Source Code Recommender Systems: The Practitioners' Perspective

Matteo Ciniselli*, Luca Pascarella*, Emad Aghajani*, Simone Scalabrino†, Rocco Oliveto†, Gabriele Bavota*
*SEART @ Software Institute, Università della Svizzera italiana (USI), Switzerland
†STAKE Lab @ University of Molise, Italy

***Abstract*—The automatic generation of source code is one of the long-lasting dreams in software engineering research. Several techniques have been proposed to speed up the writing of new code. For example, code completion techniques can recommend to developers the next few tokens they are likely to type, while retrieval-based approaches can suggest code snippets relevant for the task at hand. Also, deep learning has been used to automatically generate code statements starting from a natural language description. While research in this field is very active, there is no study investigating what the users of code recommender systems (*i.e.,* software practitioners) actually need from these tools. We present a study involving 80 software developers to investigate the characteristics of code recommender systems they consider important. The output of our study is a taxonomy of 70 "requirements" that should be considered when designing code recommender systems. For example, developers would like the recommended code to use the same coding style of the code under development. Also, code recommenders being "aware" of the developers' knowledge (*e.g.,* what are the framework/libraries they already used in the past) and able to customize the recommendations based on this knowledge would be appreciated by practitioners. The taxonomy output of our study points to a wide set of future research directions for code recommenders.**

***Index Terms*—Code Recommender Systems, Empirical Study, Practitioners' Survey**

I. Introduction

Recommender systems are becoming more and more popular in software engineering. These tools can support developers in several tasks [44], such as documentation writing and retrieval [19], [33], [34], [59], code refactoring [9], [50], bug fixing [26], [27], [52], bug triaging [49], [58], code review [53], [54] etc. Among these, *source code recommender systems* support developers in writing code.

Code recommender systems have been designed using different underlying techniques. For example, retrieval-based approaches [31], [57] identify relevant code elements to reuse given the code under development in the Integrated Development Environment (IDE) and/or a query describing the coding task at hand. Other approaches exploit deep learning (DL) to (i) automatically generate a needed code (or parts of it) given its textual description [37], [60] or (ii) predict the tokens the developer is likely to type given the code in the IDE [28], [55]. Building on top of this literature, GitHub recently presented Copilot [2], a tool using DL to recommend entire code statements or even whole functions.

While research on code recommender systems is extremely active, no previous work investigated what the *desiderata* of software developers are. In other words, the techniques proposed in the literature are mostly based on assumptions made by researchers. For example, a recent work by Wen *et al.* [57] targets the automatic implementation of whole code functions. However, it is unclear whether developers are actually looking for such a type of support or if, instead, they prefer the classic code completion implemented in IDEs. Also, none of the existing tools and techniques customize their code recommendations based on the developer's coding style and/or to their expertise and it is unclear whether such a functionality would be important for practitioners. To answer these questions, we present a study involving 80 practitioners which is aimed at investigating the characteristics of code recommenders that they consider important. In particular, after collecting demographics information, we asked participants their opinion about the code recommenders they use (*e.g.,* copilot, default code completion in IDE, etc.) from three perspectives: (i) *coverage* (*i.e.,* in how many coding scenarios the tool can provide recommendations), (ii) *accuracy* (*i.e.,* to what extent recommendations are close to what practitioners need), and (iii) *usability* (*i.e.,* how friendly the user interface is). For each of these three aspects participants were asked to describe improvements (if any) they would like to see in the recommender they mostly use. Then, we ask them in an open-ended question what the characteristics of code recommenders they consider important are.

Each answer we received has been independently analyzed by five authors through an open-coding inspired approach with the goal of assigning a set of tags to it. The extracted tags represent requirements expressed by practitioners for code recommenders and, after conflict resolution, have been organized in a hierarchical taxonomy. Such a process has been performed iteratively four times, with 20 practitioners taking part in each iteration. We stopped once the output taxonomy converged (*i.e.,* no additional requirements were added to the taxonomy from the answers we received in the last iteration).

The output of our study is a taxonomy of 70 "requirements" that should be considered when designing code recommender systems (Fig. 2). For example, our taxonomy highlights that practitioners are interested in *adaptive* recommendations, meaning that the recommended code should be automatically adapted to the code under development (*e.g.,* reusing identifiers when possible) and to the developer's coding style.

Significance of research contribution. The taxonomy of 70 "requirements" for code recommenders output of our study provides a rich research roadmap in the field of code recommender systems. Indeed, as our empirical evidence shows, the *desiderata* of practitioners in this context are not always aligned with what offered by state-of-the-art techniques.

Based on our taxonomy, researchers can have a clear understanding of what the priorities are when designing code recommenders.

Data availability. We release the survey used in the study, the collected answers with the results of the manual analysis we performed on them, plus additional material in our replication package: https://code-recommenders.github.io.

II. Study Design

The *goal* of the study is to investigate what the *desiderata* of software practitioners are when it comes to code recommender systems. The *context* consists of *objects*, *i.e.,* a survey designed to investigate the study goal, and *subjects* (referred to as "participants"), *i.e.,* 80 practitioners recruited through Amazon Mechanical Turk (MTurk) [1] and personal contacts.

We aim at answering the following research question:

> *What are the characteristics of code recommender systems that are considered important by practitioners?* Despite the many code recommender systems proposed in the literature, no study investigated what practitioners actually need in terms of automatic support during coding activities. Answering our RQ can guide the development of better code recommender systems.

A. Context Selection — Participants

We recruited participants through two channels. First, we used MTurk [1], a crowdsourcing website to hire people for on-demand tasks. We enrolled participants that (i) have successfully completed in the past at least 50 tasks on MTurk; (ii) have an approval rate for their past tasks grater than 90% (*i.e.,* more than 90% of the tasks they performed in the past have been approved by the requester); (iii) hold the MTurk Master qualification assigned to "top workers". We only involved practitioners in our survey, excluding students (at any level) and researchers.

Participants who completed our survey were payed 10$ upon a manual verification in which we made sure that the provided open answers (described in the following) were meaningful and written in correct English. We collected 31 complete surveys from MTurk. In addition, we invited practitioners in the authors' contact network. This resulted in additional 49 answers, leading to a total of 80 participants. As explained later, developers were not invited all together, but in four rounds of 20 participants each. Demographics about participants are presented in Section III.

B. Context Selection — Survey

Our survey has been implemented in Qualtrics [3] and is available in our replication package [4].

Participants were initially presented with a welcome page, which explained the goal of the study, reported its expected duration (∼15 minutes), and set the context by explaining what source code recommender systems are:

> *With source code recommender systems we refer to approaches that can be used to automatically suggest code to developers while they are writing code. The classic example in this context is the code completion feature implemented in IDEs.*
> *However, some tools go beyond the classic code completion task and recommend longer pieces of code to developers to autocomplete a task they perform (see,* e.g., *https://copilot.github.com/).*

By agreeing to participate, they started our survey composed of three steps.

Step ①: Demographic Information. We asked participants to indicate (i) their job position (*e.g.,* developer, tester), (ii) the programming language and the IDE they mostly use, and (iii) the number of years of programming experience.

Step ②: Experience with Code Recommenders. The second step included questions about the participants' experience with code recommender systems. We asked what tool(s) they use as code recommender, with the possibility of selecting "*The default one in the IDE*" and/or specifying the tool(s) in an open text box. If participants indicated that they did not use any code recommender, the survey stopped. We also collected the frequency with which participants check code recommendations: Occasionally (*i.e.,* less than 50% of times a recommendation is available), Most of times (*i.e.,* more than 50%, but not always), and Always.

