# CoCoCoLa: Code Completion Control Language

Nhat
research@nhat.run
Formal Methods & Tools, University of Twente
Enschede, The Netherlands

Vadim Zaytsev
vadim@grammarware.net
Formal Methods & Tools, University of Twente
Enschede, The Netherlands

## Abstract

In software development, the efficiency and accuracy of code completion systems are crucial for productivity and codebase discovery. From simple spell checkers to advanced AI-powered tools, there are more ways to complete your code than ever. This results in an explosion in the number of possible valid proposals, especially when working with today's increasingly large codebases. Over the years, a lot of effort has been put into developing effective ranking systems to prioritise proposals with more potential. Yet developers still often struggle with an overwhelming number of suggestions, leading to reduced productivity and increased cognitive load.

In this paper, instead of just performing completion by name, we propose *CoCoCoLa* — an alternative approach to give back the control over the presented proposals to the developer. By investigating the recorded code completion events, frequencies of desirable code elements' properties were calculated to identify useful control factors. To avoid adding further complexity to the completion process, we propose a simple language, defined within the boundary of a valid identifier of the 50+ most popular software languages in 2024. This language allows developers to specify and filter for desired properties of the proposals.

***CCS Concepts:*** • **Software and its engineering → Domain specific languages**; *Software development techniques*; **Integrated and visual development environments**.

***Keywords:*** code completion, code completion control, code completion filtering, domain-specific language, dsl

**ACM Reference Format:**
Nhat and Vadim Zaytsev. 2025. CoCoCoLa: Code Completion Control Language. In *Proceedings of the 24th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '25), July 3–4, 2025, Bergen, Norway*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3742876.3742883

## 1 Introduction and Motivation

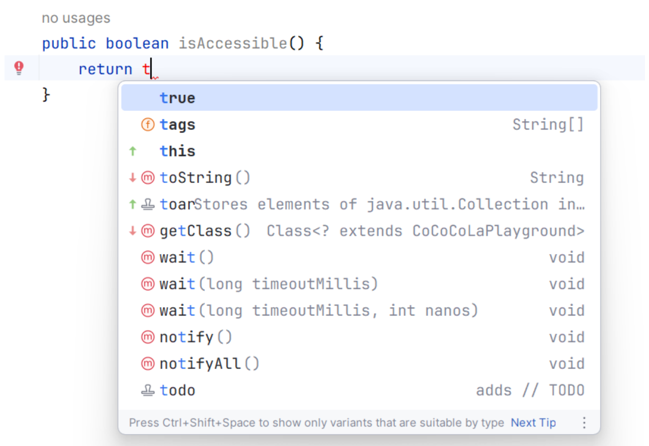
The "type less, write more" idea has become familiar to most developers whether they realise it or not, with applications ranging from query completion in modern search engines based on popular queries, to character suggestions when typing in languages with thousands of symbols in their alphabets. For code, *completion*, when triggered, commonly appears in the form of a list of suggestions based on the user's prompt and the surrounding context. The earliest code completion systems can be seen as merely spell checkers [17] by today's standards. Modern code completion systems can also collect and prioritise suggestions of words that already appeared in the file/codebase [2]. This leads to inspection systems that perform syntactic and semantic analysis on the codebase to gain an understanding of the code to provide richer and more contextually relevant completions.

Code completion has become a norm, an expectation, and a reliance for modern IDEs and code editors to have a code completion system that can offer correct types, variables, fields, methods, classes, and other language-specific constructs [27]$^{\text{(Finding 4)}}$. This kind of modern code completion is commonly known as intelligent code completion, code suggestion, code prediction, auto-completion, *IntelliSense* [13], etc., depending on the provider. Code completion significantly enhances the coding experience and productivity, while improving code quality by increasing efficiency and accuracy, promoting discoverability, reducing cognitive load and enhancing focus, and encouraging consistency and best practices.

Machine learning (ML) and artificial intelligence (AI) have been heavily invested in recently, and rapidly integrated into AI-powered code completion solutions, such as GitHub Copilot, Cody, Tabnine, JetBrains AI Assistant, etc. Relying on them leads to numerous legal and ethical issues. However, on the other hand, they provide much longer and more tailored completion based on the codebase. In this paper, we will not focus on this kind of generative code completion.

Today, code completion can do more than just suggesting valid code elements. Many approaches [6, 7, 12, 16] have been proposed to expand the capabilities of code completion, allowing code completion to become helpful in a wider range of scenarios. It is not uncommon to see in many modern IDEs that different code completion approaches are offered alongside in the same pop-up code completion suggestion box, with only a small visual indication to distinguish them from one another.

Fig. 1 shows an example of the default code completion in *IntelliJ*. While `true` and `this` are keyword completions with no icon prepending, `toar` and `todo` are template completions,



```
no usages
public boolean isAccessible() {
    return t
}
    true
    tags                                          String[]
    this
    toString()                                      String
    toarStores elements of java.util.Collection in…
    getClass()  Class<? extends CoCoCoLaPlayground>
    wait()                                            void
    wait(long timeoutMillis)                          void
    wait(long timeoutMillis, int nanos)               void
    notify()                                          void
    notifyAll()                                       void
    todo                                     adds // TODO
    Press Ctrl+Shift+Space to show only variants that are suitable by type  Next Tip
```

**Figure 1.** Default code completion in *IntelliJ*

indicated by the stamp icon, `tags` is a field completion, and the rest are method completions. On the right-hand side of the suggestion box, the type of the code element is given as additional information. However, this is not always the case, e.g., for template completions, the description of the template is shown instead. We believe this can make the code completion experience even more inconsistent and confusing for the programmer.

As more suggestions are available in a specific code context, it becomes harder to find what the developers actually desire, leading to the paradox of choice [25]. Having too many options to assess and choose from can lead to confusion, indecision, or even paralysis of choice. A recent study [11] showed that with a longer list of suggestions/proposals and the further desired element is down the list, the inspection time increases accordingly, resulting in cases where the developers forfeited the completion entirely. This study also showed that *IntelliSense* is most likely to either produce higher quality suggestion lists with only a couple of items that have 25%–50% chance of being selected, or suggestion lists with 250 items (capped by *IntelliSense*) with almost certain cancellation. The average length of the completion suggestion list is 50.37, with 250 being the second most common at 16%.

The negative effect of the explosion in the number of suggestions can be somewhat mitigated by ranking each suggestion and ordering them based on how likely they are being selected given a code context. Yet, the only way to actually narrow down the suggestion list is to further type in more part of the desired code element name [8]. Contemporary code completion systems entirely rely on: (1) the names of the code elements to analyse, find, offer, and filter for suggestions; (2) the competency of the suggestion ranking system to hopefully bring forth the desired elements. There is little to none control over what will be offered as suggestions, apart from using names.

