

A Survey of Learning-based Method Name Prediction

HANWEI QIAN, State Key Laboratory for Novel Software Technology, Nanjing University, China and Engineering Research Center of Electronic Data Forensics Analysis, China

WEI LIU, City University of Macau, China

TINGTING XU, City University of Macau, China

JIE YIN, Engineering Research Center of Electronic Data Forensics Analysis, China

XIA FENG, Hainan University, China

WEISONG SUN*, Nanyang Technological University, Singapore and State Key Laboratory for Novel Software Technology, Nanjing University, China

ZIQI DING, University of New South Wales, Australia

YUCHEN CHEN, State Key Laboratory for Novel Software Technology, Nanjing University, China

YUN MIAO, State Key Laboratory for Novel Software Technology, Nanjing University, China

JIAXUN LI, School of Mathematical Sciences, Soochow University, China

JIANHUA ZHAO, State Key Laboratory for Novel Software Technology, Nanjing University, China

CHUNRONG FANG, State Key Laboratory for Novel Software Technology, Nanjing University, China

The choice of method names significantly influences code comprehension and maintenance, posing a considerable challenge, especially for novice developers. Automating the prediction of appropriate method names based on the method code body has emerged as a promising approach to address this challenge. In recent years, numerous machine/deep learning (ML/DL)-based method name prediction (MNP) techniques have been proposed. However, a systematic review of these techniques is currently lacking, hindering future researchers from understanding the research status, development trends, challenges, and opportunities in this field. To fill this gap, in this paper, we conduct a systematic literature review on learning-based MNP studies. Specifically, we first perform a thorough review of the literature concerning publication venue, publication year, and contribution types. This analysis enables us to discern trends in studies related to MNP. Second, we depict the

*Weisong Sun is the corresponding author.

Authors' addresses: **Hanwei Qian**, qianhanwei@smail.nju.edu.cn, State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, Jiangsu, China, 210093 and Engineering Research Center of Electronic Data Forensics Analysis, Nanjing, Jiangsu, China, 210031; **Wei Liu**, D24092110297@cityu.edu.mo, City University of Macau, Macau, Macau, China, 999078; **Tingting Xu**, D25092110136@cityu.edu.mo, City University of Macau, Macau, Macau, China, 999078; **Jie Yin**, yinjie@jspi.cn, Engineering Research Center of Electronic Data Forensics Analysis, Nanjing, Jiangsu, China, 210031; **Xia Feng**, xiafeng@hainanu.edu.cn, Hainan University, Haikou, Hainan, China, 570228; **Weisong Sun**, weisong.sun@ntu.edu.sg, Nanyang Technological University, Singapore, 50 Nanyang Avenue, Singapore, 639798 and State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, Jiangsu, China, 210093; **Ziqi Ding**, ziqi.ding1@unsw.edu.au, University of New South Wales, Sydney, New South Wales, Australia, 2052; **Yuchen Chen**, yuc.chen@smail.nju.edu.cn, State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, Jiangsu, China, 210093; **Yun Miao**, miaoyun001my@gmail.com, State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, Jiangsu, China, 210093; **Jiaxun Li**, 20224207007@stu.suda.edu.cn, School of Mathematical Sciences, Soochow University, Soochow, Jiangsu, China, 215000; **Jianhua Zhao**, zhaojh@nju.edu.cn, State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, Jiangsu, China, 210093; **Chunrong Fang**, fangchunrong@nju.edu.cn, State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, Jiangsu, China, 210093.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2024 Association for Computing Machinery.

1049-331X/2024/0-ART1 \$15.00

<https://doi.org/>

general workflow of learning-based MNP techniques, which involves three consecutive subprocesses: context extraction, context preprocessing, and context-based prediction. Subsequently, we investigate contemporary techniques/solutions applied in the three subprocesses. Third, we scrutinize the widely used experimental databases, evaluation metrics, and replication packages utilized in MNP studies. Moreover, we summarize existing empirical studies on MNP to facilitate a quick understanding of their focus and findings for subsequent researchers. Finally, based on a systematic review and summary of existing work, we outline several open challenges and opportunities in MNP that remain to be addressed in future work.

CCS Concepts: • **Software and its engineering** → **Automatic programming**.

Additional Key Words and Phrases: Method Name Prediction, Method Name Suggestion, Method Name Recommendation, Machine Learning, Deep Learning, AI and Software Engineering

ACM Reference Format:

Hanwei Qian, Wei Liu, Tingting Xu, Jie Yin, Xia Feng, Weisong Sun, Ziqi Ding, Yuchen Chen, Yun Miao, Jiaxun Li, Jianhua Zhao, and Chunrong Fang, 2024. A Survey of Learning-based Method Name Prediction. *ACM Trans. Softw. Eng. Methodol.* 0, 0, Article 1 (2024), 60 pages.

