# Generating Context-Aware API Calls from Natural Language Description using Neural Embeddings and Machine Translation

Hung Phan
*Department of Computer Science*
*Iowa State University*
Ames, USA
hungphd@iastate.edu

Arushi Sharma
*Department of Computer Science*
*Iowa State University*
Ames, USA
arushi17@iastate.edu

Ali Jannesari
*Department of Computer Science*
*Iowa State University*
Ames, USA
jannesari@iastate.edu

***Abstract*—API calls can be described using natural language and can be implemented using a programming language. A programming environment that can process natural language descriptions within source code and provide context aware code suggestions can significantly improve productivity of developers and make programming more accessible to non-programmers and less experienced programmers. This paper proposes a context-aware tool called API Call Programming Interface (ACPI) which allows developers to write a natural language description for methods within source code and get correct and compilable API Call (AC) based on the text description and surrounding code. Existing work and code suggestion tools only consider the user's natural language description as input and ignore the contextual surrounding code. We take surrounding code into account and include information about local variable names within the code suggestion. Our approach consists of three modules. First, Method Name Generator, an unsupervised neural-embeddings-based algorithm to map the natural language description of methods to a list of most likely method names. Second, an AST Generator, a supervised Machine Translation model that predicts the structure of the AST from the list of the method names. And third, a Code Synthesizer that assigns local variables names to the AST to get the final method calls. Further, we include a Ranking Module that ranks the list of suggested method names based on their completeness. We evaluated our approach on data from 1000 high-quality Java projects and achieved an accuracy of 61% for API calls suggestions from natural language descriptions, which outperforms prior work and demonstrates the potential of our approach. We also conducted productivity experiments with 148 undergraduate participants to measure the usefulness of ACPI. The experiment showed that programming with ACPI can reduce programming time by 45% and increase programming accuracy from 19% to 83% when compared to programming without ACPI in four code completion tasks.**

***Index Terms*—API Call, Machine Translation, Program Synthesis**

## I. Introduction

Developing large and complex software projects requires well trained and experienced programmers. First, they need to be familiar with the basic statements or structures of programming languages they are working with. Since each programming language (PL) has a list of common keywords, developers need to remember the vocabulary of keywords and know how to use them to be able to implement a program in that language. Usually, developers are able to use this type of vocabulary after a short time period because of the finite size of the keywords. However, to be able to develop complex functions, developers need to inherit previously written code using a method call (MC). There are two types of method calls that can be used in the Java programming language. The first is the use of a local method which was previously developed by themselves or other developers who are working on the same software project. The second way is to use application programming interface (API) calls which are functions developed by well-known software organizations. Unlike vocabularies of keywords and API calls, the size of API calls is much larger which causes challenges for developers to use them for doing software programming tasks. Working with API calls requires experience in programming languages since the results returned by engines are large and the API calls can be obsolete [1].

To alleviate the task of searching correct API calls from natural language descriptions, there are several approaches to automatically generate the API calls from natural language. anyCode [2] provides a solution for using Context-Free Grammar (CFG) as a language model for code generation from NL description of API calls. NLPToCode [3] and NLI [4] output code templates with missing elements like parameters, which the developers are asked to input manually. Both of these methods don't take advantage of the surrounding code and code context of NL description to infer the code from NL. In practice, the NL description can be dependent on the surrounding code. In the example shown in Listing 1, Google search engine returned the popular print statement in Java as $System.out.println$, while the actual result shows the API Call as $PrintWriter.println$ instead.

To the best of our knowledge, there is no other work in Software Engineering (SE) and Machine Learning (ML) that is able to directly generate source code from context-aware natural language descriptions while this is considered an important research problem in Software Engineering. In related work, beside anyCode [2] and NLP2Code [3] mentioned above, other works appliy and optimize Machine Translation (MT) using

978-1-6654-3583-3/21/$31.00 ©2021 IEEE
DOI 10.1109/ASEW52652.2021.00050

supervised approaches for inferring code from natural language descriptions and vice versa. [5] proposes tree-based code generation which can be integrated with a Recurrent Neural Network (RNN) for Python. [6] applies a Statistical Machine Translation (SMT) to learn pseudo-code from actual code for code-summarization applications. This trend of research requires manually annotated data for supervised learning, which is expensive when applied to other types of natural language. Creating a parallel corpus of natural language descriptions and code from large-scale code repositories, is usually erroneous and noisy. Barone et al. [7] created a Python corpus by automatically extracting the documentation and implementation of the corresponding Python methods. They found that applying Machine Translation models on such a corpus is challenging and the accuracy on both Statistical Machine Translation and Neural Machine Translation (NMT) models was quite low. None of this work [2], [5]–[7] on generating code from natural language as input can successfully generate correct and context aware code from natural language descriptions.