In this paper, we focus on the realm of object-oriented languages and provide answers to the following research questions:

**RQ1** How frequently does each code element type appear in code completion suggestions? Is there a difference compared to the set of only selected suggestions?
**RQ2** Which properties of which code elements can be helpful to use as control factors for code completions?
**RQ3** What are the criteria for designing a language to control code completions?
**RQ4** What would a language to control code completions look like and behave?

Answering these questions will help us understand the current state of code completion systems and how we can improve them by providing more control over the suggestions. As developers, users have more information about the code they are writing than just the names of the code elements they want to complete. With the variety of code elements, each with its own set of properties, there is a wealth of information that is yet to be explored to gain more control over the code completion process. The current code completion systems are incapable of using these additional properties to narrow down and provide more relevant suggestions. Completion by name alone is not enough anymore as the number of suggestions increases and the code elements become more complex. While recent advancements in AI and ML have shown great promise, the quality of the generated code is still highly dependent on the training data. AI-generated code can be highly variable and unpredictable with little to no control over what is generated. This can lead to suggestions that may not align with the developers' intention. We believe that there is still a need for a more controlled approach to code completion, where developers can explicitly specify what they want in a completion.

The first step towards this goal is to identify among these properties which ones are useful as completion control factors. We investigated code elements' properties and their frequencies in real-world code completions. Based on how prevalent they are among suggestions and across completions, we can identify which properties are more likely to be useful as control factors. As a second step, we explored the design of a language that can be used to control code completions. The result of this is *CoCoCoLa* (Code Completion Control Language), a host-language-agnostic language that allows specification of different code completion control factors. *CoCoCoLa* was designed to work in conjunction with existing code completion systems without breaking the existing code completion process.

## 2 Code Completion Control Factors

Overall, the structure and process of a code completion are as follows:

1. *Triggering and initial suggestion generation:* A code completion event can be triggered implicitly (e.g., while typing) or explicitly via a user action (e.g., a keyboard shortcut) at a certain position in the code editor. From the triggered position, the code completion system collects the surrounding code context for analysis and presents the user with a list of *suggestions*, also called *proposals* or *candidates*, that are relevant to the context. The suggestions are typically ordered by their relevance to the context based on some ranking algorithms, with the most relevant ones at the top of the list.
2. *Prompting and refinement:* The user can then optionally type in a *completion prompt* to help guide the code completion system. Typically, this prompt refers to the name, or part of it, of the item the user is looking for, e.g., a method name, a field name, etc. Smarter code completion systems can also perform fuzzy matching, i.e., matching the name with the suggestions even if they are not identical. Based on the completion prompt, the suggestion list is refined.
3. *Selection and termination:* The user can *select* one of the suggestions from the list to view more details. Selecting a suggestion will not make any alterations to the code. This action can be done as many times as the user wants.
   a. If satisfied with a suggestion, the user can confirm their active selection (e.g., by pressing the Enter ⏎ or Tab ⇥ key), which will perform the necessary transformations to the code to *apply* the *completion edit* to the code. This can be as simple as inserting the selected suggestion into the code at the position where the code completion was triggered, or more complex, such as also adding appropriate `import` statements.
   b. If not satisfied with all of the suggestions, the user can either go back to the previous step to refine the suggestions further, or *cancel* the code completion event. Depending on the code completion system, continue typing, i.e., changing the completion prompt, may cancel the current code completion event and trigger a new one, since the context had changed. For other systems, this is considered as a refinement of the current code completion event.

In this work we are interested in the *prompting and refinement* step of the code completion process. Code completions are done by looking up valid code elements with the name that match a given input *completion prompt.* In a lexical-structural sense, this name is called an identifier, as its main purpose is to identify different code elements.

In many languages, identifiers are not unique. Different scopes can have code elements with identical identifiers, as they can belong to different separate scopes (e.g., packages, classes, functions, etc.), or shadowing the ones from the outer scopes. Even within the same scope, code elements with the same name can overload each other (e.g., functions with different parameter lists). It is clear that the term "identifier" is somewhat misleading in this context, as one cannot identify and differentiate between code elements just by looking at their identifier alone. This ambiguous nature of identifiers makes it difficult to refer to, display, compare, and select code elements with the same identifier while performing code completion, especially for today's ever larger codebases.

For clarity's sake, in this paper, we will refer to identifiers as just names, as opposed to fully qualified names. Fully qualified names (FQNs) are names that can unambiguously point to a particular code element, regardless of the calling context. Typically, FQNs are constructed as a hierarchical concatenation of identifiers leading up to and including that code element, e.g., `com.domain.package.subpackage.ObjectClass.set3dPosition(x:Int,y:Int,z:Int)`. As a result and a drawback, FQNs are often long and unwieldy to manually specify. Using FQNs for code completion is, therefore, not ergonomic and practical. In addition, FQNs only contain hierarchical structural information of the target code elements.

Code completion by name can only get you so far. Code elements have more properties than just their names. Although having less identifying power compared to names, the other properties can potentially help in filtering out undesired elements as control factors. This opens doors to better control over the completion results and also possibilities to perform code completion by non-name properties.

### 2.1 Data Analysis

In order to investigate the contributing factors of a successful (or applied) code completion, an analysis was performed on a dataset consisting of code completion events. A recent study by Hellendoorn et al. [5] showed that there is a large discrepancy between real-world code completions and synthetic ones for benchmarking. Therefore, we chose to perform this evaluation with the *MSR 2018* dataset to get a more realistic identification of desirable control factors.

The *MSR 2018* dataset [19] originated from the *Mining Challenge* of the 2018 *International Conference on Mining Software Repositories (MSR)* [14]. This dataset was collected by *FeedBaG++* [20] from the *KaVE* project that includes *enriched event streams* [18] of real-world user interactions in *Visual Studio* from consented participants, with an emphasis on code completion events.

In this data analysis, we focus on applied code completions and aim to count the occurrence rates of different code elements and their properties of the completion suggestions offered, compared against the corresponding properties of the applied suggestion of the same code completion. We are also interested in the set of selected suggestions as they can signal what the user was looking for. A selected suggestion is a suggestion the user interacted with, (e.g., via mouse or keyboard), and the applied suggestion (if any) is always the last selected suggestion.