1 INTRODUCTION

In numerous programming languages, such as Java, Python, and C++, methods (also called functions) are essential building blocks of programming languages, allowing developers to encapsulate logic, promote code reuse, and enhance the organization and readability of their code. Usually, each method has a method name that serves as an identifier for specific functionalities or operations within the code. Meaningful and succinct method names are significant in software development, as they can help developers quickly grasp the key functionality of the methods [79]. What's more, choosing good names can be especially critical for methods that are part of public **Application Programming Interfaces (APIs)**, as poor method names can doom a project to irrelevance [59].

However, constructing high-quality method names is often challenging, especially for inexperienced/novice developers [39]. On the one hand, novice developers may grapple with the intricacies of code abstraction and struggle to encapsulate the essence of a method's functionality in a concise and meaningful manner. On the other hand, they might also face difficulties in adhering to established naming conventions and patterns, which are crucial for maintaining consistency across a codebase. As a result, novice developers might unintentionally introduce ambiguity or inconsistency in method names, complicating collaboration within a development team.

The art of naming methods involves not only the technical aspect of accurately representing the code's behavior but also considering the broader context of the application and its potential future modifications [4]. MNP, also known as method name suggestion, is the task of suggesting appropriate names for methods based on the context of the method code [113]. The method code is also called target method [10]. By suggesting well-crafted, meaningful, and contextually relevant method names, MNP techniques guide best practices, helping developers learn effective naming conventions over time. MNP research focuses on developing advanced techniques for automatically generating method names for method code bodies. Given a target method, MNP techniques can automatically generate an accurate method name for it.

To fulfill the MNP task, numerous researchers have sequentially proposed a plethora of techniques. The origins of MNP research can be traced back to the work of Allamanis et al. [7], who trained an n-gram language model called giga-token on a GitHub Java corpus. During their analysis using the giga-token model, they discovered that method names are more predictable compared to type and variable names. Subsequently, the success of DL technology on many **Software Engineering (SE)** tasks [34, 41], a large number of researchers have widely adopted this technology to better solve MNP tasks [8, 30, 52, 113]. DL-based MNP techniques extensively employ DL-based encoder-decoder networks to train models capable of predicting names based on the method's code body.

The encoder is responsible for transforming the method code into a context vector (also known as an embedding), while the decoder uses this context vector to generate the predicted method names. Whether it is the early ML-based MNP technique or the recent DL-based MNP technique, we collectively call it the learning-based MNP technique. Despite the growing number of learning-based MNP studies, to the best of our knowledge, no systematic review has yet been conducted. Furthermore, unlike many SE tasks such as bug detection or effort estimation, MNP is inherently a cross-modal generation task that must generate concise and semantically meaningful names from complex code contexts. It is also uniquely challenging due to its extremely short output space, subjective evaluation criteria, and the need to capture high-level semantics from low-level code signals. This hinders subsequent researchers from quickly understanding this field's progress, challenges, and research opportunities. Since the introduction of MNP, all related research has employed learning-based methods, and this trend is expected to continue in the future. Therefore, this paper focuses on learning-based MNP research.

We summarize and present the publication venue, publication year, and contribution types (including new techniques and empirical studies). Then, based on an in-depth reading and review of these works, we outline the general workflow of learning-based MNP, which completes MNP through three core and consecutive subprocesses: context extraction, context preprocessing, and context-based prediction. We further investigate and summarize the optimization techniques/solutions proposed in existing MNP papers to improve the three subprocesses. In addition, to facilitate a quick evaluation of newly proposed MNP techniques by subsequent researchers, we systematically review and present the evaluation methods employed in existing studies, including experimental datasets, performance metrics, and replication packages. Likewise, we review and summarize existing empirical studies on MNP to help subsequent researchers understand their concerns and findings. Finally, drawing upon a comprehensive examination of the existing MNP literature, we delineate challenges and opportunities for future study. We envision that our findings will play a crucial role in guiding future research inquiries and advancements in this rapidly evolving field.