Additionally, DeepAPI [8] and SWIM [9] provide code completion by outputting a list of application programming interfaces (API's). While this approach can capture information about complex code structures such as loop conditions. it does not generate compilable code, only suggestions for which APIs can be used in a particular situation. To get the final code, developers need to manually combine and implement the correct API from the list of suggested APIs. NLI [4] is the most recent tool for natural language to source code translation to the best of our knowledge. It outputs a code template with missing elements like parameters, which the developers are asked to input. Even though their code template is helpful, it requires the developers to manually enter additional information as an intermediate step before code generation. Moreover, NLI can only support a specific list of natural language statements and does not work as well for arbitrary natural language descriptions. Since programming is a challenging task, we aim to avoid any extra effort by developers unlike of these prior works.

This paper is an attempt to improve the productivity and programming experience of developers by proposing API Call Programming Interface (ACPI), a tool that helps developers use natural language descriptions of API calls within the coding environment to automatically suggest context-aware and compilable code as API Calls. This is different from prior work because we consider the context i.e surrounding sequence of code tokens in addition to the user's natural language description to generate source code snippets. We provide a combined approach which is a combination of three techniques: generation of neural embeddings to get method names from surrounding context and natural language elements, machine translation to obtain the abstract syntax tree of method names, and code synthesis to add relevant contextual information from surrounding context (i.e. names of local variables) to the tree-representation and obtain the final code. To summarize, we make the following contributions:

**1.** An unsupervised neural-embeddings-based approach for generating method names based on natural language descriptions and surrounding context(i.e code tokens before and after the natural language description).
**2.** A machine translation model which converts method names to an abstract syntax tree as API calls.
**3.** API Call Programming Interface (ACPI), a code suggestion tool from natural language and surrounding context to generate the correct code.
**4.** Human productivity experiments on programmers of multiple skill levels to study the usefulness of code suggestions by ACPI.

Listing 1. Example of Context-Aware Natural Language inside Android project [10]

```
1  public void printNextItem(PrintWriter pw,
       HistoryItem rec, long now) {
2   pw.print("  ");
3   TimeUtils.formatDuration(rec.time-now, pw
        , TimeUtils.HUNDRED_DAY_FIELD_LEN);
4   pw.print(" ");
5   if (rec.cmd == HistoryItem.CMD_START) {
6    // println " START"
7    ...
8   } else {
9    if (rec.batteryLevel < 10) pw.print("00"
         );
10   else if (rec.batteryLevel < 100) pw.
         print("0");
11   pw.print(rec.batteryLevel);
12  }
13 }
```

## II. Background

In this section, we define some of the important terms used in this work. Aside from terms that are well-known in Software Engineering, we have defined a set of terms that are fundamental to our approach.

**Code Snippets** These refer to sections of code containing the API Call and surrounding code or context in a body of method declaration.

**NL-PL-Snippets (Natural Language-Programming Language-Snippets** refer to the source code that contains natural language description of API calls. An example of NL-PL-Snippets can be shown in Listing 1. In this example, this NL-PL-Snippet has line 6 as NL description and the surrounding code as the block from line 2 to line 12 except line 6.

## III. Approach

### A. The API Call Programming Interface Model

Code suggestions provided by ACPI are generated using the process illustrated in Figure 1. First, given an input code snippet with its natural language description, A) We use our Method Name Generator to extract tokens and variable names from natural language descriptions and surrounding code tokens and predict possible method names. The Method Name Generator comprises of the following two modules: Feature Extraction and Neural Embedding modules. We perform feature extraction

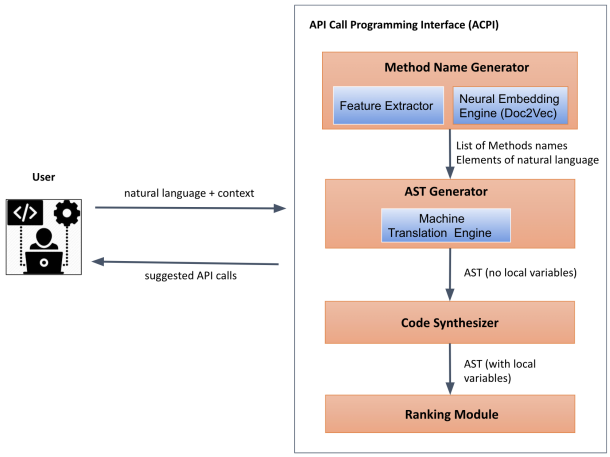


Fig. 1. ACPI's workflow diagram

to extract tokens from the raw text and surrounding code. We then use an unsupervised neural-embeddings-based approach to obtain a list of most-likely method names. Second, B) our AST Generator, a Statistical Machine Translation model (SMT), generates an AST for each method name. Third, C) the Code Synthesizer uses the the tokenized variable names extracted from the user's natural language description to add contextual information and complete the AST, and produce a final list of API Calls. Finally, D) a Ranking Module scores and sorts each translated candidate based on completeness to select the most suitable one. Natural Language-Programming Language-Snippets (NL-PL-Snippet) is the main input of our problem. NL-PL-Snippets refer to the source code that contains natural language description of API calls. An example of NL-PL-Snippets can be shown in Listing 1.