More formally, we denote a code completion as $C_m$, where $m$ is some form of index to differentiate between code completion events. $C_m^A$ then denotes an applied code completion. For each code completion, the completion suggestions/proposals are defined as:

- Suggestion (ordered) set of $C_m$:
  $C_m.S = \{C_m.s_0, C_m.s_1, \ldots C_m.s_n\}$
- The $i^{th}$ suggestion of $C_m$: $C_m.s_i$
  where $\forall 0 \leq i \leq n, C_m.s_i \in C_m.S$
- Applied suggestion: $C_m^A.s_a \quad$ where $0 \leq a \leq n$
  - $\forall C_m^A, \exists! a, C_m^A.s_a \neq \emptyset$
  - $\forall C_m^{\neg A}, \nexists a, C_m^A.s_a \neq \emptyset$
- Selected suggestions:
  - Selected suggestion set: $C_m.S^X$
  - $\forall C_m.s_i \in C_m.S^X, C_m.s_i \in C_m.S$
  - $\forall C_m^A.s_a, C_m^A.s_a \in C_m.S^X$

Each code completion suggestion can be resolved into a code element the suggestion referenced to. The type of a code element, or element type for short, refers to whether it is a method, field, local variable, etc. A code element from a suggestion is defined as:

- The code element that a suggestion references to:
  $\exists! e, C_m.s_i.e$
- The code element type of a code element:
  $\exists! t, C_m.s_i.e.t$
- Set of all code element types: $T_e$
  where $\forall C_m.s_i.e.t, C_m.s_i.e.t \in T_e$
- Code element property set:
  $C_m.s_i.e.P = \{C_m.s_i.e.p_0, C_m.s_i.e.p_1, \ldots C_m.s_i.e.p_k\}$
- A property: $C_m.s_i.e.p_j$
  where $\forall 0 \leq j \leq k, C_m.s_i.e.p_j \in C_m.s_i.e.P$

Overall, we can count the number of code completions that match an element type $t$ using the following formula:

- Count matched element type $t$:
  $N_S^t = \sum_m \left|\{C_m^A.s_i | C_m^A.s_a.e.t = C_m^A.s_i.e.t\}\right|$
- Count selected matched element type $t$:
  $N_{S^X}^t = \sum_m \left|\{C_m^A.s_i | C_m^A.s_a.e.t = C_m^A.s_i.e.t \wedge C_m^A.s_i \in C_m.S^X\}\right|$

As mentioned before, each code element can have multiple properties Going further, a matched code element property $p_j$ can be counted in two ways:

- Count by completion:
  $N_C^{p_j} = \sum_m \left|\{C_m^A | \exists i, C_m^A.s_a.e.p_j = C_m^A.s_i.e.p_j\}\right|$
- Count by suggestion:
  $N_S^{p_j} = \sum_m \left|\{C_m^A.s_i | C_m^A.s_a.e.p_j = C_m^A.s_i.e.p_j\}\right|$
- $\forall p_j, N_C^{p_j} \leq N_S^{p_j}$

The code element property count by completion provides an overview on the prevalence of that property across completions, expressing how likely such a property can appear. Contrarily, the count by suggestion indicates the commonness of such a property among suggestions, hinting at its filtering capability.

Basing solely on the appearance counts of each property to determine which property can potentially be used as a control factor, creates a paradox. If a frequent property is chosen, it will likely not have much filtering effect due to how abundant that property appears in the suggestion set. On the other hand, if a less common property is chosen, it may well be too obscure to be encountered in practice.

To address this issue, we propose ranking the properties based on the ratio between their count by suggestion and their count by completion event $p_j$:

$$R^{p_j} = \frac{N_C^{p_j}}{N_S^{p_j}}$$

If a property is not present in any suggestion, its rank will be set to infinity. The lower this rank is, the more suited a property would be as a control factor. This ranking prioritises completion count, while penalising suggestion count. More intuitively, having the suggestion appearance frequencies low means that the property has more filtering power. Whereas, with a high completion appearance count, such a property is more likely found within a completion.

The *MSR 2018* dataset consists of *enriched event streams (EES)* based on the work of Proksch [18]. EES defined over a dozen of name grammar rules for different code element members (e.g., field name, method name, property name, parameter name, etc.) and types (e.g., type name, etc.), each having its own set of diverse properties.

Preliminary inspection showed that some properties of a certain code element can be constant, i.e., always return the same value regardless of the instance of that code element. Such properties provide little gain in terms of filtering the competition suggestions. In addition, there are properties equivalent to each other within a certain code element. In many cases, the extra properties help to clarify and increase understandability, e.g., the value type and return type of a method. By investigating the implementation details of the `cc.kave:commons` library that contains common data structures to support the *KaVE* project, the following cases were discovered:

Constant properties:

- array type name: declaring type, is nested type, is void type, is simple type
- delegate type: is simple type, is void type
- predefined type name: has type parameters, type parameters (+ type parameter count), assembly, namespace, is enum type, is nullable type, is interface type, is nested type, declaring type
- type name: is simple type, is void type

Equivalent properties:

- event name
  - event : member.value type (base)
    event.handler type (return value type)
- method name
  - method : member.value type (base)
    method.return type (return value type)
- property name
  - property.is indexer (same implementation)
    property.has parameters (same implementation)
- predefined type name
  - predefined type : base type.is reference type (base)
    predefined type : base type.is struct type (return not is reference type)
    predefined type : base type.is value type (return is struct type)
- type parameter name
  - type parameter.full name (same implementation)
    type parameter.name (same implementation)
  - type parameter.is array (base)
    type parameter.is type parameter (return not is array)

Understanding the overlapping or superfluous properties helps us to better interpret the data analysis results and avoid misinterpretations.

## 2.2 Results

**2.2.1 RQ1: Overall Prevalence of Code Element Types in Suggestions and Selected Suggestions.** Tab. 1 shows the overall counts of the number of suggestions of that are of the same element type as the applied suggestion of an applied code completion. In total, 775 654 suggestions of named elements were examined, in which only around half (392 863 suggestions) were of the same element type as the applied suggestion. This is already a strong indication that code completion by name is not sufficient to include only the desired code element with nearly half of the suggestions were of unwanted element types. The code element types are taken from valid defined types of the data-collecting tool *FeedBaG++* used to create the dataset. Among the code element types, method names, type names, property names, and field names are the most common ones at 107 819, 67 396, 66 373, and 19 211 respectively. Since the dataset records the completion events of *C#*, a language that is heavily object-oriented, it is reasonable for the code completion system to show more of these types of code elements. Interestingly, local variable names and parameter names are not as common, with only 2966 and 2899 matching suggestions. For other code element types, the counts are significantly lower, with some of them without any matching suggestions at all.

In addition to the suggestion counts, Tab. 1 also shows the number of selected suggestions that are of the same element type as the applied suggestion. By observing the selected suggestion counts, what the user was looking for, can be understood better. Out of all suggestions of named elements, only 63 514 suggestions were selected, amounting to only 8.4%. Surprisingly, 58 454 suggestions of the same element type as the applied suggestion were selected, which is 92% of the selected suggestions. This hints at the fact that the users know what they are looking for and further reinforces the previous observation the code completion system is insufficient at limiting the suggestions to only the desired code elements. The selected suggestion count for each code element type is generally much lower than all suggestion counts, with similar ordering. A notable exception is that suggestion for property names was selected more than any other code element types, followed by method names, type names, and field names, in that order. Another interesting observation is that over 80% (2267 out of 2899) of suggestions for parameter names were selected.