The main contributions of our survey include:

- **Survey Methodology.** We adopt a systematic approach to collect relevant papers documented in peer-reviewed journals and conferences up to June 1st, 2024, as a starting point for future MNP research. We review 55 MNP studies that used ML/DL techniques in terms of publication trends, publication venues, and publication years.
- **Learning-based MNP.** We depict the general workflow of learning-based MNP. This workflow summarizes that learning-based MNP techniques typically achieve the generation of the predicted method names through the following three consecutive subprocesses: context extraction, context preprocessing, and context-based prediction. We further systematically examine and discuss 55 learning-based MNP techniques, employing a multidimensional analysis framework to investigate innovations across the three subprocesses.
- **Dataset, Metric, Replication Packages.** We compile commonly used 19 experimental datasets, 11 performance metrics, and 35 replication packages in MNP research.
- **Empirical studies.** We review and summarize existing empirical studies on MNP to help future researchers understand the concerns and findings of pioneering researchers.
- **Outlook and challenges.** We underscore challenges and opportunities in the learning-based MNP, drawing from our findings with the aim of fostering continued exploration in this field. Supplementary resources, datasets, and code related to our research are available at <https://github.com/software-theorem/MNPSurvey>.

Structure of the paper: The subsequent sections of this paper are structured as follows. Section 2 introduces the background information concerning the ML and DL techniques adopted in the MNP

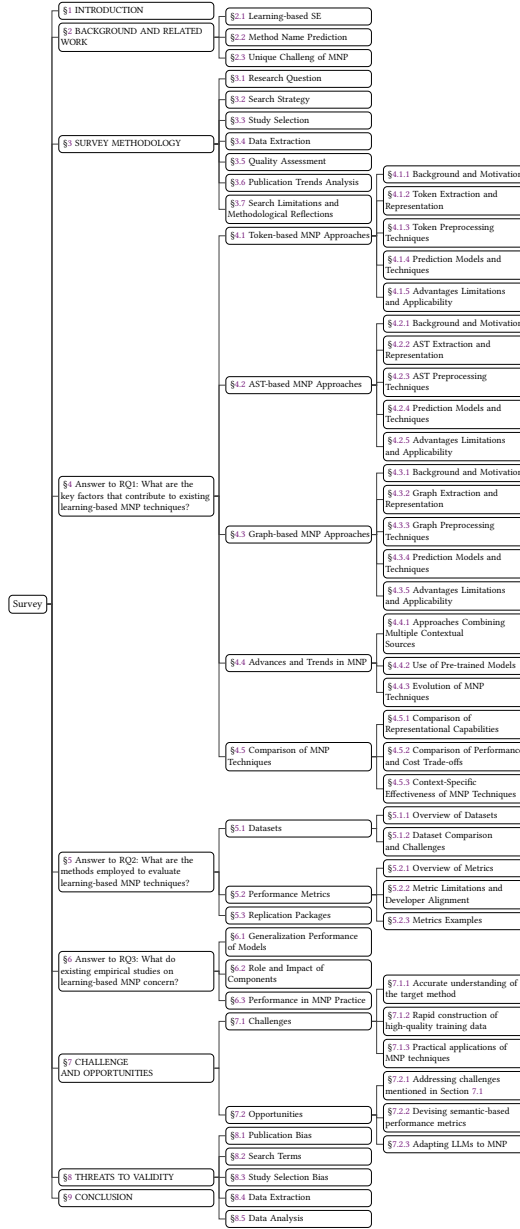


Fig. 1. Structure of the paper

task. In Section 3, we present the survey methodology about how we collect relevant papers from several databases. Sections 4 – 6 provide a comprehensive summary of the primary research questions addressed within this study, along with corresponding findings. Section 7 thoroughly examines the challenges and prospects that lie on the horizon for the field of MNP studies, shedding light on potential opportunities for future research. The threats to validity of our survey are

discussed in Section 8. Finally, Section 9 provides a conclusion of this survey. The detailed structural information of this paper is presented in Fig. 1.

2 BACKGROUND AND RELATED WORK

2.1 Learning-based SE

Due to the naturalness of software [38], driven by the accumulation of vast amounts of code in open-source repositories, the SE community leverages ML techniques to identify intriguing patterns and unique relationships within code data. This aims to automate or enhance many SE tasks typically performed by developers. Through meticulous feature engineering, researchers aim to identify significant data attributes capable of solving specific problems or automating particular tasks. As computational power advances and modern computer architectures increase available memory, ML methods have evolved to incorporate DL approaches. Enabled by vast repositories of available software code and rapid hardware advancements, DL techniques have made significant strides in various tasks within SE research. Many SE problems can naturally be framed as learning-based data analysis tasks, encompassing classification tasks that aim to categorize data instances into predefined classes, ranking tasks aimed at generating a ranking within a dataset, regression tasks used to assign real values to data instances, and generation tasks that aim to produce concise natural language descriptions. Bug detection [37, 42, 126] can be seen as a classification task for predicting whether a code segment contains bugs. Code search [85, 111] and bug localization [57, 58] in SE are ranking tasks. SE researchers also employ regression models to estimate the effort required for developing software systems [1, 49] and bug-fixing times [21, 36], among others, which fall under time-series forecasting problems. Code summarization [41, 47, 86, 90] is typically considered a generation task, aiming to extract crucial information from code and generate advanced natural language descriptions to aid programmers in understanding the functionality of the code.