Then, we asked to rate the code recommendation capabilities of the tool they mostly use from three perspectives: (i) *coverage* (*i.e.,* in how many coding scenarios the tool can provide recommendations), (ii) *accuracy* (*i.e.,* to what extent recommendations are close to what practitioners need), and (iii) *usability* (*i.e.,* how friendly the user interface is). For each of these three aspects, participants could indicate their answer on a three-point scale (Low, Medium, High), or select a "Not sure" option. We provided participants with detailed explanations about the three perspectives [4]. For each of them, we also asked the improvements participants would like to see in the code recommender(s) they commonly use (*e.g.,* what they would like to have in terms of "coverage" that is not currently supported). Finally, we ask a specific question related to the accuracy of the recommendations: *Assume that a code recommendation tool provides a list of recommendations sorted by likelihood of being relevant (*i.e., *the most relevant on top). How many of these recommendations would you be willing to read to find the right one?* Answers to this question can inform the evaluation of code recommenders by researchers. Indeed, when evaluating approaches for automatically generating code, researchers often assess their performance for the top-k recommendations (*e.g.,* top-50 in [52]). Knowing how many recommendations developers are willing to inspect can help in setting the number of generated solutions to realistic values.

TABLE I
CHARACTERISTICS OF CODE RECOMMENDATIONS COLLECTED FROM LITERATURE

Attribute	Description	References
Concise Code	The recommended code must be as short and simple as possible.	[18], [23], [36]
Correct Code	The recommended code must be bug-free.	[23]
Familiar	If multiple recommendations are possible, the one using code that is more familiar to the developer must be used (*e.g.,* the code using APIs already used in the past by the developer receiving the recommendation).	[30]
High readability	The recommended code must be readable (*e.g.,* avoid very long statements, adopt indentation).	[33]
High reusability	The recommended code must be easy to reuse (*e.g.,* a code using object types not available in the language but defined in other projects from which the recommended code has been learned is difficult to reuse).	[33]
Inline Documentation	The recommended code features comments explaining the code step-by-step.	[36]
Meets coding layout	The recommended code must be adapted to the context of the recommendation by adopting the same coding layout (*e.g.,* same indentation, spaces between code tokens).	[25]
Meets naming conventions	The recommended code must be adapted to the context of the recommendation by adopting the same naming conventions (*e.g.,* if variables are named with camelCase, the same convention must be adopted in the recommended code).	[25]
Precise Typing Information	A recommended code using AVerySpecificType should be preferred over a recommended code using Object.	[38]
Responsive	The responsiveness of the code recommendation system in terms of time needed to generate a recommendation.	[47]
Same name	The recommended code must be adapted to the context of the recommendation by using the same variable names of the code it completes when possible.	[38]
Syntactical Correctness	The recommended must not introduce syntax errors.	[6], [47], [55]
Step-by-step Solution	In case the recommended code spans across many statements, the code is divided into multiple chunks (by using a blank line), each one responsible for a sub-task.	[36]
Vulnerability-free	The recommended code must be vulnerability-free.	[46]

Step ③: Characteristics of Code Recommenders. The third and last step is the core of our survey, in which we asked participants the characteristics of code recommendations they consider important. In the first question, participants could describe in an open text box the characteristics of code recommendations they perceive as most important, accompanying each one with a short explanation/rationale. We clarified that they could include both functional and non-functional characteristics.

In the second question, the survey showed a list of 14 characteristics we preliminarily defined, asking participants to select the ones they consider important (if any). To determine such 14 categories, we manually analyzed the state of the art related to code recommender systems. Specifically, we focused on papers that presented: (i) techniques for code generation/completion (see *e.g.,* [47]), (ii) empirical studies about code generation/completion techniques (*e.g.,* [46]), (iii) techniques to generate code examples (*e.g.,* [33]), and (iv) empirical studies about code examples used by developers (*e.g.,* [36]). For example, we extracted the *high reusability* characteristic from the paper by Moreno *et al.* [33].

The full list of characteristics with the corresponding references they were extracted from is available in Table I. Note that, to define such a list, we did not perform a systematic literature review to identify **all** papers in the surveyed areas: We relied on the experience of the six authors to identify a set of 41 peer-reviewed papers published in international conferences and journals and applied backward snowballing on their references to identify additional relevant works.

At the end, we inspected 53 papers. Each paper was assigned to one author in charge of adding to a spreadsheet the list of "characteristics" described in the paper (if any). In some cases, the papers from which we extracted a characteristic did not explicitly point to the need for considering such a characteristic when building code recommender systems.

However, this could be inferred from the text of the paper. One example is the work by Schuster *et al.* [46]. The authors show that "*neural code autocompleters are vulnerable to poisoning attack*" [46]. Thus, we infer that *Vulnerability-free* is one of the characteristics to assess for code recommenders. Similarly, Nasehi *et al.* [36] studied what makes a good code example on Stack Overflow. Again, this is something not directly linked to a code recommender. However, we assume that characteristics of good code examples could be relevant for recommended code as well.

TABLE II
ROUNDS OF DATA COLLECTION (20 PARTICIPANTS EACH)

Round	New L1 Charac.	New L2 Charac.	New L3 Charac.	New L4 Charac.	Conflicts
I	5	25	13	0	24%
II	0	9	6	6	21%
III	0	0	6	0	24%
IV	0	0	0	0	24%

C. Data Collection and Analysis

Given the goal of our study, we decided to run it in multiple rounds until we reached saturation in the collected taxonomy of characteristics. First, we invited 20 practitioners to complete our survey (first round). Then, to answer our RQ, we analyzed the data collected in steps ② and ③ in the open answers by following an open-coding inspired approach: Five of the authors independently assigned a set of tags to each of the open answers provided by the 20 participants that described through written text the improvements they would like to see in terms of coverage, accuracy, and usability of the code recommenders they use and the characteristics of code recommenders they perceive as most important. Each tag was meant to encode a specific characteristic (*e.g., Early prediction* was derived from the answer "*if the completions are not timely, it is just easier to type the whole thing myself sometimes*").

Conflicts (*i.e.,* different tags assigned by the five authors to the same answer) were solved through online meetings involving all authors. The set of "characteristics" derived through this analysis was then complemented with the ones selected by developers as important from the list extracted from the literature. The output of this analysis is a hierarchical taxonomy of characteristics of code recommenders (Fig. 2), with level-1 nodes indicating root categories, level-2 nodes indicating sub-categories of a specific root category, and so on.

This first round was followed by three additional rounds, each of which added 20 participants. We stopped with this process when the execution of a new round did not result in the addition of any new characteristic in our taxonomy, indicating that a good level of saturation was reached. We are aware that additional rounds may further strengthen the generalizability of our taxonomy. However, we had to balance the comprehensiveness of the taxonomy with the feasibility of the study given our limited access to professional developers.

Table II reports for each round (i) the number of new characteristics added to the different levels of the taxonomy and (ii) the percentage of conflicts arisen during open coding. The number of characteristics reported in Table II for each level does not match the final number in Fig. 2: This happens because we reorganized the taxonomy after each round for readability reasons (*e.g.,* some level-2 characteristics were moved to level-3 as child of a new level-2 category). However, the overall number of characteristics (70) matches the one in our final taxonomy (Fig. 2).

Concerning conflicts, since we had five authors inspecting each answer in an open coding setting in which the codes (*i.e.,* characteristics of code recommenders) to extract were not pre-defined but had to emerge from the data, we had some form of conflict very frequently, *e.g.,* one of the five authors not extracting a characteristic indicated by the remaining four authors. These cases were usually trivial to solve in the online meetings. Other types of conflict required instead longer discussions, and these are the ones we document in Table II.