**2.2.2 RQ2: Overall Prevalence of Code Elements' Properties and Useful Control Factors.** Fig. 2 illustrates the prevalence of different code elements' properties among completions along with their computed ranks[1]. The size of each point indicates the number of applied completions that consist of a suggestion with the property of a specific code element that matches the applied suggestion. To make sure all the points are visible, the size is capped in the lower end. The colour of each dot corresponds to its rank as shown in the colour scale at the bottom of the figure.

As can be observed from Fig. 2, potentially useful properties are represented by larger and greener points. The larger the point, the more often that property appears in a suggestion in the applied code completions. The greener the point, the higher its rank is, i.e., the more likely that property can be used as a control factor. The most prominent property is, unsurprisingly, the name (and full name for some code elements). This aligned with our expectations and also is the norm of completion by name. Apart from that, some other useful properties include: value type (or return type for methods), declaring type, and parameter-related properties (e.g., parameter name, parameter count, parameter type).

[1]Raw data is available at https://slebok.github.io/cococola [15]

**Table 1.** Correct code element counts
Each count represents the number of top suggestions that are of the same element type as the applied suggestion of an applied code completion. In addition, the counts of only selected suggestions of applied code completions are included.

| Code element type | Suggestion count of the same element type | Selected suggestion count of the same element type |
|---|---|---|
| event | 294 | 44 |
| method | 107 819 | 14 092 |
| field | 19 211 | 5932 |
| property | 66 373 | 17 339 |
| local variable | 2966 | 1757 |
| parameter | 2899 | 2267 |
| alias | 1 | 1 |
| lambda | 0 | 0 |
| assembly | 0 | 0 |
| namespace | 1188 | 362 |
| array type | 0 | 0 |
| delegate type | 143 | 45 |
| predefined type | 123 | 51 |
| type | 67 396 | 5717 |
| type parameter | 168 | 148 |
| unknown | 16 | 0 |
| other | 124 282 | 10 699 |
| element type | 392 863 | 58 454 |
| suggestion with name | 755 654 | 63 514 |

There are also a few more large light green points that seem to show quite some potential, but further investigation showed that they are, in fact, trivial. For instance, the property static modifier is one of these cases. Code completion systems in modern IDEs are smart enough to only offer suggestions of static code elements when the directly preceding element is a class. This means that there is little to no chance that a static code element may appear in a non-class context.

We intentionally leave it undecided what the cut-off point should be for a property to be considered as a control factor, because this choice is highly dependent on the context of the code completion and the intention of the user. Fig. 2 provides an overview for how potential the different properties are, but the final decision should be made by the user.

## 3 Code Completion Control Language

### 3.1 RQ3: Design Criteria for a Code Completion Control Language

The other objective of this paper is to propose a simple language, called *CoCoCoLa*, enabling specification of completion control factors during a code completion. Rather than trying to unambiguously refer to a specific code element, *CoCoCoLa* aims to help narrow down possible valid completion suggestions by allowing extra controls to be added on top of the standard code-completion-by-name flow. For the design of our language to be successful, we formulate the following requirements:

- *Minimal syntax* — The syntax should be a natural extension to the normal completion flow with minimal added effort to add the control factors. When performing a code completion, the users are already taken away from their coding flow. Having a simple and easy to memorise syntax helps reduce friction and context switching during completions.
- *Quick syntax* — This goes hand in hand with the minimal syntax criterion above to provide a more seamless completion experience. While the previous requirement focuses more on the ease of use, quick syntax emphasises the speed of writing in this language.