### *B. Method Name Generator*

The main purpose of this module is to generate the list of most-likely method names from the user input, which comprises of natural language and the surrounding code. Since we cannot guarantee high text similarity between natural language and method names due to the ambiguity of natural language [11], we use surrounding code to create contextual embeddings for both. Our main idea is to find in the large-scale code corpus the method names that have the most similar surrounding code to the surrounding code of user's current NL description.

The illustration of Method Name Generator shown in Figure 3 demonstrates the implementation of this concept. We have 2 phases: training and testing. In the training phase, the vectorizer learns information about the context of the method name for every code snippet in the corpus. The output of training step is a data set of method names and their neural embeddings from different code contexts.We call this the neural embeddings database. Since method names can appear in multiple code contexts, a method name can have different vector representations for different code contexts. In the testing phase, the user's input NL-PL Snippet is also represented by a vector. We compare the vector representation of the user input in the testing phase to the vectors in the neural embeddings database to find the method names with the most similar context. We use cosine similarity to provide this list of the most-likely method name candidates.

**Neural Embeddings for Code Snippets in training.** In the training phase for method name generator, tokens inside method declarations or Code Snippets in training data set are used for training. At the end of the training phase, we have a trained model with the ability to generate vectors from natural language and a database of vectors for method names along with their context in the training data set.

**Neural Embeddings for NL-PL-Snippets in testing.** In the testing phase, we integrate tokens of the surrounding code and tokens of the NL description to create a textual representation of NL-PL-Snippets. The vector for the NL-PL-Snippet will be generated by the model we trained in the previous step i.e. the training phase.

**Comparison** The vector representations for NL-PL-Snippets is compared with the database of vectors from training. This gives us the list of method names whose vector representations are closest to that of a particular NL-PL-Snippet.

The core engines of this module are feature extraction and neural embeddings.

*1) Feature Extractor: Code and natural language tokenization:* Since our approach relies on the comparison between natural language and method names which have similar context, we also provide a feature extraction module to capture information about method names and tokens of their surrounding codes as shown in Figure 3. The output of feature extraction contains 2 important elements. In the training phase, it extracts the code context, natural language and method names as tokens, which are useful for comparison between natural language and method names. In the testing phase, it provides a list of tokens and their roles in the natural language description like local variables, which can be used in the Code Synthesizer.

*2) Neural Embeddings: predicting list of method names:* Generating method names from natural language is non trivial due to the the lack of labeled data sets for supervised Machine Translation. While there exist large scale parallel corpora for most Machine Translation tasks related to natural language processing, this is not the case for converting natural language to code. Therefore, we propose an unsupervised neural embeddings based approach which is more suitable than expensive supervised machine learning methods which would require the creation of a large parallel corpus of programming language and its natural language description by manually labelling. Our approach proposes a neural embedding, which maps natural language to method names as an intermediate step for code completion.

A technical problem while designing a neural embedding to get method names from an NL-PL-Snippet is how to embed a context-aware API calls. This is significant because the same API call can appear in multiple contexts. To address this, we integrate code tokens of all the method names of