In particular, we report the percentage of cases in which there was no majority in extracting a "characteristic" from an open answer (*i.e.,* less than three authors reported it). As it can be seen, this happened in ∼20% of cases in each round.

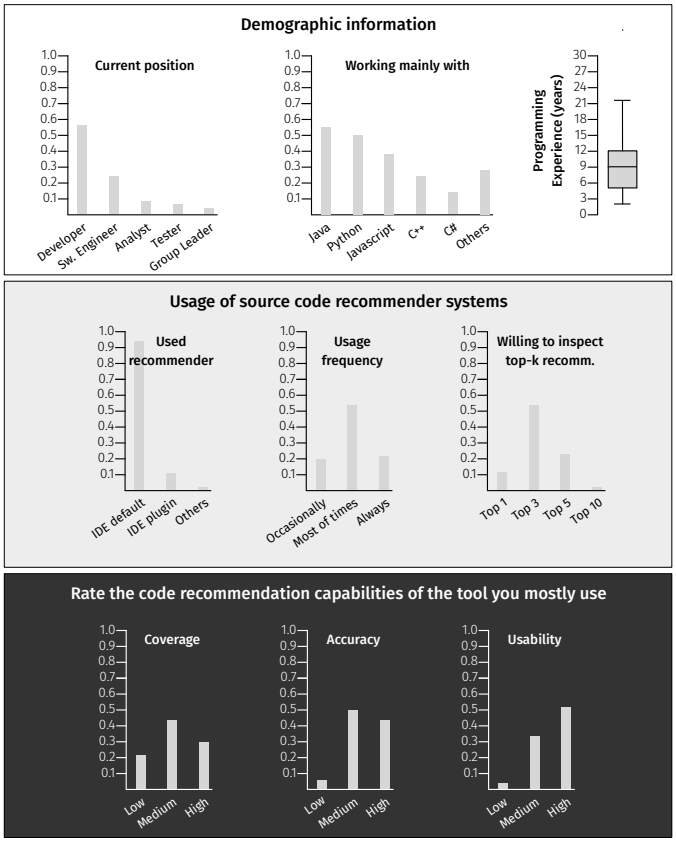

Fig. 1. Demographic information (top), usage of source code recommenders (middle), and assessment of code recommender systems (bottom).

III. Results Discussion

We start by summarizing demographic information about the study participants and their experience with code recommender systems. Then, we discuss the taxonomy of characteristics of code recommender systems, which is the main outcome of this study.

A. Demographics and Experience with Code Recommenders

Fig. 1 presents a visualization of demographic data for the 80 practitioners involved in our study. Most of the participants are developers (47) or software engineers (20), with others classifying themselves as analysts, testers, and group leaders. Participants mostly work with Java, Python and/or Javascript.

In terms of experience, 86% of participants has more than five years of programming experience, with an average of 9.7 years, and a median of 9. The vast majority (85%) of participants use the IDE's built-in code completion feature as their *only* code recommender system.

Among the used IDE plugins, GitHub Copilot is the most represented one, with 5 mentions all coming from the latest round we performed. Indeed, when we run the first three rounds Copilot was not yet publicly available.

Less than 19% of the participants claimed to "occasionally inspect the code recommendations provided", with the remaining ones looking at them *most of times* or *always* when they are available.

Participants indicated the willingness to inspect at most the top-5 code recommendations provided (95%), with the majority (56%) focusing only on the top-3. This suggests researchers to limit the assessment of code recommenders to the top-5 recommended solutions, since others are very unlikely to be considered by developers.

When asked about their assessment of the code recommender they use in terms of coverage, accuracy, and usability, developers reported that they are mostly happy of the tools they use as for these aspects. Specifically, ∼44% of them considered the accuracy high, and ∼50% medium; ∼58% evaluated the usability as high, and ∼38% as medium; and ∼31% judged the coverage high and ∼46% medium. The coverage of the provided support (*i.e.,* the variety of scenarios in which the tool is able to recommend code) is the aspect achieving the lowest rates, with ∼23% of participants reporting a low coverage.

B. Taxonomy Discussion

Fig. 2 reports the taxonomy of characteristics participants indicated as relevant for code recommender systems. The number on the top-right of the level-1 and level-2 categories indicates the number of participants who mentioned such a characteristic as important (out of 80). While some characteristics have only been mentioned by few developers (*e.g., Awareness* → *Developer's task*) we decided to include all of them for completeness.

On top of that, Fig. 3 reports the definition of all categories up to level-3. We do not report the ones for level-4 categories for space reason. However, these usually represent specifications of level-3 categories that are quite intuitive and we include in our replication package [4] complete definitions for all categories, with a search interface that helps in quickly identify the category of interest.

We discuss our taxonomy, going through each of the five root categories depicted in Fig. 2. We use the 💡 icon to highlight lessons learned for future research in the field.

Characteristics of recommended code. This root category groups characteristics of the recommended code that participants consider important. All participants mentioned at least one aspect related to the *quality* of the recommended code. Note that, at a first sight, one may think that code quality is not relevant for code recommenders, since they often recommend a few code tokens to complete a statement the developer is writing. This is, however, not the case for the new generation of code recommenders (see *e.g.,* GitHub Copilot [2] or recently proposed works in the literature [11], [47], [57]) that can recommend complete code statements or even entire functions.

Receiving recommendations having a high readability and understandability is a priority for developers (66 mentions). 39% of the participants indicated their preference for concise recommendations, *e.g.,* "*I do not use completion that suggests entire snippets. The reason is that if I get an entire snippet I'll need to understand it to make sure is what I need and, based on my experience, this is not faster than writing the code myself.*" 💡 This goes somehow in contrast with recent work targeting the automatic implementation of whole functions [2], [57]. However, as we will see later when discussing the *Coverage of Support*, developers are willing to use recommendations for complex scenarios (such as entire functions) if they have a high confidence in the received recommendations. Also, in case of recommendations composed by several statements, 14 participants indicated the importance of organizing these recommendations as *Step-by-step solutions*, meaning that the code is divided into multiple chunks (using a blank line), each one responsible for a sub-task. Such a feature requires the ability of the recommender to identify sub-tasks within the suggested code.

💡 More in general, readability and understandability are two important aspects of code recommendations, *e.g.,* "*If I don't understand what the suggestion is about at a first glance, I ignore it and I continue programming*". However, to the best of our knowledge, no code recommender explicitly focuses on these aspects when deciding which recommendation to trigger. While this could be possible exploiting the readability metrics previously defined in the literature [10], [14], [32], [39], [45], it is still unclear to what extent such metrics work on artificial code.

Other participants pointed at the importance of the recommended code to meet *best practices* and *coding standards*. 💡 This includes the possibility to customize the notion of coding standard (*e.g.,* "*meets coding standards of the company in terms of code quality*", "*pushing good coding standards, either general ones or customized by the user*"). This is another aspect currently unsupported in the state of the art.

Several other aspects of code quality have been mentioned by participants (*e.g.,* ensure good *performance*, *robustness*, and *reusability* of the recommended code). For example, in the case of performance one practitioner wrote: "*Now that more complex recommendations are possible thanks to tools like copilot, aspects related to code quality should be taken into account more, for example by picking among two possible recommendations the one ensuring better performance*".

A crucial quality aspect mentioned by 69 participants is, as expected, the *correctness* of the recommended code. This means that the recommendation must not break the syntax and/or introduce bugs/vulnerabilities in the code under development (*e.g., "the IDE must never recommend a solution that leads to a malfunction*"). While such a finding might look obvious, it triggers a few considerations about state-of-the-art techniques. For example, DL models have been proposed to support code completion [11], [28], [55].

