Smart Paste: Automatically Fixing Copy/Paste for Google Developers

Vincent Nguyen
Google
vincentdnguyen@google.com

Guilherme Herzog
Google
guiherzog@google.com

José Cambronero
Google
jcambronero@google.com

Marcus Revaj
Google
marcusrevaj@google.com

Aditya Kini
Google
akini@google.com

Alexander Frömmgen
Google
froemmgen@google.com

Maxim Tabachnyk
Google
tabachnyk@google.com

Abstract

Manually editing pasted code is a long-standing developer pain point. In internal software development at Google, we observe that code is pasted 4 times more often than it is manually typed. These paste actions frequently require follow-up edits, ranging from simple reformatting and renaming to more complex style adjustments and cross-language translations. Prior work has shown deep learning can be used to predict these edits. In this work, we show how to iteratively develop and scale Smart Paste, an IDE feature for post-paste edit suggestions, to Google's development environment. This experience can serve as a guide for AI practitioners on a holistic approach to feature development, covering user experience, system integration, and model capabilities. Since deployment, Smart Paste has had overwhelmingly positive feedback with a 45% acceptance rate. At Google's enterprise scale, these accepted suggestions account substantially for over 1% of all code written company-wide.

1 Introduction

The copy and paste command is ubiquitous within the developer workflow, allowing large amounts of code to be duplicated with a single keystroke. However, this action is often followed by a repetitive series of manual edits, requiring many more keystrokes to make the pasted code fit its new syntactic and semantic context. This manual fixing process is an involved and error prone task [22], encompassing a wide variety of necessary adjustments. These can range from standardized reformatting to conform to language-specific style guidelines, to minor variable renaming, or incrementing enumerated types. More complex pastes, such as integrating code from different sources or those that include new method calls, can demand familiarity with cross-language translations or library-specific APIs. These edits are tedious to perform, and misidentifying or incorrectly addressing them can easily lead to errors. Existing developer features in modern IDEs offer only partial solutions to the paste fixing problem:

- Code completion [8, 23, 25] can expedite fixes by offering predictive suggestions which can be accepted by tab completion to reduce keystrokes. However, this can only address incremental additions at the end of a paste region.
- Code formatters or linters [28] can address a subset of issues like style and dependencies, but tasks such as context-aware renaming or logic changes remain out of scope.
- Chatbots [1, 27], or other agentic approaches to software development [31], can make many types of changes but introduce significant latency and context-switching for the developer.

The goal of a Smart Paste feature is to address these shortcomings by developing a fast in-IDE suggestion feature. Past work [2, 15], has shown that deep learning models can be used to predict such edits. However, delivering the Smart Paste concept as a production-ready IDE feature requires overcoming several significant data, modeling, and systems challenges.

We built on this past work to develop a production-grade Smart Paste: a paste-edit suggestion feature inside Google's IDE, now used daily by tens of thousands of developers. As shown in Figure 1, the feature listens to everything pasted into the editor, then suggests fixes directly within the paste region that render as inline edits. Users can accept these suggestions with a tab or dismiss them with any other key or cursor movement.

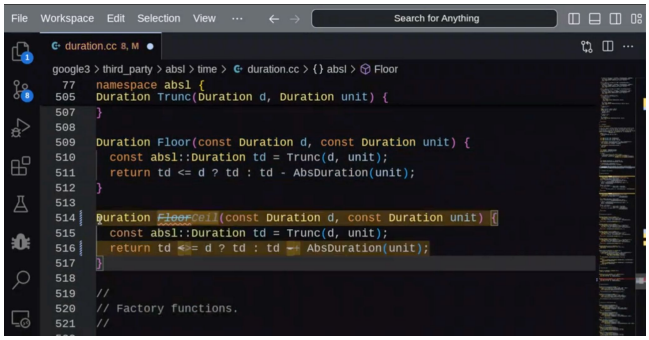

Figure 1: Smart Paste monitors a developer's IDE activity, so that when a paste takes place it can render (inline) suggested edits to adapt it to the current context.

This paper describes the practical challenges and solutions involved in building this real-world tool. Successfully productionizing

this feature required addressing a range of interconnected issues across the product stack, from devising a novel data collection pipeline for the inherently ambiguous task of "fixing" a paste, to engineering a system that could support a diverse, multilingual code base at enterprise scale with millisecond latency, and ultimately designing an intuitive, discoverable, and non-intrusive in-editor user experience. Each of these core challenges is defined and explored in greater detail in a subsequent section. Our holistic account of this development process can guide other AI practitioners in building similar coding assistance features.

The core contributions of our work are:

- A detailed account of the development of a production-grade Smart Paste IDE feature, including our novel method for creating fine-tuning datasets.
- A thorough evaluation, both offline and online, demonstrating the value of Smart Paste for Google developers at enterprise scale.

The remainder of this paper is structured as follows. We first define the problem in §2. Next, we detail our core technical contributions, covering the data curation (§3), model training (§4), and user experience (§5). We then evaluate our work through online and offline results (§6) and a discussion of usage patterns (§7). We conclude by examining threats to validity (§8), related work (§9), and directions for future work (§10).

2 Problem Statement

Let a source code file consist of a sequence of tokens, $F = (t_1, t_2, \cdots, t_n)$. Let $S = (s_1, s_2, \cdots, s_m)$ be the sequence of tokens corresponding to the snippet of code that will be pasted into F. Let $[L_{\text{start}}, L_{\text{end}}]$ be the range of lines where S will be pasted. If $L_{\text{start}} == L_{\text{end}}$, pasting consists of insertion, while if $L_{\text{start}} < L_{\text{end}}$, pasting will replace the existing contents.

Let $F^{'}$ correspond to F with S pasted at the target $[L_{\text{start}}, L_{\text{end}}]$ location. The task of Smart Paste consists of replacing a subset of tokens within $[L_{\text{start}}, L_{\text{end}}]$, producing $F^{''}$, such that the pasted content matches the surrounding context's syntax and semantics.

This definition of Smart Paste is similar to that presented in past work [2, 15], but critically, we allow for unrestricted token replacements, whereas previous approaches were often limited to variable names. Tackling this problem statement at scale gives rise to two key challenges:

High-Quality Suggestions. Generating useful suggestions requires a model that can navigate the inherent ambiguity of the paste editing task while maintaining high predictive accuracy across the many programming languages used in Google's monorepo.

Productionization Challenges. Beyond predictive accuracy, a successful deployment of Smart Paste at Google requires serving low-latency suggestions within a non-intrusive interface that allows developers to easily review edits. This builds the developer trust essential for achieving a high acceptance rate [7].