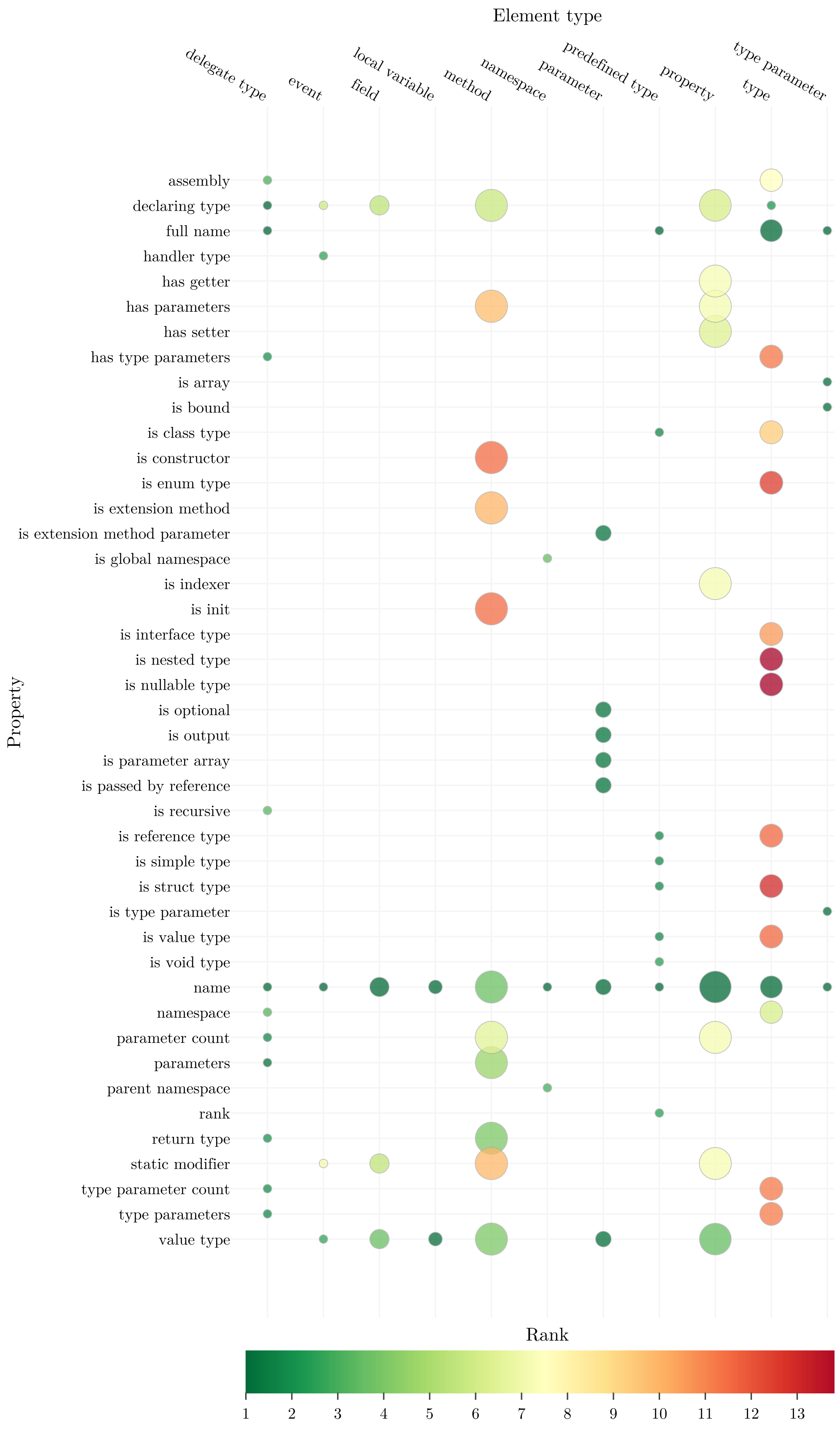


**Figure 2.** Correct code element property counts

- *Granular control* — Although the syntax should be quick and simple, it still has to provide sufficient granular control over specific code element properties to be useful as a control language for code completion.
- *Extensible control* — The design of the syntax should not hinder its extensibility to include more control factors later on. This ensures the language can evolve and adapt to new findings without introducing significant revisions and breaking changes that require the users to relearn.
- *Familiar code completion flow* — The built-in native code completion systems are the most commonly used code completion tool [27]$^{\text{(Finding 1)}}$. To prevent distancing the developers from their familiar code completion experience, the language design should maintain the standard flow as much as possible, without altering and/or breaking it. In addition, utilising the built-in native code completion systems whenever possible reduces work for tool developers. The language design, therefore, should be interoperable with the existing code completion system of the implementation platform as much as possible without significant adaptation required. By doing these, it further helps increase user learnability and acceptance by providing a familiar completion experience.

### 3.2 RQ4: CoCoCoLa — A Code Completion Control Language

*CoCoCoLa* was created as a domain-specific language (DSL) as it aims to solve a specific control problem in the code completion domain. There are two main approaches we have considered to design a code completion control language: (1) to have a syntax similar to the host language, i.e., the language of the code the user's trying to complete; or (2) to have an independent language. Each approach has its own merits and drawbacks [23].

Having the syntax similar to the host language, allows users to write completion prompts with the controls in a natural and familiar way. In this approach, the completion control language can be considered as an extension of the host language, where the completion prompt is written in a way that is similar to the syntax of the host language. Depending on how complex and/or verbose the host language is, while this can make it easier for users to learn, this can be more cumbersome to write at the same time. The expressiveness of this language is also limited by what the host language allows. For each new host language support, there must be a new version of the control language tailored to it, making it heavily language dependent. This can also lead to a lack of consistency across different host languages, making it harder for users to switch between them. In general, this approach cannot always ensure a quick and menial syntax, while maintaining granular and extensible controls.

The latter approach, on the other hand, requires the users to learn a new syntax in order to control code completion. An independent language can be designed to be as expressive as needed, allowing for more intuitive experience. While this can be a barrier for some users, it also allows for a more flexible and consistent design across different coding contexts, as the same syntax can be used regardless of the underlying language.

Ultimately, this approach was chosen to design *CoCoCoLa*, as it allows for more flexibility and a more seamless integration with the existing code completion system. With a unified syntax, more focus can be put on the control factors themselves, rather than having to worry about adapting the syntax for each host language.

In *CoCoCoLa*, the input completion prompt is parsed as two parts: the name part and the control part. The name part is what has always served for code completion by name, i.e., the input string is matched against the name of the available code elements. What is new here is the control part that, as the name suggests, allows control over the completion factors. These two parts are separated by the last double or triple underscores, e.g.:

- `name_part__still_name_part__wildcardControl1_wildcardControl2`
- `name_part___still__name_part___cType1_cVal11_cVal12__wildcardControl__cType2_cVal21`

We have ensured that this underscore syntax is compatible with each of the 50+ most popular software languages in 2023–2024 [10, 26]: *JavaScript, HTML/CSS, Python, SQL, TypeScript, Bash/Shell, Java, C#, C++, C, PHP, PowerShell, Go, Rust, Kotlin, Lua, Dart, Assembly, Ruby, GraphQL, Swift, R, Visual Basic, MATLAB, VBA, Groovy, Scala, Perl, GDScript, Objective-C, Elixir, Haskell, Delphi, MicroPython, Lisp, Clojure, Julia, Zig, Fortran, Solidity, Ada, Erlang, F#, Apex, Prolog, OCaml, COBOL, Crystal, Nim, Zephyr.*

In case no name is provided, the syntax can also start with underscores, which is also supported by most of the above languages (except for *R*, *Nim*, *Ada*, *Fortran*, and *Apex*). In such cases, the completion will be done using only the specified control factors.

As a remark, the syntax of *CoCoCoLa* does not put any specific restrictions to the name part and control values, as long as they are valid identifiers in the host language. This is to ensure that the syntax of *CoCoCoLa* does not interfere or hinder the existing code completion system. *CoCoCoLa* prompts will simply be processed based on the underscore syntax.

By having the syntax of *CoCoCoLa* conform to the valid syntax of an identifier in the host language, we ensure that writing *CoCoCoLa* in the middle of a code completion session does not cause any disruptions or errors to the underlying code completion system. This allows full utilisation of the existing built-in native code completion system, which benefits not only the implementer, but also the users as the familiar completion experience will be offered. It is worth noting that the formulation of such a syntax does not always have to be valid as an identifier. What matters is that the IDE can still process it correctly within a code completion action and not terminate the event, e.g., when typing `.` to initiate another code completion.

Depending on the number of separating underscores, *CoCoCoLa* can be used in to modes:

- *Quick mode with two underscores (`__`):* In this mode, completion control factors can be specified without explicitly mentioning the control type, i.e., they will be treated as a wildcard control type. A completion suggestion is accepted if any of the control matchers deems it matched certain factors. Multiple control factors can be specified at once by separating them using a single underscore (`_`).
- *Precise mode with three or more underscores (`___`):* Although wildcard control can still be used, this mode allows control types to be explicitly stated by specifying a control factor as a pair of control types and control values separated by a single underscore (`_`). A control factor may have as many control values as needed. In this mode, control factors are separated using two underscores (`__`), instead of one in the quick mode, to maintain structural hierarchy.