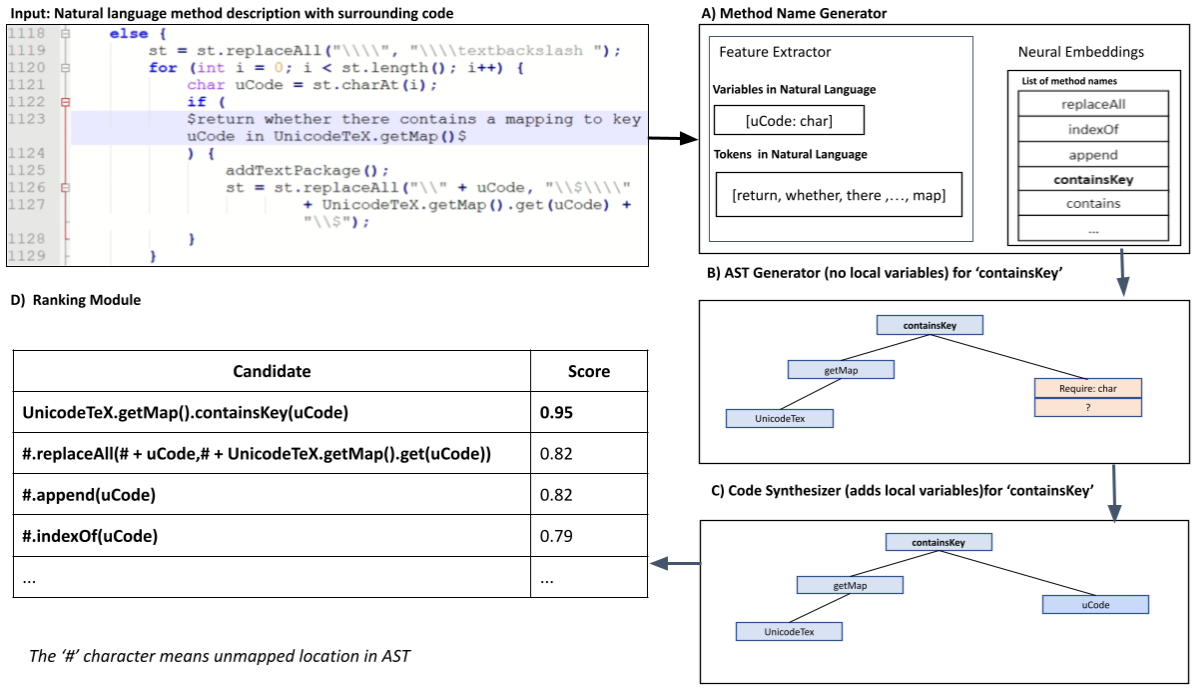


Fig. 2. Dataflow diagram of motivating example of ACPI

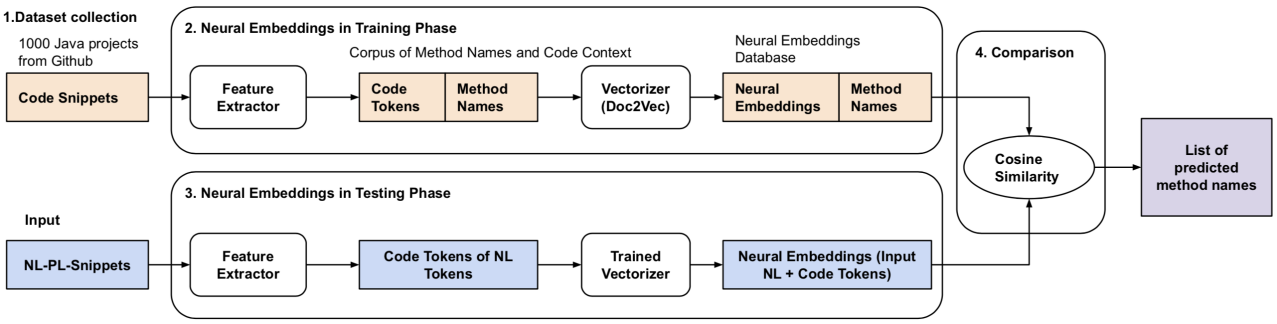


Fig. 3. Process of Method Name Generator using Feature Extraction and Doc2Vec Neural Embedding in ACPI

API calls along with code tokens of surrounding code as a training record for neural embedding. Our Method Name Generator uses Doc2Vec [12], an extension of well-known word vectorization technique Word2Vec [13], to generate neural embeddings for every code snippet in our dataset and create the Neural Embeddings Database. Code2Vec [14] is another approach which uses neural networks to represent snippets of code as continuous distributed vectors or "code embeddings". However, this approach is not suitable for us as we are also using natural language descriptions within our code snippets. Doc2Vec treats the input for vector representation as sequence of code tokens, which is feasible for the mix environment of code and natural language in our problem.

### C. AST Generator: Mapping AST from method names

We use a Statistical Machine Translation approach called Phrase-Based Machine Translation (PBMT) to predict AST from method names. The input of the PBMT model is tokenized code snippets with method names. The output of PBMT model is the code tokens and representation of AST for API calls. Since the ASTs of API calls are represented using a tree structure which cannot be obtained directly by sequence to sequence translation, we create a database of ASTs in a dictionary and use the key for each AST as an unique token on the target side of the parallel corpus of method names - ASTs of API calls.

Further, the AST generated in this step does not contain the names of the local variables. We abstracted the local variables

by their type information. This type information is later used to retrieve the local variable names and complete the AST in the Code Synthesizer module. Figure2 part B shows an example of an incomplete AST generated by the AST generator that have a place holder as type $char$ in Java.

**Statistical Machine Translation** We use Statistical Machine Translation (SMT) [15] to learn the mapping between source and target language. Statistical Machine Translation consists of two models: the language model and the translation model. While the language model measure the probability of a word or phrase appeared in the target language, the Phrase Based Machine Translation model calculates the probability of a phrase from source language translating into a phrase in a target language.

Neural Machine translation is another popular approach to translation in the field of natural language processing. However, we discovered that Neural Machine Translation models produced worse results than Statistical Machine Translation on our dataset. According to [16], Neural Machine Translation models perform poorly on source code as code introduces vocabulary at a much higher rate than natural language, due to a proliferation of new identifier names. This leads to large vocabulary and out-of-vocabulary issues which significantly degrade their performance. Another approach is to use Byte Pair Encoding [16] for Neural Machine Translation by dividing word to sub-words. However, the generated output for this work is sub-words which are not our expected output. We require the output as the tokens of ASTs which represented the structure of compilable code. Therefore, we choose SMT as our primary translation engine from method names to API calls.