In the remainder of this paper we discuss how we addressed these two core challenges to deliver a feature that is used daily by thousands of Google developers.

3 Data Collection

We first focus our discussion on data collection, which was essential for addressing the core product challenges of achieving *high-quality suggestions* and *robust multilingual support*.

A primary hurdle was the availability of relevant data. While a small dataset was sufficient for initial feasibility tests, our early explorations demonstrated that existing models produced mixed results and were insufficient for a reliable, low-latency production feature. This analysis made it clear that achieving the required accuracy demanded a pivot to a specialized fine-tuned model (discussed further in §4), which fundamentally scaled the data collection challenge from tens of examples to potentially hundreds of thousands.

This new scale required a data pipeline that could capture real-world developer copy/paste edit behaviors across many programming languages. Prior research [6] suggested we could generate this via synthetic datasets and fill-in-the-middle representations, an approach successfully used on previous features. In this approach, we would artificially create a set of "incorrect" or "incomplete" pastes and make the model predict the original correct code. However, at Google, most developers use an internal IDE called Cider, which provided us with a unique opportunity to collect real, detailed data about their coding process.

Thus, we evolved our logging system to record the entire coding journey (illustrated in Figure 2) using two main entry types:

(1) **File Snapshot**: captures the file content at a specific point in time such as on saving or when closing the IDE.
(2) **Edit Delta**: records every small change made after a snapshot.

Even though our new logging system could reconstruct edit history, a big challenge remained: the logs did not explicitly specify an edit's provenance; for example, whether an edit came from a paste, a code completion, or manual typing. Figuring out if something was a paste was not straightforward and remained a technical hurdle. We now discuss our rule-based solution to this problem.

3.1 Identifying Paste and Related Edit Actions

To create our dataset, we developed a rule-based method to identify *paste and fix* sequences from raw edit logs. First, we identify a *paste candidate* as a bulk insert of 5-10 characters spanning 1-5 non-empty lines, distinguishing it from regular typing.

Once a paste candidate is identified, we define the subsequent *fix* as any follow-up edits that start within the range of the original paste. Using this location-based heuristic, our analysis shows that 72% of all paste events receive a local fix. Our edit tracking stops as soon as an "unrelated edit" occurs outside this region (see Figure 2). However, this heuristic can be imperfect; it may prematurely stop tracking if a user momentarily edits another part of the file before returning to finish fixing the paste (we made an exception for edits that added new imports, which often occur outside the pasted code). Consequently, while not every collected example can be guaranteed to represent the user's complete and final fix, we mitigate this risk by collecting millions of examples.

Manual checks on approximately 200 samples showed that over 80% were plausible paste-and-fix sequences, which we define as edits a human would judge as reasonable without performing compilation or execution checks. This fulfilled the main goal of our data

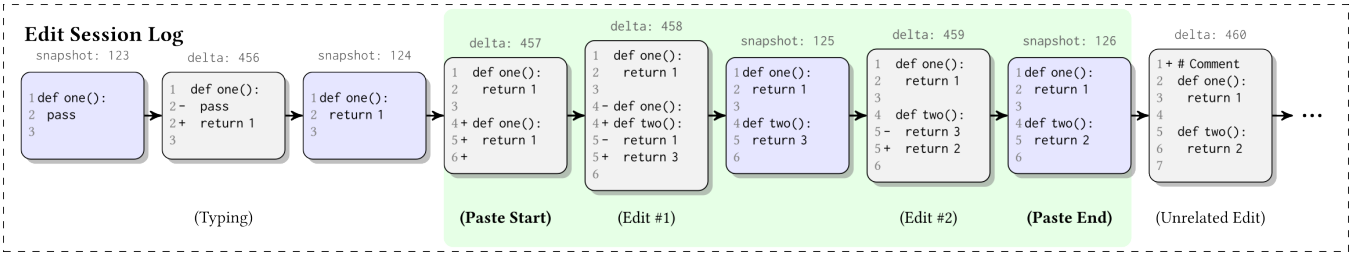

Figure 2: A developer's coding journey, reconstructed from File Snapshot and Edit Delta events. Our method identifies a Paste Start and tracks related edits to find the final Paste End state. The goal of Smart Paste is to predict this final state given only the initial paste.

collection heuristic: to capture data that gets the developer significantly closer to their intended state, aligning with Smart Paste's primary goal of reducing the keystrokes required to integrate pasted code.

3.2 No-Edit Pastes

A key requirement for high-quality suggestions is for the model to learn *when not to propose a fix*. Initially, models trained solely on examples where fixes were always applied produced an unacceptably high rate of unwanted suggestions. Our analysis showed that for 28% of paste events, no subsequent local edit was detected. This set is a composite group, including both genuinely perfect pastes and those with non-local fixes that are outside the scope of our Smart Paste feature. Since our heuristic cannot capture these non-local changes, both scenarios are treated as *no-edit* examples in our final dataset. Including this broad set of examples was essential for decreasing the overall suggestion rate and increasing the user acceptance rate.

3.3 Curating a high-quality dataset

Our final dataset aggregates millions of recent code edits, with over *7 million training examples across 24 different programming languages*. We applied several criteria filters to this data to improve its quality:

- Pastes were limited to a maximum of 20 lines to exclude large edits and ensure suggestions fit within the IDE viewport, a UX limitation of our initial solution.
- Third-party licensed code and code with undetermined provenance were excluded to comply with copyright and to bias suggestions towards Google's internal style guides.
- Logs older than four months were excluded to avoid suggestions involving deprecated code and obsolete libraries.
- We removed examples longer than 50,000 characters to filter out large data dumps or auto-generated files not representative of daily development.

4 Model Training

In this section, we describe the training process for teaching a transformer-based model to perform paste edits and how we addressed important challenges.

4.1 A Unified Model

To be successful across Google's diverse development environment—with its many programming languages and use cases (e.g., bug fixing, feature development)—Smart Paste must provide meaningful suggestions while remaining fast and easily deployable. These requirements led to two key architectural constraints: 1) the feature must be powered by a single, unified model to avoid the complexity of maintaining separate models per language, and 2) the model must be small enough to support fast inference that meets developer expectations for in-IDE suggestions. We now discuss how our modeling strategy satisfied these constraints.

4.2 Model Choice and Task Representation

To meet our requirements for a single, fast, and multilingual solution, we chose to fine-tune a small version of Gemini pre-trained on Google code [11]. To effectively leverage this pre-training, we aligned our fine-tuning task to a familiar code transformation format. As shown in Figure 3, the input prompt combines a task prefix [21] (e.g., `paste foo.py`), the surrounding code context, the pasted content wrapped in special delimiter tokens, and a final instruction for the model to predict the necessary edits.