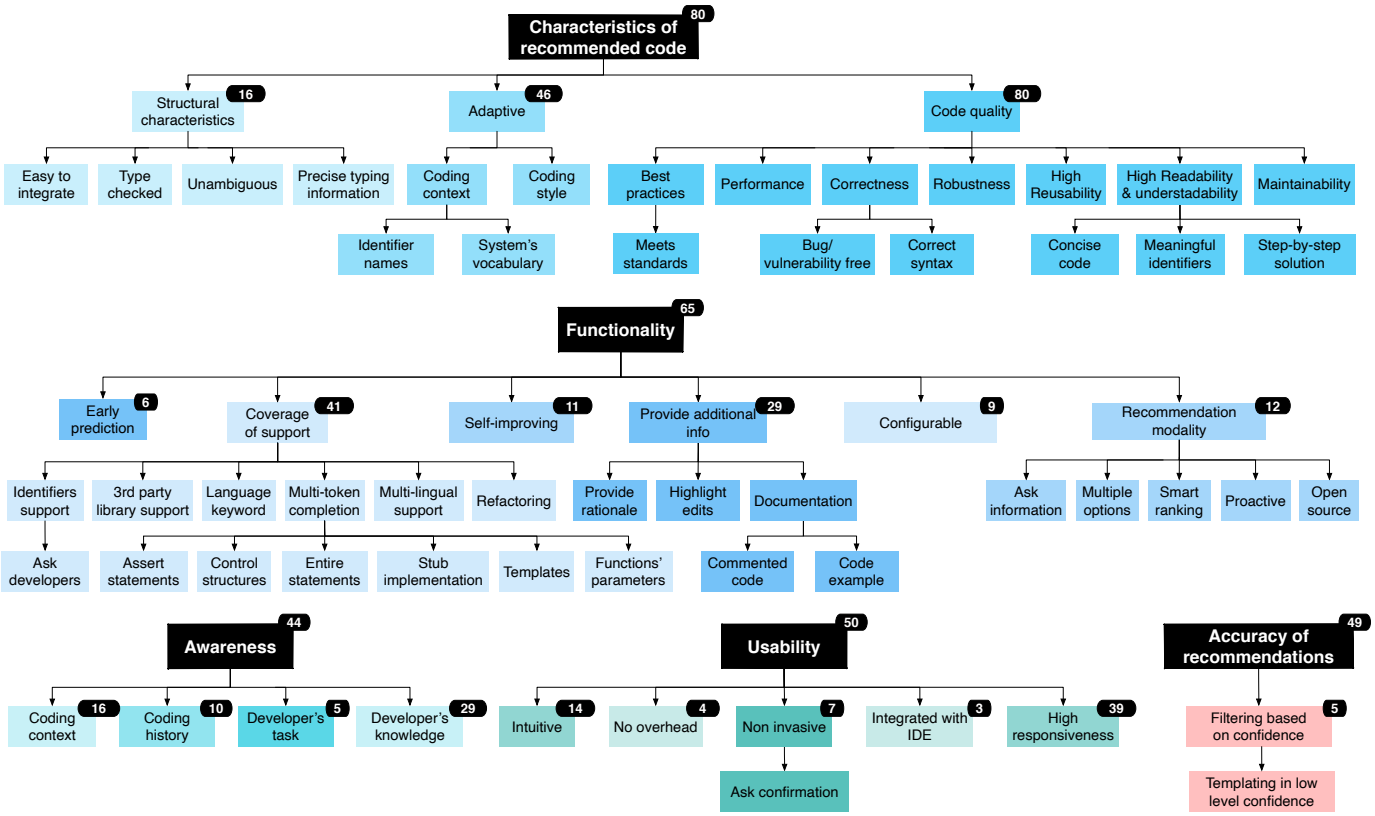

Fig. 2. Taxonomy of characteristics of code recommender systems

However, recent work by Ciniselli *et al.* [11] showed that, in the best case scenario, these models can recommend a correct code completion in less than 70% of cases, with much worst results (<30%) achieved for challenging completions (*e.g.,* recommend entire statements). While the study by Ciniselli *et al.* assessed the accuracy of the recommendations (*i.e.,* the extent to which the tool completes the code as expected), it is likely that a tool not providing an accurate recommendation breaks the code syntax or even introduces bugs. 💡 Our survey suggests that developers are unlikely to use tools that could potentially "break" their code in ∼35% of cases. For the reasons discussed, the notion of correctness is strongly related to the **accuracy of recommendations**, that is another root category in our taxonomy (see Fig. 2).

The majority of surveyed developers (61%) reported the accuracy of recommendations as an important characteristic of code recommenders (*e.g.,* "*inaccurate recommendations would create more work than manually writing the code*", "*the recommender system should have a minimum amount of false positive hits*", "*accuracy below 80% is not acceptable*"). 💡 Some developers even proposed solutions to overcome issues related to the low accuracy in specific scenarios (these led to the two child categories of *Accuracy of recommendations* in Fig. 2). For example, a participant mentioned "*it must be possible to configure the suggestions, to get less but more likely to be correct*".

Basically, the possibility to filter out recommendations in which the recommender system has a low confidence could help in addressing limited accuracy in challenging scenarios. Some developers also indicated their willingness to consider recommendations that are not 100% accurate as long as they can be easily modified to obtain the needed code: "*I often accept the recommendation even if it's not 100% accurate if I think that adjusting it takes less time than writing code from scratch*".

Going back to the *characteristics of recommended code* tree, developers are also looking for *adaptive* recommender systems: Developers mentioned that the tool should adapt the recommendations to their coding style (*e.g.,* "*once it learns my coding style it should go with that*") and to the coding context (*e.g.,* "*context-sensitivity is also important; I would like practical and stylistic compatibility with the existing code*").

💡 Automatically inferring the developer's coding style is far from trivial. One possibility would be to investigate the integration between approaches to learn coding style/conventions [5], [41] and code recommender systems. More in general, our survey seems to suggest the need for "modeling software developers and their coding practices" with all difficulties and privacy concerns this may lead to. Having such information may substantially boost the usefulness of code recommenders, that could better adapt their recommendation to the user receiving them.

Characteristics of recommended code

This category groups together specific code characteristics (detailed in the leaf nodes) that might be desirable for the recommended code.

- **Structural characteristics:** The recommended code meets specific structural characteristics of the code (detailed in the leaf nodes) that might be desirable.
  - **Easy to integrate:** The recommended code does not require major changes to be integrated in the code under development (e.g., a code using object types not available in the language but defined in other projects from which the recommended code has been learned is difficult to reuse).
  - **Type checked:** The recommender checks types when completing statements, ensuring values assigned to a variable are compatible with its type.
  - **Unambiguous:** The recommended code is unambiguous (e.g., it doesn't suggest multiple imports with the same name, making difficult to identify the one actually needed).
  - **Precise typing information:** The recommended code relies on specific types (e.g., java.time.LocalDate) rather than the generic one (e.g., Object).
- **Adaptive:** The recommended code is customized based on specific factors (detailed in the leaf nodes).
  - **Coding context:** The recommended code is adapted to the code the developer is writing (see leaf nodes).
  - **Coding style:** The recommended code is adapted to the developer's coding style (e.g., using extra parenthesis to better format code if this practice is used by the developer).
- **Code Quality:** The recommended code meets specific quality criteria (see leaf nodes).
  - **Best Practices:** The recommended code meets best coding practices. The latter depend on the language/paradigm in use.
  - **Performance:** The recommended code is optimized in terms of performance (e.g., it does not introduce unneeded operations).
  - **Correctness:** The recommended code must be bug-free.
  - **Robustness:** The recommended code is robust when dealing with possible erroneous situations (e.g., a null check is implemented if needed).
  - **High Reusability:** The recommended code is easy to reuse. This is particularly relevant for recommender systems suggesting code components at higher granularity (e.g., methods).
  - **High Readability & Understandability:** The recommended code must be readable (e.g., avoid very long statements, adopt indentation) and easy to understand.
  - **Maintainability:** The recommended code is easy to maintain.