### D. Code Synthesis: Assigning variables to AST

The output of machine translation for each code snippet in our dataset is a list of AST without local variable names. This module is used to complete this missing information. The input to this algorithm includes the list of variables in natural language description and the corresponding AST. Nodes that do not have any variables are checked to see if they match with one of the variables of the raw text. The AST provides the type of the missing variables, which are then compared with information about local variables in the raw text. In example in Figure 2, the variable `uCode` in NL description will be assigned to AST to make the API call since it has the type as char, which is consistent to the missing place of AST.

### E. Ranking Module: Ranking translated candidates

This module sorts the list of translated API Calls based on their completeness in terms of both Natural Language and Programming Language. For measuring completeness, we provide a heuristic approach which uses a combination of four scores. These are ScoreMatchOfSAST (S1), ScoreComplexityOfSAST (S2), ScoreVarsNLE (S3), ScoreTermsNLE (S4). **S1** This score penalizes the API call with missing local variables. These missing variables occur when there are cases where there are no variables in the surrounding code and in the natural language description that match the expected types of the missing variables in the AST. Since the AST with missing variables requires developers to input the variables manually, we reduce the score of suggestion for this case. **S2** penalizes an API call with a simple structure. This score penalizes the cases where the AST is too obvious and easy for the requirement for the code suggestion task. **S3** penalizes an API call if there are variables in the raw text that it does not include. **S4** penalizes a API call that does not include several terms from the natural language description.

The total score **S** for ranking of AC **m** is calculated using Equation 1. We have fixed four parameters $\alpha, \beta, \gamma, \delta$ as $0.6, 0.2, 0.1, 0.1$ respectively.

$$S(m) = \alpha * S1(m) + \beta * S2(m) + \gamma * S3(m) + \delta * S4(m) \quad (1)$$

TABLE I
DATASET OVERVIEW

| Item | Number |
|---|---|
| Java projects | 1000 |
| Num of method names and context for training | 1770000 |
| Size of code snippets for training | 800000 |
| Num of developers to conduct data set for evaluation | 6 |
| Num of code snippets for evaluation | 100 |
| Num of inexperienced developers in RQ 2 | 148 |
| Num of programming tasks in RQ 2 | 57 |

## IV. EVALUATION

In this section, we evaluate ACPI from multiple perspectives. Our research questions are as follows:

**RQ 1** What is the overall accuracy of ACPI for code generation from a given natural language description, compared to state-of-the-art work?
**RQ 2** How well can ACPI improve accuracy and productivity of inexperienced developers?

### A. Data Preparation

We use an automatic approach for collecting high-quality Java projects in our evaluation. In this approach, we use the Github API to collect the most-starred Java projects for the training data set. Experienced developers were hired to create NL-PL-Snippets for the evaluation data set manually. Statistics on our data set are shown in Table I. anyCode [2] is the only reported (but not publicly available) similar data set and it includes 45 natural language queries and their corresponding code. However, we cannot use this data set for several reasons. First, some software libraries used in the evaluation data set of this corpus are outdated, for e.g. `java.awt` (which has been replaced by `javax.swing`). Second, the expected results of their queries are API calls with no contextual information like local variable names, which need to be filled up manually into a predicted template..Further, many expected API calls in this data set do not appear in well-known Q&A forums such as StackOverflow [17] and ProgramCreek [18].

TABLE II
ANALYSIS BETWEEN EXPECTED RESULTS AND TRANSLATED RESULTS

| ID | Natural Language | Expected result | Translated result |
|---|---|---|---|
| 21 | return random number with max value iterationWeight for Random | new Random().nextInt(iterationWeight) | new Random().nextInt(iterationWeight) |
| 100 | create and return EntityManagerFactory for "options" using Persistence | Persistence. createEntityManagerFactory ("options") | Persistence. createEntityManagerFactory ("options") |
| 25 | return whether there contains a mapping to key uCode in UnicodeTeX.getMap() | UnicodeTeX.getMap() .containsKey(uCode) | UnicodeTeX.getMap() .containsKey(uCode) |
| 80 | return the minimum of lp.height and height | Math.min(lp.height,height) | Math.min(height,lp.height) |
| 20 | format now.getTime() into String and append to dateFormatter for SimpleDateFormat | dateFormatter.format(now.getTime()) | dateFormatter.format(now.getTime()) .toString() |
| 68 | return a color from default key "MenuItem.acceleratorForeground" for UIManager | UIManager. getColor( "MenuItem. acceleratorForeground" ) | UIManager.put( "MenuItem. acceleratorForeground", color) |

TABLE III
RESULT OF COMPARISON ON PRODUCTIVITY FOR INEXPERIENCED PROGRAMMERS IN CODING MANUALLY AND CODING WITH ACPI

| ID | Task | Tasks' Score | |
|---|---|---|---|
| | | Manually | ACPI |
| 1 | binary search | 4% | 82% |
| 2 | square root operation | 46% | 96% |
| 3 | get name of thread | 21% | 88% |
| 4 | Database connection | 6% | 67% |
| | **Overall grade** | **19%** | **83%** |
| | **Participants** | **81** | **67** |
| | **Avg TIme (seconds)** | **518** | **300** |

**State-of-the-art work**. anyCode is the latest state-of-the-art work before ours since they focus the problem on API calls generation. We contacted the author of anyCode [2] and the tool and the data used for this work are unavailable. NLP2Code [3] and NLI [19] rely on the StackOverflow database for their input and their output consists os code blocks instead of context -aware API calls. Thus, we use results provided in the anyCode paper as the latest state-of-the-art result to compare our approach with.