Each explicit control factor is defined as an underscore-separated pair of control type and control value. The control type indicates what control factor is being specified, while the control value indicates the value of that control factor:

- `en_ElementName` — This control factor specifies the name of the code element, where `ElementName` is the name of the targeting code element. Although an element's name can be specified in the name part, this control factor allows for more flexibility in (re-)specifying its name in the control part.
- `et_ElementTypeName` — This control factor specifies the type of the code element, where `ElementTypeName` can be field, parameters/arguments, variables, methods/functions, etc.
- `vt_ValueTypeName` — This control factor specifies the value type of the code element, where `ValueTypeName` is the name of the targeting type. This is only applicable to code elements that hold a value. For fields, variables, parameters, and literals, this is the (resolved) type of the value they hold. For methods/functions and method/function calls, this is the (resolved) return type of the method/function. For classes, this is the class itself as an object type.
- `rt_ReturnTypeName` — This control factor specifies the return type of the code element, typically a method or function, where `ReturnTypeName` is the name of the targeting type. This behaves exactly the same as `vt_ValueTypeName`.
- `dt_DeclaringTypeName` — This control factor specifies the declaring type of the code element, where `DeclaringTypeName` is the name of the targeting type. As opposed to the value type, the declaring type is the type where the code element is declared. This control factor can be used to restrict the suggestions to only those that are declared in a specific type in the type hierarchy.
- `pc_ParameterCount` — This control factor specifies the number of parameters of the code element, where `ParameterCount` is a non-negative integer. This is only applicable to code elements that have parameters, e.g., methods/functions, constructors, and function calls.
- `pn_ParameterName` — This control factor specifies the name of the parameter of the code element, where `ParameterName` is the name of the parameter. This is only applicable to code elements that have parameters, e.g., methods/functions, constructors, and function calls.
- `pt_ParameterTypeName` — This control factor specifies the type of the parameter of the code element, where `ParameterTypeName` is the name of the targeting type. This is only applicable to code elements that have parameters, e.g., methods/functions, constructors, and function calls.

In general, the control types are typically are two characters long to allow for quick typing. Writing a *CoCoCoLa* prompt is as simple as joining desirable control factors with double underscores (`__`) in the control part. In any case, if multiple controls of the same control type are present, the later control will override the former one, i.e., only the last of that control type will be considered for matching. This is designed so that the user can continue typing to refine and try out different controls without having to manually navigate backward to modify specified controls.

Let us take the completion prompt `foo__et_method__et_field__vt_int__en_bar` as an example. This prompt has the same results as `bar__et_field__vt_int` since the former `et_method` control is overridden by the later `et_field` control of the same control type and the `en_bar` control overrides `foo` in the name part. The simplified prompt `bar__et_field__vt_int` ultimately restricts the suggestions to only those with names containing `bar`, which are fields and have a value type name containing `int`.

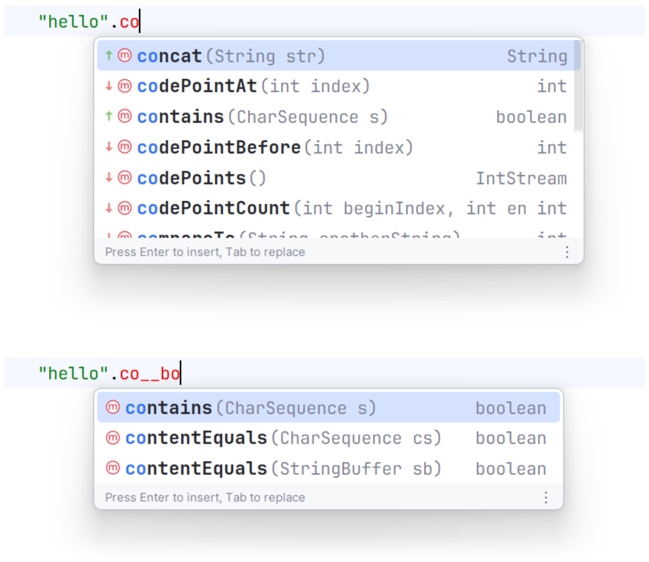
## 3.3 *CoCoCoLa IntelliJ* Plugin

To demonstrate that such an approach to code completion control is feasible, we built a prototype[2] as a plugin for the *IntelliJ Platform*, which provides support to any *JetBrains'* IDE (including *IntelliJ IDEA*). This prototype uses the *JetBrains' Program Structure Interface (PSI)* to handle the user's code.

[2]Available at https://slebok.github.io/cococola

*PSI* is an abstract API layer for any language supported by the *IntelliJ Platform*. Therefore, this plugin, in theory, should support a wide range of languages. However, the current implementation was only checked against *Java* programs, as it is the main language supported by *IntelliJ IDEA*, the IDE used to develop the prototype.

This implementation currently supports the following control factors: element name, code element type, value type, declaring type, parameter count, parameter name, and parameter type. This control factor selection is directly influenced by the results obtained from the above data analysis. Fig. 3 shows some screenshots of this prototype in *IntelliJ*.



```
"hello".co
↑ⓜ concat(String str)                          String
↓ⓜ codePointAt(int index)                      int
↑ⓜ contains(CharSequence s)                    boolean
↓ⓜ codePointBefore(int index)                  int
↓ⓜ codePoints()                                IntStream
↓ⓜ codePointCount(int beginIndex, int en int
Press Enter to insert, Tab to replace

"hello".co__bo
ⓜ contains(CharSequence s)             boolean
ⓜ contentEquals(CharSequence cs)       boolean
ⓜ contentEquals(StringBuffer sb)       boolean
Press Enter to insert, Tab to replace
```

**Figure 3.** Code completion in *Java* in *IntelliJ* without (top) and with (bottom) *CoCoCoLa*

The implementation of *CoCoCoLa* works by intercepting the code completion event and parsing the input completion prompt to extract the name part and the control part. The name part is then passed to the built-in code completion system to get the initial set of suggestions. These suggestions should be the same as the suggestions that would be obtained without *CoCoCoLa*. From the control part, control factors are extracted and simplified into a set of unique filters. Our plugin is designed to be extensible, allowing for new control factors to be added in the future as needed. Since control factors of the same type are allowed to be specified multiple times, only the last one of each type is kept in the final set of filters. The filtering process is done by iterating through the suggestions and checking each suggestion against the filters in the order they were specified in the given completion prompt. The filtered suggestions are then displayed to the user as the final completion suggestions as shown in Fig. 3. This allows users to specify additional constraints on the code completion process without having to manually navigate through all the available options. This plugin acts as a proof-of-concept for the integration of completion control factors into the standard code completion process, and, specifically, the *CoCoCoLa* language in a real-world scenario.