Many studies have observed the trend of applying learning-based methods in SE, summarized and synthesized existing research, and conducted an in-depth analysis of future developments [20, 61, 80, 102, 108, 114, 125]. For example, Wang et al. [102] review the literature on ML/DL in SE from 2009 to 2023, analyzing trends, complexity, and reproducibility challenges. It examines differences in ML/DL approaches to SE tasks and identifies key factors influencing model selection. Surveys on learning-based SE typically cover a wide range of topics but may not delve deeply into the specifics of each task. Surveys focused on specific SE tasks, on the other hand, provide more in-depth analysis and insights, serving as a valuable complement. Sun et al. [87] surveys code search techniques, focusing on query-end, code-end, which predominantly uses DL-based code representations, and match-end optimization. They provide a detailed analysis of each dimension and highlight open challenges and future research opportunities in code search. Zhang et al. [122] surveys learning-based **Automated Program Repair (APR)** techniques, highlighting their use of DL to fix bugs through neural machine translation. It reviews the APR workflow, key components, datasets, and evaluation metrics, and discusses practical guidelines and open issues for future research and application. Omri et al. [62] review various software fault prediction approaches, highlighting how recent DL algorithms address the limitations of traditional feature-based methods by capturing semantic and structural information for more accurate predictions.

The emerging application of DL in SE shows significant potential for executing various traditional software development tasks. The current and future trend is to leverage learning-based methods to address SE challenges. Traditionally, MNP has relied on developers' experience for manual naming, but the advent of learning-based methods has made it possible to automate the suggestion and verification of method names.

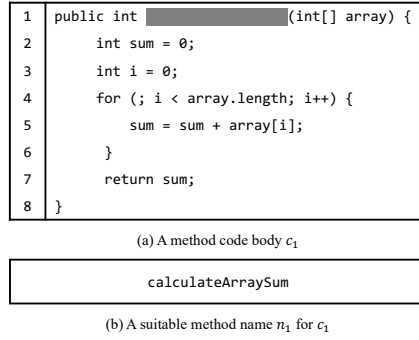


Fig. 2. Example of MNP

2.2 MNP

MNP is tasked with recommending a proper method name from a source code snippet or stripped binary code according to the method's context (e.g., its internal implementation, known as local context, and its surrounding codebase, known as global context), which will be discussed in detail in Section 4.4.1. Fig. 2 illustrates an example. The target method code snippet c_1 presented in Fig. 2(a) is provided by the developer. The `calculateArraySum` in Fig. 2(b) is a suitable method name that fulfills the developer's requirement.

The tokens composing the identifier names in the contexts appear together regularly and naturally due to the developers' intention to realize the method's functionality. Specifically, developers tend to combine related identifier name tokens when writing code to achieve specific method functionalities, such as in the method `getHostAddress` used for retrieving the host address, which in code implementation might be a process to *get* the *local* Ethernet listening *address*. An abstract, concise method name captures this functionality. Therefore, the occurrence of tokens in identifier names within the context may influence the tokens in method names. Nguyen et al. [59] collect 2,127,355 files from GitHub, including 17,012,754 methods. Their empirical study shows that for a method, 65.0% of the tokens in its name also exist in the tokens of identifier names in its context.

As a result, statistical models can be established using ML, DL, and similar methods to explore the relationship between method names and their context (including the method body). From a ML perspective, the primary objective of these models is to learn the latent relationship between method context and method names, formalized as $P(L_{name}|C_{context})$, where $C_{context}$ represents the context of the method, and L_{name} name corresponds to the target name of the method. These models derive a probability distribution model P from training data. Once trained, the models can predict the most probable method name given a specific method body context. MNP can be regarded as a generation task. Its objective is to generate an appropriate method name as output based on a given code snippet or context. In practice, MNP typically uses a fixed vocabulary as the target space for token prediction, thereby categorizing it as a form of classification task. It is generally believed that the closer the tokens predicted by a model are to the tokens of the actual method name, the higher the precision and recall, and the more valuable the predicted results.