Functionality

This category groups together the features developers would like to see in source code recommenders (see leaf nodes).

- **Early Prediction:** The recommender is able to trigger recommendations early in the coding process (e.g., the developer just wrote a few code tokens).
- **Coverage of Support:** This subcategory details the desiderata of developers when it comes to the coverage of the recommender tool (i.e., the different coding scenarios it can support).
  - **Identifiers support:** The recommender provides support for identifiers, suggesting them when needed.
  - **3rd party library support:** The recommender is able to propose code recommendations when dealing with code using third-party libraries (e.g., invocation of an API in a library).
  - **Language keywords:** The recommender is able to suggest the language keywords when needed.
  - **Multi-token completion:** The recommender can generate code recommendations that are not limited to a single token, but span several tokens.
  - **Multi-lingual support:** The recommender can generate code recommendations even when multi-lingual code statements (e.g., C# and SQL) are written by the developer.
  - **Refactoring:** The recommender is able to autocomplete a refactoring operation started by the developer.
- **Self-improving:** The recommender improves the suggestions based on the feedback received by the developer for past recommendations.
- **Provide additional info:** The recommender provides additional information together with the suggestion
  - **Provide rationale:** The tool provides a rationale to justify the given recommendation (e.g., you are seeing this code because ...).
  - **Highlight edits:** If the code that the developer is writing is very similar to code that can be recommended, the tool highlights the differences that might be implementation errors.
  - **Documentation:** The recommended code includes documentation.
- **Configurable:** The recommender is customizable based on developers' preferences (e.g., see leaf node).
- **Recommendation modality:** The modality used by the recommender when suggesting a recommendation.
  - **Ask information:** If the recommender cannot infer what the developer is doing it is able to ask for additional information to better contextualize the recommendations.
  - **Multiple options:** The recommender is able to generate multiple recommendations for a given scenario.
  - **Smart ranking:** When multiple options are available, the tool ranks the recommendations using a smart criterion rather than alphabetical order.
  - **Proactive:** The recommender perceives when the developer needs help (e.g., they are not writing for long time) and starts triggering recommendations. It does not need to be explicitly invoked by the developer.
  - **Open source:** If the recommended code is retrieved from a dataset, such a dataset must feature open source code which is less likely to result in licensing issues.

Awareness

This category groups together information items the recommender should be aware of to improve its recommendations.

- **Coding context:** The recommended code is based on a specific coding context, usually representing the code written by the developer in the IDE.
- **Coding history:** The recommended code is generated/improved based on development activities performed in the past. The past development activities could be related to the developer receiving the recommendation or to other developers. In other words, "natural" coding solutions are favored.
- **Developer's task:** The code recommender is aware of the task(s) the developer is working on. Differently from the coding context that only captures the code under development in a specific moment, in this case a more complete view of the developer's tasks is available (e.g., the tool could mine from the issue tracker the issues assigned to the developer and use this information to generate/improve its recommendations).
- **Developer's knowledge:** If multiple recommendations are possible, the one using code that is more familiar to the developer must be used (e.g., the code using APIs already used in the past by the developer receiving the recommendation).

Usability

This category groups together usability aspects they would like to see in source code recommenders (see leaf nodes).

- **Intuitive:** The recommender is intuitive (e.g., shallow learning curve).
- **No overhead:** The recommender does not hinder coding (e.g., a button can be pressed to accept/decline the recommendations) and does not require context-switch.
- **Non invasive:** The recommender is never in control of the code writing, it only provides suggestions.
  - **Ask confirmation:** The recommender asks confirmation to the developer before implementing code.
- **Integrated with IDE:** The recommender is integrated within the IDE.
- **High responsiveness:** The recommender is responsive, not causing lagging while coding.

Accuracy of recommendations

The recommended code is accurate (i.e., the recommender is able to suggest a code semantically equivalent to the one the developer was going to write).

- **Filtering based on confidence:** The recommender does not trigger recommendations when its confidence is low.
  - **Templating in low level confidence:** If the recommender has a low confidence about a recommendation, it can suggest a template for the recommendation rather than the raw code.

Fig. 3. Definitions for taxonomy's characteristics up to level-3

Finally, the *structural characteristics* subtree groups characteristics of the recommended code that concern its structure. These have been considered relevant by a lower number of developers (16). An interesting point to highlight here is the need for code recommendations that are *easy to integrate* in the code under development. This aspect is related to the adaptability of the recommendations: if the recommended code adapts to the coding context, for example, reusing identifiers when needed, it can make the developer's life easier. This is in contrast with what has been done in retrieval-based approaches that recommend relevant pieces of code from a code base (*e.g.,* Stack Overflow) leaving the integration effort on the developer's shoulders. 💡 An approach able to automate, at least in part, the integration process could substantially increase the usefulness of code recommenders.

Functionality. This category groups features developers would like to see in code recommenders. Most of the answers point to specific wishes in terms of *coverage of support*. This means that developers would like to have a wider support in terms of code recommendations, something that goes beyond the tools they currently use. Such a subtree supports several of the directions that the software engineering research community is currently investigating.

Approaches for *multi-token completion* (*i.e.,* code completion that goes beyond recommending the next token the developer is likely to write, for example, an entire statement) are a hot research topic [2], [11], [22], [24], [28] as well as the automated generation of *assert statements* [51], [56] (see Fig. 2). 💡 Our results show the need for techniques able to support developers in complex code completion scenarios (*e.g.,* "*code completion provides good support in finding the right API in a class and a few other things, but rarely suggests something challenging such as conditional statements*", "*cool would be suggesting which asserts are needed to test the code under development*"). While no developer asked for the automated implementation of whole functions from scratch [57], some mentioned the possibility to automatically implement *stub functions* starting from their signature as recently done in Copilot [2]: 💡 "*When writing new functions I often create stubs of the other functions I need to invoke; the IDE could propose implementations for those stubs*".

💡 An interesting research challenge also comes from the suggestion for *multi-lingual support*: "*autocomplete when mixing languages such as inline SQL code when writing database statements in C#*". Such a support is currently missing in code recommenders. Finally, for what concerns the *coverage of support*, it is also interesting the possibility to integrate the capabilities to autocomplete *refactoring* operations started by the developer. This is something that has been accomplished by Foster *et al.* [17] with their WitchDoctor tool.

Another representative category in the functionality tree is "*provide additional info*", which relates the will of developers to receive additional information accompanying the code recommendation.

This includes the possibility to *provide a rationale* justifying non-trivial code recommendations (*e.g.,* "*it should give a quick reason that I can understand why it's suggesting I do it*"), or documenting the recommended code (*e.g.,* "*assuming the recommended code is not trivial, a documentation/explanation for the suggestion is needed*").

While code recommenders mostly focus on generating meaningful code recommendations, retrieval-based techniques (*i.e.,* those retrieving relevant code from a code base) and properly trained DL-based techniques can provide support for the automated documentation of the recommended code [2], [20]. 💡 More challenging is the generation of a rationale explaining to the developer why a given recommendation is relevant for what they are doing.