Input

```
paste foo.py
def one():
 return 1

<STARTPASTE>def one():
 return 1
<ENDPASTE>
What changes, if any, are needed?
```

Target

```
@@
-  <STARTPASTE>def one():
+  def one():
-    return 1
+    return 2
-  <ENDPASTE>
+
```

Figure 3: Task representation for the Smart Paste model using the example from Figure 2 – the output unidiff format allows us to represent edit suggestions and no-edit suggestions (not shown) in the same way.

The target output is a unidiff-style [14] patch that relies on the special delimiter tokens—rather than line numbers—to apply the edit. This approach minimizes the number of generated tokens, which helps reduce latency. Additionally, this format unifies the task for both pastes that require edits and no-edit instances. Our

experiments confirmed that having a consistent output representation for both edits and no-edits was essential for achieving good performance across both scenarios.

4.3 Capturing Non-Local Context while Minimizing Latency

While large context windows are beneficial, including entire files can introduce unacceptable latency for an in-the-flow feature like Smart Paste. However, important non-local context, such as library imports, is often required to correctly adapt pasted code. To balance this trade-off, we restrict input sequences to 4096 tokens and employ a greedy algorithm (Algorithm 1) to strategically select the relevant local and non-local code for the model.

Algorithm 1 Greedy Context Selection

Input: L, the sequence of file lines where L_i is the ith line; R, the paste region; B, a token budget set to 4096 in our experiments.
Output: C, a set of lines from L representing the final context.

procedure BuildContext(L, R, B)
  $l_s, l_e \leftarrow \min(R), \max(R)$ ← First and last line of paste
  $C \leftarrow \{L_0\} \cup \{L_i \mid i \in R\}$ ← Seed context
  if Tokens(C) $> B$ **then return** $\emptyset$
  end if
  $p_h, p_p, p_s \leftarrow 1, l_s - 1, l_e + 1$ ← Initial expansion pointers
  loop
    $C_{prev} \leftarrow C$ ← Snapshot context to detect changes
    if $L_{p_h} \notin C \wedge \text{Tokens}(C \cup \{L_{p_h}\}) \leq B$ **then**
      $C \leftarrow C \cup \{L_{p_h++}\}$
    else if $L_{p_p} \notin C \wedge \text{Tokens}(C \cup \{L_{p_p}\}) \leq B$ **then**
      $C \leftarrow C \cup \{L_{p_p--}\}$
    else if $L_{p_s} \notin C \wedge \text{Tokens}(C \cup \{L_{p_s}\}) \leq B$ **then**
      $C \leftarrow C \cup \{L_{p_s++}\}$
    end if
    if $C = C_{prev}$ **then break**
    end if ← Exit if no lines were added this round
  end loop
  return C
end procedure

This process begins by seeding the context (C) with the paste region (R) and the file's first line (L_0). The context is then iteratively expanded by greedily adding lines using three pointers: one moving downwards from the file's start (p_h), one moving upwards from the start of the paste region (p_p), and one moving downwards from the end of the paste region (p_s). This continues until the token budget ($B = 4096$) is met or no more lines can be added ($C = C_{prev}$). The pointer starting from the top (p_h) is crucial for capturing headers, which contain critical information such as library imports. This context-building approach is used consistently during both training and inference.

4.4 Multilingual Training

As mentioned previously, we chose to use a model that was previously pre-trained on code at Google to better serve predictions across the many contexts in which Google engineers write code. We first experimented by fine-tuning our model exclusively on Python data. This initial model showed surprisingly strong out-of-sample performance on other general-purpose languages, likely due to knowledge transfer from pre-training and the similarity of common paste edits across languages. However, its performance on domain-specific languages, such as a variant of CSS, was insufficient.

To address this, our final approach utilized a mixed dataset collected across 24 different languages. The proportion of each language in the training data was weighted by its real-world paste frequency by Google developers over a one-month period. Our evaluation shows this multilingual training strategy successfully improved performance across all languages, including the previously underperforming languages, without causing any performance regression on Python.

4.5 Data Batching

During training, we observed that the configuration of training batches was important. Specifically, we included a mix of Smart Paste tasks that require edits and those that do not require edits – the latter account for approximately 30% of examples per batch. Batches also include a mix of language examples. Both of these resulted in larger batch sizes (32 and 64) and resulted in better performance.

4.6 Inference

To make suggestions for new paste instances, we initially scored each potential output diff by its probability under the trained model and used a threshold to reject low-probability suggestions. However, we found that gating suggestions based on probability was no longer necessary due to model improvements and so later model versions allowed us to remove the threshold altogether.

5 User Experience and Interface Design

The Smart Paste user experience must be discoverable, non-intrusive, and keep developers in a "flow state" while clearly presenting changes. These criteria guided our design through several iterations to produce our final user interface. We discuss alternative designs and their tradeoffs in detail in Section 6.4.

5.1 Inline Diff Suggestions

We chose to render suggestions as inline code diffs, italicizing the entire suggestion, with insertions shown in grey text and deletions marked by a strikethrough. For highlighting, we applied a yellow tint to the entire modified line. This single "alert" color—unlike standard green (ok) or red (error)—draws user attention to all changes, even those outside their immediate review periphery, communicating that the suggestion requires careful inspection. While this approach had known limitations, such as an inability to render new-line additions (as decorations only apply to existing lines) and difficulty interpreting diffs with many changes, an internal user experience study confirmed it was still developers' preferred design, striking an effective compromise for an inline diff experience.

Given our UX requirements, and based on user research, we prioritized a non-intrusive experience that kept developers in their flow above all else, even at the expense of getting fewer acceptances. Thus, we initially hid suggestions immediately if the user moved the cursor, typed, or scrolled the paste range out of view.

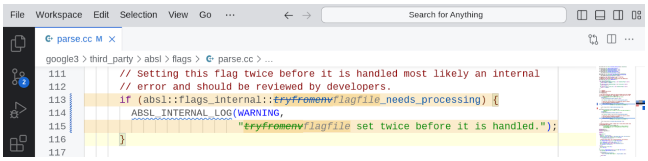

Figure 4: Our user interface renders suggestions as an inline diff with yellow highlighting. This design allows for easy review, which is critical for building the developer trust needed to drive feature adoption.

However, this initial approach, while intending to be non-intrusive, was too aggressive and we found that users frequently saw a helpful suggestion but accidentally dismissed it before they could accept. We refined this behavior by ensuring suggestions remain visible for a minimum of 1.5 seconds–later increased to 2 seconds–after cursor movements or scrolling. This small change yielded a surprising 30.4% increase in our acceptance rate, demonstrating that the original ephemeral design was preventing many users from acting on valid suggestions.