## 3.4 Application Scenarios

Let us take a look at some scenarios of how *CoCoCoLa* can be used to control code completion. In different development context, developers may choose or are required to follow certain naming conventions. For instance, a Boolean value is often prefixed with `is` or `has`, e.g., `isAccessible = true` or `hasAccess = true`. In another convention, `access = true` is also acceptable as a more concise way of naming a Boolean value. These conventions help to keep the codebase consistent and maintain readability within the working context. However, if frequent context switching is required, developers may also want to keep track of which naming convention to use, increasing cognitive load unnecessarily. To anticipate for these conventions (when using completion by name), then completion prompt can be relaxed to only include the most differentiating part of the name, i.e., as `access`. This will most likely include the desired element, but also other suggestions for unrelated elements, e.g., `accessibleItems: List[Item]` or `grantAccess(): void` that are not Boolean values nor return Boolean values. With *CoCoCoLa*, the suggestions can be restricted to only include Boolean values with the completion prompt `access__bool`, `access__boolean`, or more precisely, `access___t_boolean`. Such a behaviour cannot be achieved with completion by name alone.

Consider another scenario where a *Python* developer needs to perform some string-manipulating tasks in *Java*. To get a substring of a string, *Python* provides a convenient "slicing" syntax `foo[start:end]` which returns all characters in the string `foo` from `start` up until `end` exclusively. When switching to *Java*, the *Python* developer could not find a `foo.slice(start, end)` function and the standard code completion by name cannot help much here either. They then have to rely on personal experience, conventions, and common sense to make and check each guess. Through a series of trials and errors, and a bit of luck, they finally found a function that they know it has to be there — `foo.substring(start, end)`. Knowing that such a function (1) is callable from a string, (2) accepts two arguments as its parameters, and (3) returns a string, this struggle to find a function with (4) an unknown name can be avoided by using *CoCoCoLa* with the completion prompt `foo.___pc_2__t_string` on a string `foo`. This completion prompt leaves the name part (before the `___`) empty to allow for any possible names, addressing (4); the `pc_2` control indicates that the desired element must have a parameter count of 2, addressing (2); and the `t_string` control ensures that only elements of type string or return a string are accepted, addressing (3). An extra control `dt_string` can also be added to ensure it has the declaring type string, i.e., can be called from a string, to address (1). Since this completion prompt is called from a string, such control is redundant in this case.

As some smaller, yet no less useful, scenarios where code completion controls can be applied: (1) *CoCoCoLa* can help exploring available APIs without having to jump to the (decompiled) source code file where they are declared. By staying at the current code position, users can reduce context switching and prevent losing their current context. (2) A completion control can be defined to target a specific code completion type (e.g., element, keyword, template, etc.) to avoid confusions after applied the selected suggestion.

These clearly illustrate that code completion (by name) lacks the ability to fully express desired properties that the user might want in the resulting suggestions. *CoCoCoLa* along with the concept of completion control factors were created to fill in this gap to aid and accelerate coding productivity.

## 4 Threats to Validity

*Internal Validity* — Internal validity relates to the correctness of the results obtained from the presented analysis. For this project, the internal validity mainly concerns how the rank is calculated and the threshold criteria to deem a code element property impactful. Different formulations of these values may lead to different results. Nevertheless, the chosen definitions yield results that are consistent with expectation. In addition, in spite of the *CoCoCoLa* language design closely following the set criteria, it is still subject to personal choices and should be further validated for practical usefulness, which deserves its own user study.

*External Validity* — External validity relates to the generalisability of the results obtained from the presented analysis. For this project, the external validity mainly concerns the generalisability of the results obtained from analysing the *MSR 2018* dataset. Although *MSR 2018* dataset contains 11 million interaction events used for a data mining challenge, all of these events were collected from *C#* programs. *C#* is a good representative for an object-oriented language, but the results may not entirely reflect on other paradigms. Beyond that, data analysis can only be done to available code element types provided by the dataset. In addition, meta-properties of code elements were not considered in the analysis, e.g., the number of parameters of a method. These meta-properties can be extracted from the code elements' properties, but deciding which meta-properties to include in the analysis deserves its own investigation. As a result, the analysis may not fully capture the impact of these meta-properties on code completion. Lastly, the presented analysis does not concern the impact of code completion approaches (indirect code completion, vertical code completion, temporal code completion, etc.) other than direct code completion by name due to the lack of these approaches in the chosen dataset.

## 5 Related Work

Jin and Servant [11] showed that there is a hidden cost of having too many code completion suggestions. This can negatively impact the ranking of desired options, and in turn, worsen the user's completion experience, resulting in unsatisfied and unsuccessful completion attempts. Numerous approaches have been proposed to improve the code completion based on various techniques, such as program history [24], statistical language model [22], contextual information [1–3], Bayesian network [21], and machine learning [4, 9]. However, Wang et al. [27] still shared similar concerns about the suggestion ranking in *Finding 5*. This hints at the need for a simple solution to gain back control over the competition results.

*Fully qualified name (FQN)*, as mentioned before, is a possible option to refer to a code element. Nevertheless, FQN is a concept, not a standardised structure. Although very similar, each software language has its own definition for FQNs, commonly using the *dot syntax*. Moreover, FQNs in object-oriented languages only concern namespaces and class associations. Hence, limited-scoped code elements, such as parameters and local variables, do not have clear FQN representations.

*Enriched event steam (EES)* proposed by Proksch [18] provides a well-defined *naming scheme for code elements* that can unambiguously refer to a wide range of code elements in object-oriented programs. However, this similar problem as using fully qualified names as it tends to be extremely elaborated and detailed that gear towards data analysis, making it impractical to use for code completion.

For all the above approaches, they only concern distinguishing and identifying code elements, mainly using a form of hierarchical structure with some addition metadata. They lack away to express property information of the code elements, which showed to be useful to control the results of a code completion. In addition, using unrestricted grammar prevents them from easily utilising the existing built-in native code completion system, which has been found to be the most used code completion tool by Wang et al. [27] in *Finding 1*.

## 6 Conclusion and Future Work

In this paper, we investigated how frequent properties of different code elements appeared within the suggestion list of a code completion and across multiple completions by analysing the collected code completion events from the *MSR 2018* dataset. Our results showed that, apart from the currently trivial use of completion by name, value/return type, declaring type, and parameter properties (e.g., name, type, and count) have prominent presences, which can be leveraged to improve the completion experience as control factors. While other properties are less significant, together, they serve as an overview on what the users look for when doing code completions.

Having control over the code completion results is increasingly useful as the more complex and lengthy the completion suggestion list becomes. We presented a set of design criteria for a minimal language to allow the user to quickly specify and combine multiple control factors while performing code completions, without deviating from the standard completion flow. *CoCoCoLa* is our proposal of such a language, which syntax is compatible with 50+ most popular software languages in 2023–2024 [10, 26] to harness the power of the existing built-in native code completion systems to provide a more seamless experience.