Finally, the other child categories under *functionality* have been mentioned by a few developers. They concern: (i) the ability of the tool to improve over time based on accepted/rejected recommendations (*self-improving*); (ii) the possibility to configure aspects of the tool, such as defining shortcuts (*configurable*); (iii) the way in which the recommendations are presented or generated (*recommendation modality*), and (iv) the need for having the recommendation quickly triggered to avoid manually writing most of the code thus reducing the usefulness of the recommendation (*early prediction*). For the sake of brevity, these categories are described in Fig. 3.

Usability. This root category concerns aspects that influence the usability of the code recommender. Among all, *high responsiveness* was the most important requirement highlighted by developers (39 mentions) (*e.g., "these recommendation systems are often slow and lag the entire UI, especially on large projects"*). 💡 This confirms the relevance of works investigating efficiency aspects in code completion tools [48].

Along this line, participants expect tools which are *intuitive* (*e.g.,* allowing a gradual learning curve for all developers) and *well integrated with IDE*, as one developer underlined: *"clean interface making it easy to use is pretty important"*.

Finally, an interesting aspect highlighted by some developers is the need for *non invasive* code recommenders, always leaving the final word to the developer: "*It shouldn't change the code without my confirmation, even though the code I'm writing is not logically right; a warning would be nice*".

Awareness. The last root category in our taxonomy is *awareness*: the code recommender must be "aware" of different aspects to improve its recommendations. Most importantly from the participants' point of view (29 mentions) is the awareness of developer's knowledge. This implies that the same recommender system triggered on the same code by two different developers could produce different recommendations. 💡 Code recommendations using APIs familiar to the developer (*e.g.,* that the developer used in the past) can be favored as well as recommendations using code constructs that are familiar to the developer. As previously observed, this requires the ability of the recommender to "model" the developer's knowledge, exploiting it in generating the recommendations with the goal of minimizing the comprehension effort for the developer.

A second important aspect is the awareness of *coding context*, *i.e.,* the code recommender should consider the current code context (*e.g.,* the code the developer is writing in the IDE) when providing suggestions. It is important to explain the difference between *awareness of coding context* and *adapted to coding context* previously described: in the former case the recommended code is triggered by what has been written in the IDE (*e.g.,* it completes the implementation of a method under development), while in the latter the suggested code is adapted (*i.e.,* changed) to match up with the current code context (*e.g.,* to reuse identifiers present in the code). These two aspects should be combined together to obtain useful recommendations.

Other interesting requirements developers expressed are the awareness of *coding history* and of the *developer's tasks*. 💡 Exploiting the change history of the system on which recommendations are generated [42], [43] can "*help with repetitive tasks*". Instead, being aware of the tasks assigned to the developer to which recommendations are proposed can be exploited for a better customization of the recommendations. For example, the recommender can identify the issues assigned to the developer by mining the issue tracker, inferring the one they are working on in the IDE and targeting recommendations aimed at completing the issue being addressed.

IV. Validity Discussion

Threats to **construct validity** concern the relationship between theory and observation. The characteristics output of our study are personal opinions of the surveyed developers. To provide information about the "support" each characteristic had, we included the number of participants who mentioned the characteristics (at least for top-level categories, complete data available in [4]).

Threats to **internal validity** concern factors internal to our study that could have influenced the results. The participants to our survey are likely more interested in code recommenders than others, thus providing a "biased view" of the investigated phenomenon. However, they still provided quite different views on what it is important in code recommenders.

When presenting the results of our study, we often report "*quotes*" from the answers provided by participants. These quotes are not always verbatim, with changes introduced to fix typos or to shorten them without, however, changing their meaning.

To limit subjectivity bias during the open coding procedure, five authors independently inspected each answer we received, with a following discussion aimed at solving conflicts when needed. On top of that, the answers we collected together with the codes (*i.e.,* categories of our taxonomy) we assigned them are publicly available in our replication package [4].

Finally, contrary to our expectation, we found difficulties in recruiting software practitioners through MTurk. Indeed, when we run our survey, we did not set the strict selection criteria for participants described in Section II-A. This resulted in the collection of mostly low-quality answers, often clearly copied/pasted from online sources that we had to exclude.

Also, we had to forbid the access to our survey from specific countries due to bots providing random answers. At the end, we preferred to collect less answers but of high quality. Indeed, whenever we had a doubt about the quality of answers collected through our survey, we discarded the corresponding response.

Such a "quality issue" is less relevant when it comes to answers we collected through our contact network, which represent the majority of completed surveys in our study (49 out of 80). Indeed, those are all professional developers (*i.e.,* working in companies) who voluntarily agreed to participate in our survey. To check the extent to which our taxonomy generalizes to these two different groups of participants we involved (*i.e.,* practitioners from MTurk *vs* practitioners in our contact network) we checked the number of categories in our final taxonomy that have been indicated as relevant (i) by both groups, (ii) by MTurk participants only, and (iii) by practitioners in our contact network only. Out of the 70 categories in our taxonomy 51 (73%) have been indicated by participants belonging to both groups, 3 (4%) by MTurk's participants only, and 16 (23%) by practitioners in our contact network only. These results indicate an overall "agreement" among the two groups for two-thirds of our taxonomy. Also, the fact that only three out of 70 categories have been contributed exclusively by MTurk's participants support the validity of the answers collected through this platform, since most of the "requirements" we extracted from them have been confirmed by practitioners in our contact network. The three categories being MTurk-only are: *Functionality* → *Coverage of support* → *Language keywords*, *Functionality* → *Recommendation modality* → *Open source*, *Functionality* → *Recommendation modality* → *Proactive: perceives when developer needs help*.

Threats to **external validity** concern the generalization of our findings. The obvious threat is the limited number of participants involved in the survey (80). Such numbers are in line with previously published survey studies (*e.g.,* [8], [13], [16]) but, of course, replications can help in corroborating our findings and complementing them.

V. Related Work

In their seminal paper, Murphy *et al.* [35] found that developers rely on *content assist* (*i.e.,* code completion) features in IDEs as much as on other common editing features (*e.g.,* copy & paste). Such a result showed how improving code completion could have benefited developers. Code recommender systems greatly evolved since then.

For the sake of brevity, we do not focus on the many works focusing on improving code recommendations (*e.g.,* [5], [22]–[24], [26]–[28], [31], [33], [34], [37], [51], [52], [56], [57], [59]) but on empirical studies looking at code recommenders from different perspectives.

Proksch *et al.* [40] evaluated a method-call recommender system on a real-world dataset featuring interactions captured in the IDE. They observed that commonly used evaluations based on synthetic datasets extracted *a-posteriori* from released code do not take into account context change: This has a major effect on the prediction quality.

Hellendoorn *et al.* [18] compared code completion models on a real-world dataset and on synthetic datasets. They found that the experimented tools are less accurate on the real-world dataset, showing that synthetic benchmarks are not representative enough. Moreover, they found that such tools are less accurate in challenging scenarios, when developers would need them the most.

Ciniselli *et al.* [12] investigated the extent to which code recommenders copy code from their training set when generating recommendations. They found that ∼10% of of short recommendations represent clones of training set's code. However, as soon as the size of the recommended code increases (*e.g.,* a few statements), then it is unlikely that code recommenders copy code from the training set.

Mărășoiu *et al.* [29] studied how developers use code completion in practice. They observed that many recommendations are not accepted by the users. Similar findings have been reported by Arrebola and Junior [7], who advocate for context-awareness.

Jin and Servant [21] investigated the *hidden costs* of code recommendations. They found that the code completion tool they evaluated (IntelliSense) sometimes provides the right recommendation far from the top of the list. They observed that longer lists discourage developers from selecting a recommendation.

Xu *et al.* [61] run a controlled experiment with 31 developers who were asked to complete implementation tasks with and without the support of two code recommenders. They found no significant gain in developers' productivity when using the code recommenders.