**Preparing Training Data** We collected the 1000 most popular (most starred in Github) Java projects that contain API calls in 6 popular Java libraries: `GWT`, `JodaTime`, `Java Development Kit (JDK)`, `android`, `Hibernate` and `Xstream`. We inherited the criteria for selecting Java projects from Software Engineering research on type suggestion from incomplete code snippets [20].

**Machine Translation and Neural Embeddings' configurations.** For machine translation, we used the Phrasal [15] toolkit and Doc2Vec [21] with their default configuration to learn from the input of method name and surrounding code to the sequence which represents for the ASTs of API calls.

In the default mode, n-grams is set to 7-grams and beam size for candidate searching is 100. For generating the Neural Embedding model, we use the default neural network configuration of the Python library `gensim.Doc2Vec` [21]. We tune the parameters of Doc2Vec and get the best configuration with dimension of vector as 100 and $max\_epoch$ as 1000. We trained both the machine translation model and the neural embedding model on a high-end computer with core-i7 processor, 32GB of RAM and an RTX 2080 graphics card with 8GB of memory.

**Preparing Evaluation Data** We extracted the data set of NL-PL-Snippets in the following manner. First, we randomly selected a subset of 100 code snippets from our corpus of 1000 Java projects. Then, we hired a team of 6 software developers with more than 3 years of experiences to replace information of API calls with natural language descriptions. They created natural language descriptions for methods in each code snippet, cross-checked them with each other and reached mutual agreement on each description. This process resulted in 100 natural language descriptions embedded in 100 code snippets for evaluation.

**Accuracy Measurement**. To compare the predicted code with the expected code, we check each case of the evaluation set to find the edit distance Leveinstein similarity [22] between predicted API calls and expected API calls. We cannot use test cases for the accuracy measurement since the API calls were called inside method declarations of large scale software projects that causes the infeasibility to run the tests for each specific method declarations.

### *B. RQ1. Evaluation of ACPI for code generation from natural language*

The **RQ1** shows us the benefit of ACPI. We compare our expected API calls to the list of predicted API calls using a well-known similarity metric called Leveinstein distance [22]. **We achieved the accuracy as 61%, which is remarkably higher than anyCode, that has the accuracy as 44% [2].** Since anyCode uses evaluation data as API calls collected since 2013 and most of them are obsolote, we couldn't use their evaluation data set. Some examples of expected APIs and predicted APIs can be shown in Table II.

*Analysis and Discussion:* We investigated our work for potential areas of improvement and directions for future work. We discovered that out of our 100 code snippets,in 31 cases the translated results completely matched the expected results. Examples of these cases are cases *21, 25, 100* of Table II. Out of these cases, we noticed that the exact method names are usually not mentioned in natural language descriptions,

which causes textual similarity approaches to fail to achieve the correct method names. For example, in task `21`, there is no word in natural language that had the same meaning as `"next"` in the implementation. This case caused challenges with mapping the method names to natural language only by text similarity. The surrounding code context provides an advantage for inferring the correct method names. For cases where the translated results do not entirely match expected result, we made some observations that will help us improve and extend our work. We summarize our investigation of 2 types of mismatch.

**Different codes with semantic similarities**. In these cases, the functionality of the translated result is the same as the expected result. In our example in Table II, there are 2 tasks that show this type of mismatch. First, as shown in case `80`, the order of 2 arguments changed in the translated result. Second, the translated result provides a string representation of the object like `20`. In this case, the developers need to modify the translated result by removing an extra API $toString()$ to get the correct code.

**Ambiguities in NL.** Currently, we use a simple strategy for detecting variables that appear in natural language by textual matching. However, a drawback of this solution is shown in case `68` of Table II. In this case, the term `"color"` has been incorrectly mapped as a variable in the natural language description, which leads to an incorrectly translated result and goes on to map this incorrect variable to its AST. We investigate this case and see that the expected method invocation is actually solved as the initialization of variable `color` in the code. We can address this by restricting the set of variables in the source code by their relationship to the API calls of the natural language descriptions.

### *C. RQ 2. Evaluation of the productivity of inexperienced developers when using ACPI*

To answer RQ 2, our object of study is the inexperienced Java programmer. We invited 148 undergraduate students in their third year to complete 4 programming tasks. In order to evaluate the usefulness of ACPI, we divided these students into 2 groups. We asked the participants of the first group to manually complete given programming tasks using the NL-PL Code Snippets provided to them without the help of ACPI. We then asked the participants of the second group, to perform the same programming tasks with the provided NL-PL Snippets using code completion suggestions from ACPI. To ensure that the participants were new to the test, we allowed one participant to do the tasks only once. For both the groups, a time limit of 10 minutes was given to complete the tasks.