Preliminary experiments and exposure to the concept of code completion control and *CoCoCoLa* showed promising results and positive reactions. With an implementation of *CoCoCoLa* ready, our next step is to perform empirical studies to evaluate the usability of *CoCoCoLa* in practice and the helpfulness of code completion control, compared to traditional code completion methods.

## References

[1] Shamsa Abid, Hamid Abdul Basit, and Shafay Shamail. 2022. Context-aware Code Recommendation in Intellij IDEA. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 1647–1651. https://doi.org/10.1145/3540250.3558937

[2] Muhammad Asaduzzaman, Chanchal K. Roy, Kevin A. Schneider, and Daqing Hou. 2014. Context-Sensitive Code Completion Tool for Better API Usability. In *Proceedings of the 30th International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 621–624. https://doi.org/10.1109/ICSME.2014.110

[3] Muhammad Asaduzzaman, Chanchal K. Roy, Kevin A. Schneider, and Daqing Hou. 2014. CSCC: Simple, Efficient, Context Sensitive Code Completion. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 71–80. https://doi.org/10.1109/ICSME.2014.29

[4] Matteo Ciniselli, Nathan Cooper, Luca Pascarella, Antonio Mastropaolo, Emad Aghajani, Denys Poshyvanyk, Massimiliano Di Penta, and Gabriele Bavota. 2022. An Empirical Study on the Usage of Transformer Models for Code Completion. *IEEE Transactions on Software Engineering* 48, 12 (2022), 4818–4837. https://doi.org/10.1109/TSE.2021.3128234

[5] Vincent J. Hellendoorn, Sebastian Proksch, Harald C. Gall, and Alberto Bacchelli. 2019. When Code Completion Fails: A Case Study on Real-World Completions. In *Proceedings of the 41st IEEE/ACM International Conference on Software Engineering (ICSE)*. 960–970. https://doi.org/10.1109/ICSE.2019.00101

[6] IntelliJ. 2023. Live templates | IntelliJ IDEA Documentaion. https://www.jetbrains.com/help/idea/using-live-templates.html

[7] IntelliJ. 2023. Postfix code completion | IntelliJ IDEA Documentaion. https://www.jetbrains.com/help/idea/auto-completing-code.html#postfix_completion

[8] IntelliJ. 2024. Narrow down the suggestion list | Code completion | IntelliJ IDEA Documentaion. https://www.jetbrains.com/help/idea/auto-completing-code.html#wildcards

[9] Maliheh Izadi, Jonathan Katzy, Tim Van Dam, Marc Otten, Razvan Mihai Popescu, and Arie van Deursen. 2024. Language Models for Code Completion: A Practical Evaluation. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. ACM, Article 79, 13 pages. https://doi.org/10.1145/3597503.3639138

[10] JetBrains. 2023. Languages — The State of Developer Ecosystem in 2023 Infographic | JetBrains. https://www.jetbrains.com/lp/devecosystem-2023/languages/

[11] Xianhao Jin and Francisco Servant. 2018. The Hidden Cost of Code Completion: Understanding the Impact of the Recommendation-List Length on its Efficiency. In *Proceedings of the 15th International Conference on Mining Software Repositories (MSR)*. ACM, 70–73. https://doi.org/10.1145/3196398.3196474

[12] Yun Young Lee, Sam Harwell, Sarfraz Khurshid, and Darko Marinov. 2013. Temporal Code Completion and Navigation. In *Proceedings of the 35th International Conference on Software Engineering (ICSE)*. IEEE CS, 1181–1184. https://doi.org/10.1109/ICSE.2013.6606673

[13] Microsoft. 2023. IntelliSense in Visual Studio. https://learn.microsoft.com/en-us/visualstudio/ide/using-intellisense

[14] MSR. 2018. MSR 2018 — Mining Challenge. https://2018.msrconf.org/track/msr-2018-Mining-Challenge

[15] Nhat and Vadim Zaytsev. 2025. Data underlying the publication: "CoCoCoLa: Code Completion Control Language". 4TU.ResearchData, https://doi.org/10.4121/24c94759-9f9a-4423-9dcb-893816e470af.v1.

[16] Luiz Laerte Nunes da Silva Jr., Thiago Nazareth de Oliveira, Alexandre Plastino, and Leonardo Gresta Paulino Murta. 2012. Vertical Code Completion: Going Beyond the Current Ctrl+Space. In *Proceedings of the Sixth Brazilian Symposium on Software Components (SBCARS): Architectures and Reuse*. IEEE CS, 81–90. https://doi.org/10.1109/SBCARS.2012.22

[17] James L. Peterson. 1980. Computer Programs for Detecting and Correcting Spelling Errors. *Communications of the ACM* 23, 12 (12 1980), 676–687. https://doi.org/10.1145/359038.359041

[18] Sebastian Proksch. 2017. *Enriched Event Streams: A General Platform For Empirical Studies On In-IDE Activities Of Software Developers*. Ph. D. Dissertation. Technische Universität Darmstadt, Darmstadt. http://tuprints.ulb.tu-darmstadt.de/6971/

[19] Sebastian Proksch. 2020. KaVE Project — Datasets. https://www.kave.cc/datasets

[20] Sebastian Proksch. 2020. KaVE Project — FeedBaG++. https://www.kave.cc/feedbag

[21] Sebastian Proksch, Johannes Lerch, and Mira Mezini. 2015. Intelligent Code Completion with Bayesian Networks. *ACM Transactions on Software Engineering Methodology* 25, 1, Article 3 (12 2015), 31 pages. https://doi.org/10.1145/2744200

[22] Veselin Raychev, Martin Vechev, and Eran Yahav. 2014. Code Completion with Statistical Language Models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 419–428. https://doi.org/10.1145/2594291.2594321

[23] Lukas Renggli. 2010. Dynamic Language Embedding With Homogeneous Tool Support. https://doi.org/10.7892/boris.104713

[24] Romain Robbes and Michele Lanza. 2010. Improving Code Completion with Program History. *Automated Software Engineering* 17 (06 2010), 181–212. https://doi.org/10.1007/s10515-010-0064-x

[25] Barry Schwartz. 2005. *The Paradox of Choice: Why More Is Less*. Harper Perennial.

[26] Stack Overflow. 2024. Languages | Technology | 2024 Stack Overflow Developer Survey. https://survey.stackoverflow.co/2024/technology#most-popular-technologies-language

[27] Chaozheng Wang, Junhao Hu, Cuiyun Gao, Yu Jin, Tao Xie, Hailiang Huang, Zhenyu Lei, and Yuetang Deng. 2023. How Practitioners Expect Code Completion?. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 1294–1306. https://doi.org/10.1145/3611643.3616280

Received 2025-04-10; accepted 2025-05-19