Ziegler *et al.* [62] run a survey with Copilot users to investigate which quantitative measure better capture their perceived productivity when using the tool. They found that the acceptance rate of shown suggestions is the best predictor of perceived productivity.

The discussed papers suggest that a lack of grounding in reality may be detrimental for the advancement in such a field. We try to further fill this gap: while previous work mostly focused on specific aspects, such as accuracy [18], [40], lack of context-awareness [7], or hidden cost [21], we provide a complete developer-oriented view on the possible challenges in the design of code recommenders.

VI. Conclusion and Future Work

Our study partially fills the lack of empirical investigations aimed at collecting practitioners' *desiderata* when it comes to code recommenders. We run a survey involving a total of 80 practitioners to investigate characteristics of code recommenders they perceive as important. As output of our study, we defined a taxonomy of 70 characteristics (Fig. 2) that can drive future research in the field. We make all (anonymized) answers we collected, the tags we assigned to them, and study material available in our replication package hosted at https://code-recommenders.github.io.

Our future works stem from the findings of our surveys, and will focus on improving characteristics of code recommenders that our study highlighted as relevant. We detail three research directions we plan to pursue, with the goal of also showing how our taxonomy can be used to point at future research:

- *Improving the awareness of code recommenders*. One aspect several of the participants in our study stressed as important is what we defined as the "awareness of the code recommenders". In other words, what the information available to the recommender are when it synthesizes suggestions. Being aware of the developers' knowledge (*i.e.,* their expertise, past implementation tasks, etc.) could result in more relevant recommendations that might be particularly suitable and easy to understand and reuse for the developer receiving them. Integrating such a "knowledge" in the recommender is far from trivial and requires the development of techniques allowing to automatically infer (i) the programming style of software developers, and (ii) their expertises, namely the specific programming languages, libraries, notions (*e.g.,* design patterns), etc. they are at ease with. This could be done by mining the past developers' activities from software repositories (*e.g.,* versioning system, issue tracker).
- *Making code quality a first-class citizen in code recommendations*. Given the increasing complexity of the recommendations supported by tools such as GitHub Copilot, code quality must become a priority for the recommended code. This is a clear outcome of our survey. Important aspects to focus on are the absence of bugs/vulnerabilities and the promotion of readability and understandability. All these quality aspects can benefit from tailored design decisions made (i) when training the code recommender, by curating the quality of the code snippets composing the training set; and (ii) at post-processing stage, with checks done before triggering the recommendation to the user. Also, alternative recommendations could be ranked based on their quality.
- *Augmenting the coverage of support*. Despite the gigantic steps ahead made in the last few years, code recommenders still struggle in specific coding scenarios for which they do not offer (or offer limited) support. In this context, two interesting research directions stemming from our taxonomy are the (i) better support for multi-lingual code, and (ii) generation of templates rather than raw source code when the recommender is not confident. Concerning the first point, we plan to assess the effectiveness of recent transformer models trained on several programming languages (*e.g.,* CodeBERT [15]) when dealing with multi-lingual code. Depending on the observed performance, the proposal of alternative strategies to deal with this problem will be investigated. As for the template generation, we plan to work on a model specifically trained for generating abstract code templates rather than raw source code. Such a model could be triggered when the standard code recommender is not confident in suggesting the raw code.

ACKNOWLEDGMENT

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 851720).

REFERENCES

[1] "Amazon mechanical turk https://www.mturk.com."
[2] "Github copilot https://copilot.github.com."
[3] "Qualtrics https://www.qualtrics.com."
[4] "Replication package https://code-recommenders.github.io."
[5] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Learning natural coding conventions," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014, 2014, pp. 281–293.
[6] L. E. d. S. Amorim, S. Erdweg, G. Wachsmuth, and E. Visser, "Principled syntactic code completion using placeholders," ser. SLE 2016, 2016, p. 163?175.
[7] F. V. Arrebola and P. T. A. Junior, "On source code completion assistants and the need of a context-aware approach," in *International Conference on Human Interface and the Management of Information*. Springer, 2017, pp. 191–201.
[8] G. Bavota, B. Dit, R. Oliveto, M. D. Penta, D. Poshyvanyk, and A. D. Lucia, "An empirical study on the developers' perception of software coupling," in *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*. IEEE Computer Society, 2013, pp. 692–701.
[9] G. Bavota, A. D. Lucia, A. Marcus, and R. Oliveto, "Automating extract class refactoring: an improved method and its evaluation," *Empir. Softw. Eng.*, vol. 19, no. 6, pp. 1617–1664, 2014.
[10] R. P. L. Buse and W. Weimer, "Learning a metric for code readability," *IEEE Transactions on Software Engineering*, vol. 36, no. 4, pp. 546–558, 2010.
[11] M. Ciniselli, N. Cooper, L. Pascarella, A. Mastropaolo, E. Aghajani, D. Poshyvanyk, M. D. Penta, and G. Bavota, "An empirical study on the usage of transformer models for code completion," *IEEE Transactions on Software Engineering*, no. 01, pp. 1–1, 2022.
[12] M. Ciniselli, L. Pascarella, and G. Bavota, "To what extent do deep learning-based code recommenders generate predictions by cloning code from the training set?" in *IEEE/ACM 19th International Conference on Mining Software Repositories, MSR 2022, Pittsburgh, PA, USA, May 23-24, 2022*. IEEE, 2022, pp. 167–178.
[13] S. C. B. de Souza, N. Anquetil, and K. M. de Oliveira, "A study of the documentation essential to software maintenance," in *Proceedings of the 23rd Annual International Conference on Design of Communication: Documenting & Designing for Pervasive Information*, ser. SIGDOC '05. ACM, 2005, pp. 68–75.
[14] J. Dorn, "A general software readability model," *MCS Thesis available from (http://www. cs. virginia. edu/weimer/students/dorn-mcs-paper. pdf)*, vol. 5, pp. 11–14, 2012.
[15] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "CodeBERT: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020*. Online: Association for Computational Linguistics, Nov. 2020, pp. 1536–1547.
[16] A. Forward and T. C. Lethbridge, "The relevance of software documentation, tools and technologies: A survey," in *Proc. of the 2002 ACM Symp. on Doc. Eng. (DocEng)*. ACM, 2002, pp. 26–33.
[17] S. R. Foster, W. G. Griswold, and S. Lerner, "Witchdoctor: Ide support for real-time auto-completion of refactorings," in *2012 34th International Conference on Software Engineering (ICSE)*, 2012, pp. 222–232.
[18] V. J. Hellendoorn, S. Proksch, H. C. Gall, and A. Bacchelli, "When code completion fails: A case study on real-world completions," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 960–970.
[19] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation," ser. ICPC '18, 2018.
[20] Y. Huang, S. Huang, H. Chen, X. Chen, Z. Zheng, X. Luo, N. Jia, X. Hu, and X. Zhou, "Towards automatically generating block comments for code snippets," *Information and Software Technology*, vol. 127, p. 106373, 2020.
[21] X. Jin and F. Servant, "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*, 2018, pp. 70–73.
[22] R. Karampatsis and C. A. Sutton, "Maybe deep neural networks are the best choice for modeling source code," *CoRR*, vol. abs/1903.05734, 2019. [Online]. Available: http://arxiv.org/abs/1903.05734
[23] J. Kim, S. Lee, S. Hwang, and S. Kim, "Adding examples into java documents," in *2009 IEEE/ACM International Conference on Automated Software Engineering*, 2009, pp. 540–544.
[24] S. Kim, J. Zhao, Y. Tian, and S. Chandra, "Code prediction by feeding trees to transformers," *arXiv preprint arXiv:2003.13848*, 2020.
[25] H. H. S. Kyaw, S. T. Aung, H. A. Thant, and N. Funabiki, "A proposal of code completion problem for java programming learning assistant system," in *Conference on Complex, Intelligent, and Software Intensive Systems*. Springer, 2018, pp. 855–864.
[26] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each," in *2012 34th International Conference on Software Engineering (ICSE)*, 2012, pp. 3–13.
[27] Y. Li, S. Wang, and T. N. Nguyen, "Dlfix: Context-based code transformation learning for automated program repair," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20, 2020, p. 602?614.
[28] F. Liu, G. Li, Y. Zhao, and Z. Jin, "Multi-task learning based pre-trained language model for code completion," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2020. Association for Computing Machinery, 2020.
[29] M. Mărășoiu, L. Church, and A. Blackwell, "An empirical investigation of code completion usage by professional software developers," in *Proceedings of the 26th Annual Workshop of the Psychology of Programming Interest Group*, 2015.
[30] M. R. Marri, S. Thummalapenta, and T. Xie, "Improving software quality via code searching and mining," in *2009 ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation*. IEEE, 2009, pp. 33–36.
[31] C. McMillan, D. Poshyvanyk, M. Grechanik, Q. Xie, and C. Fu, "Portfolio: Searching for relevant functions and their usages in millions of lines of code," *ACM Trans. Softw. Eng. Methodol.*, vol. 22, no. 4, pp. 37:1–37:30, 2013.
[32] Q. Mi, J. Keung, Y. Xiao, S. Mensah, and Y. Gao, "Improving code readability classification using convolutional neural networks," *Information and Software Technology*, vol. 104, pp. 60–71, 2018.
[33] L. Moreno, G. Bavota, M. Di Penta, R. Oliveto, and A. Marcus, "How can i use this method?" in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE '15, 2015, p. 880?890.
[34] L. Moreno, G. Bavota, M. D. Penta, R. Oliveto, A. Marcus, and G. Canfora, "Arena: An approach for the automated generation of release notes," *IEEE Transactions on Software Engineering*, vol. 43, no. 2, pp. 106–127, 2017.
[35] G. C. Murphy, M. Kersten, and L. Findlater, "How are java software developers using the elipse ide?" *IEEE software*, vol. 23, no. 4, pp. 76–83, 2006.
[36] S. M. Nasehi, J. Sillito, F. Maurer, and C. Burns, "What makes a good code example?: A study of programming q a in stackoverflow," in *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, 2012, pp. 25–34.
[37] T. Nguyen, P. C. Rigby, A. T. Nguyen, M. Karanfil, and T. N. Nguyen, "T2api: Synthesizing api code usage templates from english texts with statistical translation," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016, 2016, p. 1013?1017.
[38] D. Perelman, S. Gulwani, T. Ball, and D. Grossman, "Type-directed completion of partial expressions," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '12, 2012, p. 275?286.
[39] D. Posnett, A. Hindle, and P. Devanbu, "A simpler model of software readability," in *Proceedings of the 8th working conference on mining software repositories*, 2011, pp. 73–82.
[40] S. Proksch, S. Amann, S. Nadi, and M. Mezini, "Evaluating the evaluations of code recommender systems: a reality check," in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2016, pp. 111–121.
[41] S. P. Reiss, "Automatic code stylizing," in *Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '07, 2007, p. 74?83.