The first task involved binary search which can be implemented by a function of $java.util.Collection$. The second task was also a popular math operation with an integer variable. The third task was popular in thread programming and required the programmer to return the name of the thread. The fourth task, which seemed to be the most challenging task, required the developer to create a static API call.

The result of this experiment is shown in Table III. It shows that ACPI not only improved the accuracy for inexperienced students in programming, but also reduced the average coding time. Without ACPI, the accuracy of the implementation of the 4 tasks ranged from 4% to 46%. We realize that like the observation from RQ 1, participants tend to be incorrect if the generated AST contains a complex structure and multiple elements. We see that in the binary search action for task 2, participants usually used incorrect method names or order of arguments. With ACPI suggestions, the accuracy is increases from 67% to 96%. In total, the average accuracy increased from 19% to 83%. Moreover, ACPI can also helped in reducing coding time. With the average time for completing the code at 300 seconds, ACPI significantly improved the performance of participants by 45%.

## V. Related Work

Converting natural language to code is an important research problem in Software Engineering. A well-known concept related to this problem is known as Naturalistic Programming (NP) [22]. anyCode [2] constructs a PCFG model for synthesizing API calls from natural language queries.Some researchers have worked on converting natural language to code with other types of code as in [23]–[27]. NLI [9] and NLP2Code [3] are able to generate a complex code template but it requires the developer to input certain contextual information to complete the code template.

There exists work that applies Statistical Learning and Deep Learning for natural language to code and vice versa as well. [6] generates pseudo-code from source code using Tree-to-String translation. Our work inherits this idea to provide a solution for code generation in the form of inference of AST's from from method name tokens. [7], [8] applies deep learning translation for code generation. The output of [8] is at a token level which cannot be used directly in program compiler, while [7] shows the challenge of deriving code using a supervised approach based on noisy practical data. [16] pointed out that big vocabulary in SE corpus prevents Neural Machine Translation from getting good accuracy. Word2Vec [28] and Doc2Vec [12] are two approaches for vectorizing at words level and sentence level using skip gram and bag of words model. In Software Engineering, the context of surrounding code is a good input for solving several researches problems. Nguyen et al [29] proposes api2vec, a tool for representing APIs as vectors for API mapping between different programming languages. Alon et al [14] propose code2vec, which is a model for method declaration vectorization and can be applicable for method identifier suggestion.

## VI. Threat of Validity

There are some threats of validity to our approach. First, we created the data set of code snippets combined with natural language manually after obtaining the code snippets from Github. Our data set contains 100 natural language descriptions inside the coding environment, which might be not representative. We try to alleviate this problem by selecting

data from high-quality of Github projects i.e. highest starred projects. Second, our natural language description describes the implementation correctly and completely. In reality, with some types of software documentation such as code comments, the natural language description is usually ambiguous with lack of information and is considered technical debt. Third, we use a heuristic strategy for code candidate ranking. This can be improved with more rigorous parameter tuning for different data sets. Fourth, our studies which require experienced and inexperienced developers can be improved that We can increase the number of participants.

## VII. Conclusion and Future Work

In this work, we introduced the API call Programming Interface (ACPI), an automatic tool for allowing developers to get compilable, context-aware code suggestions from natural language descriptions. We are able to do this with a 61% accuracy and experiment results show that our approach outperforms state of the art. A limitation of our work is that we haven't evaluated the cases where the natural language description is vague (e.g. ambiguous pronouns). We plan to explore how code context can help solve these cases in future works. Some other future directions include improving Neural Machine Translation models to solve big vocabulary problems like [16], formulating advanced vectorization and embedding techniques that are suitable for source code structures, extending the work to support generation of general code fragments instead of only API calls, improving the approach by analyzing possible ambiguities in natural language descriptions, and testing our system with larger datasets that may have multiple text descriptions in a single code snippet. All the code, data and results of ACPI are open source available at [30].

## References


[1] R. Article, "Deprecation of apis," https://tinyurl.com/sh75k7ht.

[2] T. Gvero and V. Kuncak, "Synthesizing java expressions from free-form queries," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 416–432. [Online]. Available: https://doi.org/10.1145/2814270.2814295

[3] B. A. Campbell and C. Treude, "Nlp2code: Code snippet content assist via natural language tasks," in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2017, pp. 628–632.

[4] Q. Shen, S. Wu, Y. Zou, Z. Zhu, and B. Xie, "From api to nli: A new interface for library reuse," *ArXiv*, vol. abs/2007.03305, 2020.

[5] P. Yin and G. Neubig, "A syntactic neural model for general-purpose code generation," *CoRR*, vol. abs/1704.01696, 2017. [Online]. Available: http://arxiv.org/abs/1704.01696

[6] Y. Oda, H. Fudaba, G. Neubig, H. Hata, S. Sakti, T. Toda, and S. Nakamura, "Learning to generate pseudo-code from source code using statistical machine translation (t)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2015, pp. 574–584.