Additionally, early testers reported that the model would occasionally suggest deleting the entire pasted snippet, which they found counterintuitive and frustrating. We implemented a simple post-processing rule to suppress any suggestion that results in a full deletion of the pasted code, while still permitting suggestions that include partial deletions.

5.2 Low Latency Suggestions

A key challenge for Smart Paste is delivering suggestions quickly enough to be useful without disrupting the user's workflow. We identified that the primary latency bottleneck was the model's prefill stage (processing the input), which stems from the asymmetric nature of the task: the input code context is typically much larger than the small output diff. To mitigate this, we deployed our model using a disaggregated serving infrastructure that separates the compute-intensive prefill stage from the less demanding decode stage. Specifically, we configured our serving slices with a 2:1 prefill:decode ratio, effectively dedicating twice the compute capacity to processing the long input prompt and significantly reducing end-to-end latency.

6 Results

We evaluate the Smart Paste feature at Google by addressing the following research questions:

- RQ1: To what extent does the Smart Paste feature deliver effective suggestions to Google developers?
- RQ2: Does Smart Paste perform effectively across multiple languages in production, and what is the impact of multilingual training data?
- RQ3: To what extent can the Smart Paste model effectively distinguish between cases that require suggestions and no-edit pastes, which require no further changes?
- RQ4: What are alternative user interface designs for Smart Paste and what are their downsides?

Here, RQ1-RQ3 directly evaluate our first core challenge of delivering high quality suggestions across a multilingual monorepo. RQ1 also addresses productionization challenges by measuring the effectiveness of the end-to-end system. RQ4 provides a deep dive into the user interface design trade-offs.

To answer these questions, we rely on three primary sources of data: (1) Google's production telemetry, measuring real-world feature usage; (2) a held-out test set of 73,000 paste examples, carefully constructed to match the training set's language distribution and prevent file-path overlap; and (3) qualitative feedback on user interface experiences. A detailed definition of our evaluation metrics is presented in Table 1.

Metric	Definition
	Online Telemetry
Acceptance Rate	The fraction of suggestions accepted by users.
Average Chars Modified	Average characters changed per accepted suggestion, measured by Longest Common Subsequence (LCS) distance.
Average Chars Added	Average new characters added per accepted suggestion.
Survival	The mean ratio of added characters that stayed in the file 30 minutes after the paste.
Throughput	The queries per second (QPS) served.
Latency	The total time from the paste event to the suggestion being rendered in the IDE.
	Offline Test Set
Exact Match	The percentage of model predictions that perfectly matched the ground-truth fix.
chrF	Median character 6-gram F-score that measures partial overlap with the ground truth. We computed only on incorrect predictions to assess the quality of near-misses.
Recall	The fraction of paste instances that require an edit where our model predicts a suggestion.

Table 1: To assess the performance of our system, we used a suite of metrics divided into two categories: online telemetry and offline test set evaluation.

6.1 RQ1: To what extent does the Smart Paste feature deliver effective suggestions to Google developers?

We assess the feature's effectiveness in Google's IDE (Cider) using online deployment metrics from a stable four-month period. This timeframe was selected to avoid confounding variables from other significant infrastructure changes.

The key performance indicators shown in Table 2 highlight the feature's successful deployment. Developers accepted 45% of all suggestions, and each acceptance saved them significant effort by modifying an average of 22 characters—a metric that captures the full scope of transformations including deletions and replacements,

not just additions. Our survival metric demonstrates the durability of these edits: 58% of characters added by Smart Paste remain in the file after 30 minutes, which compares favorably to the 54% survival rate for standard code completion. At Google's enterprise scale, this impact is substantial: *accepted characters from Smart Paste account for over 1% of all code written company-wide.*

Critically for an in-IDE tool, suggestions were delivered rapidly. The median end-to-end latency was just 346 ms, which includes 257 ms for model inference and ~100 ms for system overhead. The system also proved capable of handling enterprise-scale demand, with a peak throughput of over 20 queries per second (QPS).

Metric	Cider	SQL-IDE	Colab
Acceptance Rate	45%	49%	35%
Avg. Chars Modified	~22	~18	~20
Avg. Chars Added	~15	-	-
Survival (30 min)	~58%	-	-
Throughput (peak)	>20 QPS	>5 QPS	>2 QPS
Latency (median)	346 ms	350 ms	350 ms

Table 2: Comparison of key online performance metrics for Smart Paste across three distinct developer environments: the primary Cider IDE, the SQL scripting tool, and the Colab notebook environment. These results highlight consistent value (20+ chars per acceptance) and low latency (350 ms).

With these measurements, we conclude that (1) Smart Paste suggestions are actively used by Google developers, (2) these suggestions save them a substantial number of keystrokes, (3) the serving infrastructure is able to fulfill both high throughput and low latency requirements needed for an in-IDE feature.

Generalizing to Other Surfaces. To validate the generalizability of our approach, we integrated Smart Paste into two distinct developer platforms: Colab (a Python notebook environment) and an internal tool for SQL-like scripting. Critically, both integrations leveraged the same underlying model and backend infrastructure, requiring only frontend work to visualize the suggestions. As shown in Table 2, the feature performed strongly on both surfaces. The SQL-IDE achieved a significantly higher acceptance rate (49%) than Colab (35%), though the core utility was consistent, with both saving developers around 20 characters per acceptance at the same low latency. Average characters added and 30-min survival were not collected because these frontend integrations were scoped to prioritize characters modified as the core measure of user effort saved.

We hypothesize that the discrepancy in acceptance rates between these environments is attributable to the structural complexity of the Colab notebook environment. While our method of concatenating all cells provided the model with a single, unified context, the multi-cell format introduced challenges on the frontend. Applying a suggestion that added or removed lines could require dynamically expanding or contracting a cell's boundaries, which is a more complex rendering task than editing a standard text file. This frontend complexity, in contrast to the self-contained and structured nature of SQL scripts, likely contributed to a lower acceptance rate in Colab. Despite the unique frontend challenges presented by the notebook environment, this successful integration into diverse IDEs still demonstrates a key architectural advantage of our design: a single, powerful model backend can provide a consistent AI feature across a diverse ecosystem of tools, maximizing impact with minimal incremental engineering cost.

6.2 RQ2: Does Smart Paste perform effectively across multiple languages in production, and what is the impact of multilingual training data?

For Smart Paste to be a successful feature, it must deliver high-quality suggestions across the many programming languages used at Google. As shown in Table 3, online production metrics confirm that the feature achieves consistently high acceptance and 30-minute survival rates across a wide variety of languages.

