirical study on github copilot, amazon codewhisperer, and chatgpt," 2023. [Online]. Available: https://arxiv.org/abs/2304.10778

[6] G. Bakal, A. Dasdan, Y. Katz, M. Kaufman, and G. Levin, "Experience with github copilot for developer productivity at zoominfo," 2025. [Online]. Available: https://arxiv.org/abs/2501.13282

[7] S. Barke, M. B. James, and N. Polikarpova, "Grounded copilot: How programmers interact with code-generating models," vol. 7, no. OOPSLA1, Apr. 2023. [Online]. Available: https://doi.org/10.1145/3586030

[8] F. Song, A. Agarwal, and W. Wen, "The impact of generative ai on collaborative open-source software development: Evidence from github copilot," 2025. [Online]. Available: https://arxiv.org/abs/2410.02091

[9] H. Tan, Q. Luo, L. Jiang, Z. Zhan, J. Li, H. Zhang, and Y. Zhang, "Prompt-based code completion via multi-retrieval augmented generation," *ACM Trans. Softw. Eng. Methodol.*, Mar. 2025, just Accepted. [Online]. Available: https://doi.org/10.1145/3725812

[10] J. Oertel, J. Klünder, and R. Hebig, "Don't settle for the first! how many github copilot solutions should you check?" *Inf. Softw. Technol.*, vol. 183, no. C, Jun. 2025. [Online]. Available: https://doi.org/10.1016/j.infsof.2025.107737

[11] Anonymous, "Mind the metrics: Patterns for telemetry-aware in-IDE AI application development using model context protocol (MCP)," *Submitted to Transactions on Machine Learning Research*, 2025, under review. [Online]. Available: https://openreview.net/forum?id=Qhc8xDRZuH

[12] D. Nam, A. Omran, A. Murillo, S. Thakur, A. Araujo, M. Blistein, A. Frömmgen, V. Hellendoorn, and S. Chandra, "Prompting llms for code editing: Struggles and remedies," 2025. [Online]. Available: https://arxiv.org/abs/2504.20196

[13] T. Bulmer, L. Montgomery, and D. Damian, "Predicting developers' ide commands with machine learning," in *Proceedings of the 15th International Conference on Mining Software Repositories*, ser. MSR '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 82–85. [Online]. Available: https://doi.org/10.1145/3196398.3196459

[14] K. Damevski, H. Chen, D. C. Shepherd, N. A. Kraft, and L. Pollock, "Predicting future developer behavior in the ide using topic models," ser. ICSE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 932. [Online]. Available: https://doi.org/10.1145/3180155.3182541

[15] A. N. Meyer, L. E. Barton, G. C. Murphy, T. Zimmermann, and T. Fritz, "The work life of developers: Activities, switches and perceived productivity," *IEEE Trans. Softw. Eng.*, vol. 43, no. 12, p. 1178–1193, Dec. 2017. [Online]. Available: https://doi.org/10.1109/TSE.2017.2656886

[16] Y. Zhang, Y. Bajpai, P. Gupta, A. Ketkar, M. Allamanis, T. Barik, S. Gulwani, A. Radhakrishna, M. Raza, G. Soares, and A. Tiwari, "Overwatch: learning patterns in code edit sequences," vol. 6, no. OOPSLA2, Oct. 2022. [Online]. Available: https://doi.org/10.1145/3563302

[17] J. Fogarty, S. E. Hudson, C. G. Atkeson, D. Avrahami, J. Forlizzi, S. Kiesler, J. C. Lee, and J. Yang, "Predicting human interruptibility with sensors," *ACM Trans. Comput.-Hum. Interact.*, vol. 12, no. 1, p. 119–146, Mar. 2005. [Online]. Available: https://doi.org/10.1145/1057237.1057243

[18] G. Desolda, A. Esposito, F. Greco, C. Tucci, P. Buono, and A. Piccinno, "Understanding User Mental Models in AI-Driven Code Completion Tools: Insights from an Elicitation Study," *arXiv e-prints*, p. arXiv:2502.02194, Feb. 2025.

[19] A. de Moor, A. van Deursen, and M. Izadi, "A transformer-based approach for smart invocation of automatic code completion," in *Proceedings of the 1st ACM International Conference on AI-Powered Software*, ser. AIware 2024. New York, NY, USA: Association for Computing Machinery, 2024, p. 28–37. [Online]. Available: https://doi.org/10.1145/3664646.3664760

[20] Anonymous, "The realhumaneval: Evaluating large language models' abilities to support programmers," 2024. [Online]. Available: https://openreview.net/forum?id=M7SO74I9mo

[21] N. Tang, M. Chen, Z. Ning, A. Bansal, Y. Huang, C. McMillan, and T. J.-J. Li, "Developer behaviors in validating and repairing llm-generated code using ide and eye tracking," in *2024 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2024, pp. 40–46.

[22] X. Chen, S. Xiao, X. Zhu, J. Xie, M. Liang, D. Chen, W. Jiang, Y. Li, and P. Di, "An efficient and adaptive next edit suggestion framework with zero human instructions in ides," 2025. [Online]. Available: https://arxiv.org/abs/2508.02473