[42] R. Robbes and M. Lanza, "How program history can improve code completion," in *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, 2008, pp. 317–326.

[43] ——, "Improving code completion with program history," *Automated Software Engineering*, vol. 17, no. 2, pp. 181–212, 2010.

[44] M. P. Robillard, W. Maalej, R. J. Walker, and T. Zimmermann, *Recommendation Systems in Software Engineering*. Springer Publishing Company, Incorporated, 2014.

[45] S. Scalabrino, M. Linares-Vásquez, R. Oliveto, and D. Poshyvanyk, "A comprehensive model for code readability," *Journal of Software: Evolution and Process*, vol. 30, no. 6, p. e1958, 2018.

[46] R. Schuster, C. Song, E. Tromer, and V. Shmatikov, "You autocomplete me: Poisoning vulnerabilities in neural code completion," 2020.

[47] A. Svyatkovskiy, S. K. Deng, S. Fu, and N. Sundaresan, "Intellicode compose: Code generation using transformer," *arXiv preprint arXiv:2005.08025*, 2020.

[48] A. Svyatkovskiy, S. Lee, A. Hadjitofi, M. Riechert, J. V. Franco, and M. Allamanis, "Fast and memory-efficient neural code completion," in *18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021*. IEEE, 2021, pp. 329–340.

[49] A. Tamrawi, T. T. Nguyen, J. M. Al-Kofahi, and T. N. Nguyen, "Fuzzy set and cache-based approach for bug triaging," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE '11, 2011, p. 365?375.

[50] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou, "Ten years of jdeodorant: Lessons learned from the hunt for smells," in *25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018*, R. Oliveto, M. D. Penta, and D. C. Shepherd, Eds. IEEE Computer Society, 2018, pp. 4–14.

[51] M. Tufano, D. Drain, A. Svyatkovskiy, and N. Sundaresan, "Generating accurate assert statements for unit test cases using pretrained transformers," *CoRR*, vol. abs/2009.05634, 2020.

[52] M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, and D. Poshyvanyk, "An empirical study on learning bug-fixing patches in the wild via neural machine translation," *ACM Trans. Softw. Eng. Methodol.*, vol. 28, no. 4, pp. 19:1–19:29, 2019.

[53] R. Tufano, S. Masiero, A. Mastropaolo, L. Pascarella, D. Poshyvanyk, and G. Bavota, "Using pre-trained models to boost code review automation," in *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. IEEE, 2022, pp. 2291–2302.

[54] R. Tufano, L. Pascarella, M. Tufano, D. Poshyvanyk, and G. Bavota, "Towards automating code review activities," in *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 2021, pp. 163–174.

[55] W. Wang, S. Shen, G. Li, and Z. Jin, "Towards full-line code completion with neural language models," *arXiv preprint arXiv:2009.08603*, 2020.

[56] C. Watson, M. Tufano, K. Moran, G. Bavota, and D. Poshyvanyk, "On learning meaningful assert statements for unit test cases," in *Proceedings of the 42nd International Conference on Software Engineering, ICSE 2020*, 2020, p. To Appear.

[57] F. Wen, E. Aghajani, C. Nagy, M. Lanza, and G. Bavota, "Siri, write the next method," in *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 2021, pp. 138–149.

[58] X. Xia, D. Lo, Y. Ding, J. M. Al-Kofahi, T. N. Nguyen, and X. Wang, "Improving automated bug triaging with specialized topic model," *IEEE Transactions on Software Engineering*, vol. 43, no. 3, pp. 272–297, 2017.

[59] T. Xie and J. Pei, "Mapo: Mining api usages from open source repositories," ser. MSR '06, 2006.

[60] F. F. Xu, Z. Jiang, P. Yin, B. Vasilescu, and G. Neubig, "Incorporating external knowledge through pre-training for natural language to code generation," in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, Jul. 2020.

[61] F. F. Xu, B. Vasilescu, and G. Neubig, "In-ide code generation from natural language: Promise and challenges," 2021.

[62] A. Ziegler, E. Kalliamvakou, X. A. Li, A. Rice, D. Rifkin, S. Simister, G. Sittampalam, and E. Aftandilian, "Productivity assessment of neural code completion," ser. MAPS 2022, 2022, p. 21?29.