We find that the feature achieves high acceptance rates for core languages like C++ (47.8%), Java (44.6%), and Python (45.4%). This strong performance extends across the board, as the diverse, aggregated groups of less common languages also show consistently high acceptance rates, proving the feature's tangible value to developers working in different parts of the codebase.

	Acceptance (%)	Survival (%)
Popular Languages		
C++	47.8	59.1
Java	44.6	58.3
Python	45.4	60.3
Less Common Languages		
Build & Config.	42.7	55.4
Data & Query	40.5	65.8
General-Purpose	44.4	55.9
Web Development	42.5	56.9

Table 3: Online acceptance rate and survival metric for Smart Paste in Cider over 16 high-volume languages (groups are aggregated with average) show that a high fraction of suggestions are accepted and a large portion of the resulting characters survive for over 30 minutes.

Recall from Section 4, that our Smart Paste model was trained on multilingual data. To measure the impact of multilingual training data, we carried out an offline evaluation where we compare the base Smart Paste model with one trained exclusively on Python. This analysis was performed offline, as our goal is always to deploy the best-performing model to users.

As summarized in Table 4, a model fine-tuned only on Python achieves a respectable cross-language capability, with a 45.1% overall exact match from our initial data. However, training on multilingual data provides a significant and consistent performance lift across all language categories. We do not observe any regression in Python, in fact this provides a marginal 0.8 percentage point improvement in overall exact match accuracy. The multilingual

training impact is most pronounced in diverse language categories, with both the Build & Configuration and General-Purpose groups seeing an average performance increase of 3.9 percentage points. This confirms that multilingual training is a critical factor in the feature's ability to serve a diverse developer base effectively.

	Edit		No-Edit		Overall	
	Python	All	Python	All	Python	All
Popular Languages						
C++	27.9	31.6	74.9	78.8	41.8	45.6
Java	29.5	35.3	74.6	79.1	43.1	48.5
Python	29.0	29.7	81.1	82.3	47.0	47.8
Less Common Languages						
Build & Config.	29.2	32.5	81.8	86.8	49.4	53.3
Data & Query	30.5	32.0	80.0	83.1	49.1	51.2
General-Purpose	30.2	33.9	78.3	82.9	45.9	49.8
Web Development	29.4	34.3	81.7	82.5	46.1	49.8
Overall	29.4	33.5	78.2	81.9	45.1	49.0

Table 4: Exact match accuracy comparison between a Python-only model and the multilingual production model on the held-out test set shows training on multilingual data improves predictions.

6.3 RQ3: To what extent can the Smart Paste model effectively distinguish between cases that require suggestions and no-edit pastes, which require no further changes?

Table 5 also shows the exact match accuracy between paste instances that require an edit and those that do not. We observe that the exact match accuracy for no-edit (i.e. cases where the model should predict an empty diff), is high across languages and groups, when trained on multilingual data.

In addition, Table 5 shows that the Smart Paste model is able to recover most of the paste instances that require edits – with an overall recall of 69.8%. We also find that when the model output is incorrect based on exact match, the median chrF is high [20]. This indicates that incorrect suggestions still maintain a high degree of similarity to the ground truth, rather than generating arbitrary or irrelevant code.

Build and configuration languages demonstrate the highest overall accuracy (53.3%), no-edit accuracy (86.8%), and recall (79.0%). We attribute this to the structured syntax and common usage patterns in these files, such as string manipulations for correcting file paths.

6.4 RQ4: What are alternative user interface designs for Smart Paste and what are their downsides?

During development, we explored several alternative UI designs to find the best way to render and interact with suggestions. We ultimately discarded each due to key trade-offs that failed to meet our core criteria of being discoverable, clear, and non-intrusive.

	Accuracy			Recall	chrF
	Edit	No-Edit	Overall		
Popular Languages					
C++	31.6	78.8	45.6	61.4	95.1
Java	35.3	79.1	48.5	69.0	95.1
Python	29.7	82.3	47.8	66.1	94.8
Less Common Lang.					
Build & Config.	32.5	86.8	53.3	79.0	95.0
Data & Query	32.0	83.1	51.2	74.2	94.7
General-Purpose	33.9	82.9	49.8	70.3	95.4
Web Development	34.3	82.5	49.8	69.0	94.8
Overall	33.5	81.9	49.0	69.8	94.9

Table 5: Offline metrics for the multilingual model. The model recovers a majority of required edits (69.8% recall), and the high median chrF on incorrect suggestions, indicates that even the model's mistakes have a substantial overlap with the ground-truth solution.

Auto-Apply with a Hint: Convenient but Intrusive. We first considered an "auto-apply" design (Figure 5), similar to auto-imports, hoping convenience would outweigh the disruption of occasional reverts. However, a user experience (UX) study found that even infrequent incorrect suggestions were highly disruptive and eroded developer trust. Participants universally preferred having control over the suggestion, a pattern consistent with other AI features in Cider. This led us to pivot to user-action-based designs.

Figure 5: The "auto-apply with hint" design automatically applied suggestions, but this intrusive nature eroded user trust and disrupted developer flow, especially when incorrect.

Code Actions: Unintrusive but Undiscoverable. Leveraging the native VS Code Code Actions [17] (the "light bulb" icon) provided an unintrusive entry point (Figure 6). After interaction with the light bulb, a diff view would open in a new tab, showing the change to the pasted content as seen in Figure 7.

While non-intrusive, this approach was inconvenient, requiring multiple interactions (open dropdown, select "Paste Fix", review, accept) that disrupted the user's flow. Furthermore, UX studies for other AI features confirmed the lightbulb icon lacks discoverability.

Peek Suggestions: Easy To Review but Disruptive. We also considered using VSCode Peek View [16] to show a line-by-line diff in a window beneath the cursor, displaying insertions and deletions and allowing user modification, as shown in Figure 8.

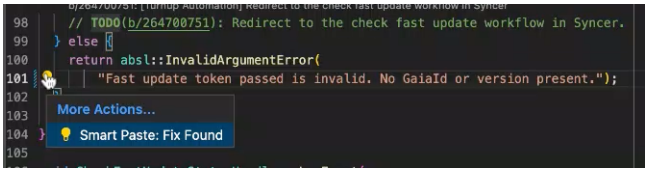

Figure 6: Using the native "light bulb" as an entry point was non-intrusive, but suffered from poor discoverability and led to a disruptive, multi-step review process.