[7] A. V. M. Barone and R. Sennrich, "A parallel corpus of python functions and documentation strings for automated code documentation and code generation," *CoRR*, vol. abs/1707.02275, 2017. [Online]. Available: http://arxiv.org/abs/1707.02275

[8] X. Gu, H. Zhang, D. Zhang, and S. Kim, "Deep api learning," 2016.

[9] M. Raghothaman, Y. Wei, and Y. Hamadi, "Swim: Synthesizing what i mean - code search and idiomatic snippet synthesis," *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pp. 357–367, 2016.

[10] "Android aopk project," https://tinyurl.com/sp6wwjan.

[11] O. Pulido-Prieto and U. Juárez-Martínez, "A survey of naturalistic programming technologies," *ACM Comput. Surv.*, vol. 50, no. 5, Sep. 2017. [Online]. Available: https://doi.org/10.1145/3109481

[12] Q. V. Le and T. Mikolov, "Distributed representations of sentences and documents," *CoRR*, vol. abs/1405.4053, 2014. [Online]. Available: http://arxiv.org/abs/1405.4053

[13] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2*, ser. NIPS'13. Red Hook, NY, USA: Curran Associates Inc., 2013, p. 3111–3119.

[14] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: learning distributed representations of code," *Proceedings of the ACM on Programming Languages*, vol. 3, pp. 1 – 29, 2019.

[15] S. Green, D. Cer, and C. D. Manning, "Phrasal: A toolkit for new directions in statistical machine translation," in *In Proceddings of the Ninth Workshop on Statistical Machine Translation*, 2014.

[16] R.-M. Karampatsis, H. Babii, R. Robbes, C. Sutton, and A. Janes, "Big Code != Big Vocabulary: Open-Vocabulary Models for Source Code," *arXiv e-prints*, p. arXiv:2003.07914, Mar. 2020.

[17] StackOverflow, "Stackoverflow," https://tinyurl.com/pmtltux, April 2020.

[18] ProgramCreek, "Programcreek," https://tinyurl.com/y93rah3r, April 2020.

[19] D. Price, E. Rilofff, J. Zachary, and B. Harvey, "Naturaljava: A natural language interface for programming in java," in *Proceedings of the 5th International Conference on Intelligent User Interfaces*, ser. IUI '00. New York, NY, USA: Association for Computing Machinery, 2000, p. 207–211. [Online]. Available: https://doi.org/10.1145/325737.325845

[20] H. Phan, H. A. Nguyen, N. M. Tran, L. H. Truong, A. T. Nguyen, and T. N. Nguyen, "Statistical learning of api fully qualified names in code snippets of online forums," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 632–642. [Online]. Available: https://doi.org/10.1145/3180155.3180230

[21] Gensim, "Gensim doc2vec library," https://tinyurl.com/y7eojef3, April 2020.

[22] dzone website, "Levenshtein distance," https://tinyurl.com/ybp4nr9t, April 2020.

[23] 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. New York, NY, USA: Association for Computing Machinery, 2016, p. 1013–1017. [Online]. Available: https://doi.org/10.1145/2950290.2983931

[24] Q. Chen, X. Wang, X. Ye, G. Durrett, and I. Dillig, "Multi-modal synthesis of regular expressions," 2019.

[25] M. Mefteh, N. Bouassida, and H. Ben-Abdallah, "Towards naturalistic programming: Mapping language-independent requirements to constrained language specifications," *Science of Computer Programming*, vol. 166, pp. 89 – 119, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167642318301941

[26] Y. Mashima, S. Hirokawa, and K. Takeuchi, "Ties between mined structural patterns in program and their identifier names," in *Integrated Uncertainty in Knowledge Modelling and Decision Making*, H. Seki, C. H. Nguyen, V.-N. Huynh, and M. Inuiguchi, Eds. Cham: Springer International Publishing, 2019, pp. 335–346.

[27] Y. Feng, R. Martins, Y. Wang, I. Dillig, and T. W. Reps, "Component-based synthesis for complex apis," *SIGPLAN Not.*, vol. 52, no. 1, p. 599–612, Jan. 2017. [Online]. Available: https://doi.org/10.1145/3093333.3009851

[28] O. system, "Article on method invocation," https://tinyurl.com/y7knpotw, April 2020.

[29] T. D. Nguyen, A. T. Nguyen, H. D. Phan, and T. N. Nguyen, "Exploring api embedding for api usages and applications," in *Proceedings of the 39th International Conference on Software Engineering*, ser. ICSE '17. IEEE Press, 2017, p. 438–449. [Online]. Available: https://doi.org/10.1109/ICSE.2017.47

[30] ACPI, "Acpi replication package," https://github.com/pdhung3012/ACPI-ASENLPWorkshop, May 2021.