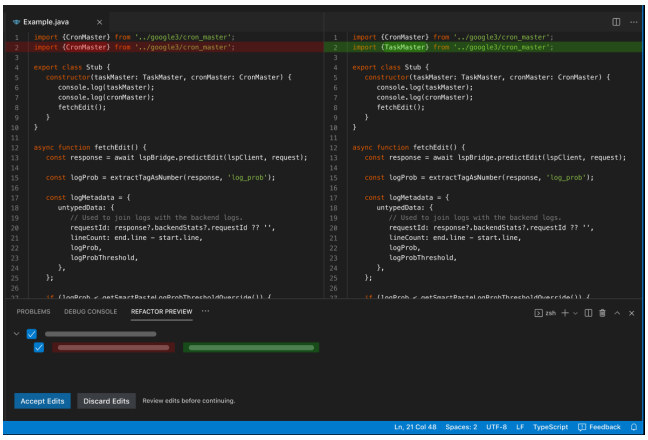

Figure 7: The side-by-side diff view that resulted from the Code Action forced a context switch to a new tab for review, proving too disruptive to the developer's workflow, which was a significant barrier to adoption.

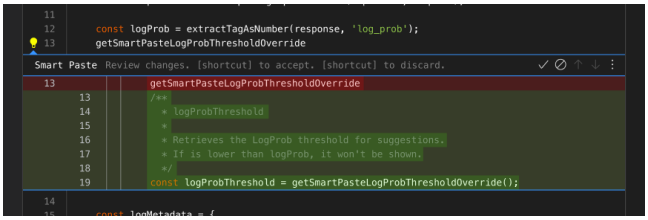

Figure 8: The Peek View suggestion window. Although capable of showing complex diffs and supporting user modifications, this UI was too heavyweight and disruptive for the common case of simple paste edits, creating a poor user experience.

We ultimately discarded this approach. The flexibility for complex diffs was deemed too cumbersome for simple insertions, and we identified that most paste fixes involved small partial changes, not complete rewrites which would have benefited by this type of rendering.

Inline Ghosting: Lacked Replace Support. Another option was to leverage the pattern from the widely known and successful AI Code Completion feature [26], an in-flow experience where suggestions are rendered as grey, slanted text accepted via [TAB] (Figure 9).

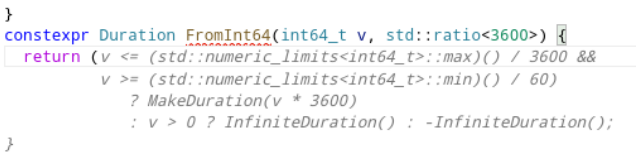

Figure 9: The inline "ghost text" pattern, adopted from code completion was ultimately unsuitable because it could not represent replace or delete operations, fundamentally limiting the quality and scope of possible suggestions.

Unfortunately, this pattern did not effectively support replace or delete operations, only inserts. This would have significantly hindered Smart Paste's functionality and conflicted with the existing code completion feature. Ultimately, we opted to adapt this pattern with major adjustments to overcome the challenge of representing replacements.

7 Discussion

Deploying Smart Paste at an enterprise scale revealed several emergent developer behaviors and sophisticated usage patterns that evolved as users became more familiar with the feature.

Dependency and Filepath Management. A primary use case involved developers pasting raw file paths into code, relying on Smart Paste to automatically format them into correct import statements as shown in Figure 10. We also observed the reverse, where import statements were pasted into BUILD files and correctly translated into the required dependency syntax. The model proved highly effective at resolving incomplete paths, streamlining a typically tedious and manual process.

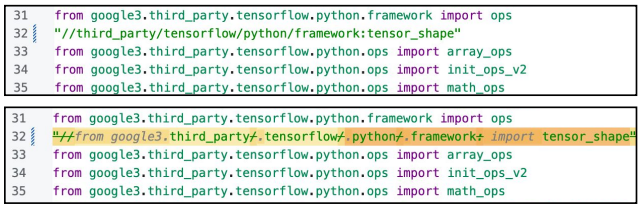

Figure 10: After deployment, users employed Smart Paste in unexpected ways, such as converting package dependencies into language imports

Lightweight Repetitive Refactoring. Beyond simple variable renaming—a common task cited in prior work—we observed a novel "chaining" behavior. Developers would execute a rapid sequence of copy-paste-accept actions to apply incremental changes across multiple lines, such as renaming sequential protobuf fields. This effectively transformed Smart Paste into a tool for lightweight, repetitive refactoring tasks. Figure 11 shows one such example, where a user creates a variant of an existing method using Smart Paste.

API Lookup Mechanism and Quick Syntax Fixes. Similar to code completion, developers sometimes use Smart Paste to look up common APIs without needing to open documentation or browse

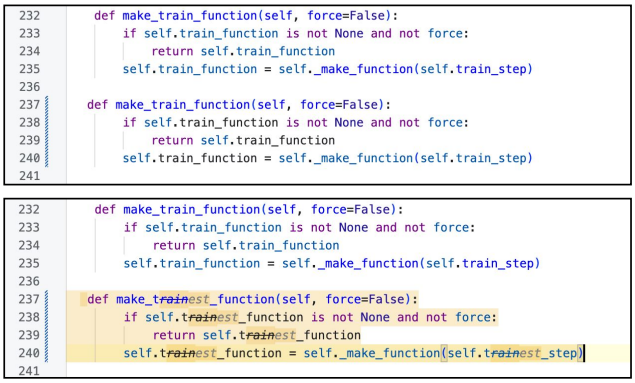

Figure 11: Developers employed chains of pastes and Smart Paste suggestions to function as a lightweight refactoring tool.

existing code. Specifically, developers paste incomplete or pseudo-code like fragments and rely on Smart Paste edits to make them functional. This is particularly useful across domain-specific code editors (e.g., Colab and SQL-IDE), where it is common to copy/paste entire contents. This allows developers to express their intentions in a pseudo-code format and then quickly convert the entire block into working code with just a few keystrokes.

Context-Aware Cross-Language Translation. The unified multilingual model not only translated code between different languages (e.g., from one scripting language to another) but also demonstrated the crucial ability to discern when not to translate. In files containing mixed languages by design, such as Markdown with embedded Python snippets, the model correctly identified the context and preserved the pasted code's original language, avoiding erroneous transformations.

Workflow Adaptation and Partial Acceptance. A key finding was that the feature's acceptance rate increased over time with no changes to the underlying model. We observed that developers began to accept suggestions that were imperfect but directionally correct, as making a small manual tweak to the suggestion was faster than fixing the original paste. We hypothesize that this "partial acceptance" phenomenon is evidence of users building trust, where they integrate the tool into their workflow to minimize effort rather than to achieve a perfect, one-shot fix.

8 Threats to Validity

The results presented in this work reflect conditions of a large software company and may not generalize to other settings (e.g. outside of enterprise environments). Similarly, many of our design choices were driven by specific requirements, such as delivering low latency paste edit suggestions. We now describe the main factors to consider for the generalizability of this work to different settings.

Collecting the dataset needed to effectively train the Smart Paste model was facilitated by keystroke-level telemetry enabled in our in-house IDE (Cider). Other enterprise settings may not have access to such granular data, and so curating a similarly high-quality dataset may require adapting or extending the heuristics presented in this work. This same telemetry allowed us to measure the impact of our feature on developers' typing, and we used this information to evolve it. Without access to such measurements, developing a similar tool may require additional metrics for success.

We chose to train our model on paste actions in a variety of programming languages. This choice was driven by the need to serve the same feature in a large multilingual codebase. Other settings, where a single programming language dominates, may not require such multilingual training and could be simplified.

Our user interface built on prior experience with related coding assistance features, such as code completion. Importantly, users could receive Smart Paste suggestions through a familiar interface and did not need to be aware of completely new modes of interaction. Releasing a Smart Paste feature in a different editor may require a different user interface to accommodate existing code features. Indeed, as we discuss in our evaluation section, we generalized Smart Paste to a computational notebook and a SQL-like IDE by reimplementing the frontend components, applying lessons learned from our main experience.

9 Related Work

Code completion is one of the most popular and successful applications of large language models for code authoring. Industrial code completion offerings include Github Copilot [12], Amazon CodeWhisperer [3], and Google Gemini Code Assist [13], among others. While classical code completion has typically been based on a combination of formal program analysis [19] and statistical modeling [23], most modern code completion engines now employ LLM-based predictions and have shown to scale to whole-line and multi-line code suggestions. Such LLM-based code completion features have been successfully employed in industrial settings [10]. Similar to this line of work, we employ an LLM to provide real-time code authoring assistance. We share challenges such as the need for low-latency responses. However, in contrast to code completion, which typically suggests only a suffix of the current code, our work is focused specifically on automatically integrating copy/paste code into a new context, which may require edits beyond suffix completion. We showed that this interaction raises unique user experience challenges that led us to exploring diverse interface designs.

LLMs have previously been successfully used to perform targeted code edits. Tufano et al. [29] framed code editing as a neural machine translation task. Chakraborty and Baishakhi [9] treat code editing as a machine translation task, but provide a natural language hint as an additional modality. Nam et al. study this type of natural language-based code transformation within Google [18], based on a popular internal code transformation feature. Similarly, Smart Paste is an LLM-based code editing feature. However, Smart Paste is scoped to paste events, only edits paste regions, and does not use any additional hints, instead relying exclusively on the model's training and the context of the paste to suggest edits.

Closely related to adaptive source code pasting, Amidon et al. [4] introduced the concept of code transfer within the context of program repair. Their work shows how to automatically replace or combine code from multiple applications to eliminate program defects. Sidiroglou et al. [24] employed a combination of dynamic and static analysis to further refine the code transplanted between

two programs. Our work similarly explores how to adapt code to a new context, adjusting expressions as necessary. In contrast to code transfer work, our use case is real-time code authoring assistance, which limits the use of techniques from dynamic analysis, and we instead rely on the predictive power of LLMs.

Allamanis and Brockschmidt [2] defined Smart Paste as a standalone code assistance task and showed that deep learning could be used to successfully adapt variable usage in C# code snippets pasted into a new context program. Liu et al. [15] integrated a similar form of intelligent copy-paste – using a transformer model pre-trained on a dataflow-based task – into an IDE. Their evaluation showed this approach could adapt Python code snippets with high accuracy and a user study found that this feature could reduce the time spent performing code copy/paste.

Our work builds on this line of research and shows that Smart Paste can be successfully delivered as an actively used feature in an industrial context. Doing so led to addressing multiple key challenges not previously tackled. First, our Smart Paste work had to support authoring in many programming languages, while past work had focused their evaluation C# and Python. Second, a smooth user experience is critical for deployed features used by tens of thousands engineers, and so our design had to address production serving constraints and consider user interface concerns. Third, we report on a wide range of deployment-based statistics, showing the uptake of Smart Paste among professional developers.

10 Future Work

While Smart Paste has proven highly effective at fixing code within the pasted region, we envision a series of future milestones to evolve this capability into a more comprehensive, context-aware developer assistant.

Milestone 1: Expanding Suggestion Scope. The most immediate next step is to expand suggestions beyond the strict boundaries of the pasted content. This involves generating follow-up suggestions in other locations after the initial paste and fix are accepted. For instance, the model could suggest adjusting the surrounding code in the current file to properly integrate the new snippet, modifying other dependent files to align with API changes introduced by the paste, or even updating related unit tests to ensure the new logic is covered. This expands the feature from a localized "paste fixer" to an integration assistant.

Milestone 2: Next Edits for Broader Events. A further iteration involves generalizing the trigger for "next edit" suggestions. Instead of only activating after a paste, the model could suggest the next logical code change following other high-intent events, such as the acceptance of a code completion or another AI-driven refactoring. The key advantage of this approach is that the suggestion is offered at a moment when the developer is already considering their next step, which minimizes cognitive disruption and maintains their workflow momentum.

Milestone 3: Proactive, Model-Initiated Suggestions. The third milestone aims to make next-edit suggestions proactive and independent of pre-configured events. Here, the model would learn to recognize opportune moments within the development workflow to provide suggestions based on the evolving context of the code. Rather than being purely reactive to a user action, the system would anticipate the user's needs, understanding when a suggestion for the next edit is most likely to be useful and accepted.

Milestone 4: The Next-Action Assistant. The eventual goal is to support the developer with their holistic "next action", moving beyond simple code edits to encompass the entire development process. This ultimate milestone involves assisting the user while they are in the flow of implementing a feature. A "next action" could be a complex code modification, but it could also be a command to execute a tool—like running a build or specific tests—or even triggering an autonomous agent to tackle a more complex, multi-step sub-task on the developer's behalf. The latter would link the feature to the more recent advancements in modern IDEs (e.g. Cursor [5] or Windsurf [30]) for agentic coding, where the user has less control but more complex tasks can be accomplished.

11 Conclusion

Copying and pasting code is a common activity when writing software, whether borrowing code from other parts of the codebase, documentation, or team mates assistance. However, in addition to pasting these code snippets, developers must very often adapt them to the surrounding context to produce the final desired snippet. Past work has shown that deep learning can be used to predict the edits necessary to produce these adapted code snippets. In this work, we showed how we trained and deployed such a Smart Paste feature within Google. We share how we overcame key challenges to our deployment, including details on our multilingual data collection, model training and serving choices, and user interaction design. Ultimately, this resulted in the current deployment of this popular AI-based IDE feature at Google, with 45% of suggestions accepted by developers, saving them on average 22 keystrokes.

12 Acknowledgements

This work is built on the collective contributions of our colleagues across Google Core Systems and Google DeepMind. We extend special thanks for the crucial support and advice from our team members and leadership, including Miltos Allamanis, Boris Bokowski, Satish Chandra, Cristopher Claeys, Madhura Dudhgaonkar, Alberto Elizondo, Simone Forte, Michael Golahi, Sandeep Katragadda, Ugam Kumar, Pascal Lamblin, Katie Le, Damien Martin-Guillerez, Henrik Muehe, Ambar Murillo, Stoyan Nikolov, Aditya Pandey, Siddhant Sanyam, Christian Schneider, Pavel Sychev, Niranjan Tulpule, Ilya Tzoop, Andy Yankovsky, and David Zhang.

References

[1] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

[2] Miltiadis Allamanis and Marc Brockschmidt. Smartpaste: Learning to adapt source code. *arXiv preprint arXiv:1705.07867*, 2017.

[3] Amazon Web Services. Amazon CodeWhisperer is now generally available. https://aws.amazon.com/blogs/aws/amazon-codewhisperer-free-for-individual-use-is-now-generally-available/, April 2023. Accessed: 2025-09-24.

[4] Peter Amidon, Eli Davis, Stelios Sidiroglou-Douskos, and Martin Rinard. Program fracture and recombination for efficient automatic code reuse. In *2015 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6. IEEE, 2015.

[5] Anysphere Inc. Cursor: The AI-first Code Editor. https://cursor.sh/. n.d.

[6] Mohammad Bavarian, Heewoo Jun, Nikolas Tezak, John Schulman, Christine McLeavey, Jerry Tworek, and Mark Chen. Efficient training of language models to fill in the middle. *arXiv preprint arXiv:2207.14255*, 2022.

[7] Adam Brown, Sarah D'Angelo, Ambar Murillo, Ciera Jaspan, and Collin Green. Identifying the factors that influence trust in ai code completion. In *Proceedings of the 1st ACM International Conference on AI-Powered Software*, pages 1–9, 2024.

[8] Marcel Bruch, Martin Monperrus, and Mira Mezini. Learning from examples to improve code completion systems. In *Proceedings of the 7th joint meeting of the European softw are engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering*, pages 213–222, 2009.

[9] Saikat Chakraborty and Baishakhi Ray. On multi-modal learning of editing source code. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 443–455. IEEE, 2021.

[10] Omer Dunay, Daniel Cheng, Adam Tait, Parth Thakkar, Peter C Rigby, Andy Chiu, Imad Ahmad, Arun Ganesan, Chandra Maddila, Vijayaraghavan Murali, et al. Multi-line ai-assisted code authoring. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, pages 150–160, 2024.

[11] Google Gemini Team. Gemini: A family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*, 2023.

[12] GitHub. GitHub Copilot is generally available to all developers. https://github.blog/2022-06-21-github-copilot-is-generally-available-to-all-developers/, June 2022. Accessed: 2025-09-24.

[13] Google Cloud. Duet AI in Google Cloud is now generally available. https://cloud.google.com/blog/products/application-modernization/introducing-duet-ai-for-google-cloud, May 2023. Accessed: 2025-09-24].

[14] Michael K Johnson. Diff, patch, and friends. *Linux Journal*, 1996(28es):2–es, 1996.

[15] Xiaoyu Liu, Jinu Jang, Neel Sundaresan, Miltiadis Allamanis, and Alexey Svyatkovskiy. Adaptivepaste: Intelligent copy-paste in ide. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1844–1854, 2023.

[16] Microsoft. Navigate and edit c# - visual studio code. https://code.visualstudio.com/docs/csharp/navigate-edit#_peek-definition. Section: Peek Definition. Accessed: 2025-09-18.

[17] Microsoft. Refactoring - visual studio code. https://code.visualstudio.com/docs/editing/refactoring, Sep 2025. Accessed: 2025-09-18.

[18] Daye Nam, Ahmed Omran, Ambar Murillo, Saksham Thakur, Abner Araujo, Marcel Blistein, Alexander Frömmgen, Vincent Hellendoorn, and Satish Chandra. Prompting llms for code editing: Struggles and remedies. *arXiv preprint arXiv:2504.20196*, 2025.

[19] Daniel Perelman, Sumit Gulwani, Thomas Ball, and Dan Grossman. Type-directed completion of partial expressions. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 275–286, 2012.

[20] Maja Popović. chrF: character n-gram F-score for automatic MT evaluation. In Ondřej Bojar, Rajan Chatterjee, Christian Federmann, Barry Haddow, Chris Hokamp, Matthias Huck, Varvara Logacheva, and Pavel Pecina, editors, *Proceedings of the Tenth Workshop on Statistical Machine Translation*, pages 392–395, Lisbon, Portugal, September 2015. Association for Computational Linguistics.

[21] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of machine learning research*, 21(140):1–67, 2020.

[22] Baishakhi Ray, Miryung Kim, Suzette Person, and Neha Rungta. Detecting and characterizing semantic inconsistencies in ported code. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 367–377. IEEE, 2013.

[23] Veselin Raychev, Martin Vechev, and Eran Yahav. Code completion with statistical language models. In *Proceedings of the 35th ACM SIGPLAN conference on programming language design and implementation*, pages 419–428, 2014.

[24] Stelios Sidiroglou-Douskos, Eric Lahtinen, Anthony Eden, Fan Long, and Martin Rinard. Codecarboncopy. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 95–105, 2017.

[25] Alexey Svyatkovskiy, Ying Zhao, Shengyu Fu, and Neel Sundaresan. Pythia: Ai-assisted code completion system. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 2727–2735, 2019.

[26] Maxim Tabachnyk and Stoyan Nikolov. Ml-enhanced code completion improves developer productivity. https://research.google/blog/ml-enhanced-code-completion-improves-developer-productivity/, Jul 2022. Accessed: 2025-09-18.

[27] Gemini Team, Rohan Anil, Sebastian Borgeaud, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, Katie Millican, et al. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*, 2023.

[28] Kristín Fjóla Tómasdóttir, Mauricio Aniche, and Arie Van Deursen. Why and how javascript developers use linters. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 578–589. IEEE, 2017.

[29] Michele Tufano, Jevgenija Pantiuchina, Cody Watson, Gabriele Bavota, and Denys Poshyvanyk. On learning meaningful code changes via neural machine translation. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 25–36. IEEE, 2019.

[30] Windsurf. Code suggestions powered by everything you've done. https://windsurf.com/tab. n.d.

[31] John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652, 2024.