2.3 Unique Challenges of MNP

Unlike classification or ranking tasks in SE, such as bug detection, where models produce discrete labels or scores based on relatively objective ground truths, MNP poses fundamentally different challenges. The goal in MNP is to transform a large, structured, and often noisy code context into a

concise, semantically meaningful method name, a process that bridges the gap between machine understanding of code and human language conventions. A key preprocessing step unique to MNP is token splitting, where compound identifiers, such as `getSourceCode`, are tokenized into subwords, including `get`, `source`, and `code`. This step forms the basis for constructing a candidate vocabulary and is often a prerequisite for framing MNP as a classification problem. In contrast, such preprocessing is typically unnecessary in tasks like defect detection or code clone identification.

Moreover, effective MNP requires context modeling that goes far beyond the method body. Crucial naming cues are often embedded in the surrounding codebase, including class names, superclasses, implemented interfaces, sibling methods, and even documentation comments. These elements reflect domain-specific terminology and naming conventions, which are often implicit and vary across programming languages, teams, and projects. By contrast, many other SE tasks rely more heavily on syntactic correctness or code behavior, with less dependence on such stylistic or linguistic features. To deeply understand the function and semantics of the target method, MNP models must be capable of capturing not only syntactic structures and data flows but also higher-level semantic concepts. This requires context encoders that go beyond structural parsing and incorporate abstract behavioral patterns, essential for bridging the semantic gap between code and natural language.

Finally, MNP presents unique challenges in evaluation. Unlike bug detection, where outcomes are binary and objectively verifiable (a bug either exists or it does not), MNP is inherently subjective. For a given method, multiple developers may produce semantically valid yet stylistically distinct names. Naming practices are shaped by unspoken conventions, such as the use of `get/set` prefixes or verb-noun structures that differ across ecosystems. This subjectivity complicates the design of consistent and fair evaluation metrics. Token-overlap measures like Precision, Recall, and **F1-score (F1)** fail to capture semantically equivalent variants, while n-gram based metrics like **Bilingual Evaluation Understudy (BLEU)** or **Recall-Oriented Understudy for Gisting Evaluation (ROUGE)**, although standard in natural language generation, are unstable and often unreliable for the short output space characteristic of method names.

For the MNP distinguishes itself from other SE tasks, this paper focuses on the research of learning-based MNP tasks, examining the technologies, methods, models, datasets, and metrics used in various studies. It explores influential techniques and methods, analyzes the current challenges in the field, and discusses potential future trends and directions.

3 SURVEY METHODOLOGY

For a systematic review of MNP research, we follow the guidelines provided by Kitchenham and Charters [83] and Petersen et al. [67] to design the review protocol, which includes research questions, search strategies, study selection, data extraction, and **Quality assessment (QA)**. Twelve researchers are involved in this study, all with experience in SE, with most having research experience in MNP. Seven graduate students collaborate throughout the study, responsible for paper collection, data extraction, and analysis, all under the guidance and supervision of experts in SE research. Additionally, five senior researchers contribute significantly by formulating and discussing research questions, addressing biases, and improving documentation for this study.

3.1 Research Question

We aim to summarize, categorize, and analyze empirical evidence from various published studies on learning-based MNP. To achieve this, we address three research questions (RQs):

- **RQ1.** *What are the key factors that contribute to existing learning-based MNP techniques?* This RQ aims to depict the general workflow of learning-based MNP and explore the key optimization techniques proposed in learning-based MNP studies.
- **RQ2.** *What are the methods employed to evaluate learning-based MNP techniques?* This RQ covers three fundamental areas of technique evaluation: databases, performance evaluation metrics, and replication packages.
- **RQ3.** *What do existing empirical studies on learning-based MNP concern?* This RQ discusses existing empirical studies that focus on evaluating learning-based MNP techniques.

By analyzing these RQs, we are able to find challenges and opportunities for learning-based MNP research. These findings are the foundation for creating a research roadmap for future endeavors.

3.2 Search Strategy

During the manual search phase, we select papers published since June 2020 from widely recognized and highly reputable venues in the SE field, such as the **International Conference on Software Engineering (ICSE)** and **ACM Transactions on Software Engineering and Methodology (TOSEM)**.

We meticulously review the initially collected papers' titles, abstracts, and keywords to identify additional keywords and phrases. Subsequently, through brainstorming sessions, we expand and refine the list of search strings, incorporating relevant terms, synonyms, and variations. This iterative process enables us to continually improve the search keyword list based on search outcomes, ensuring accurate capture of literature pertinent to MNP.

The final keywords used for our automated search include ("**method**" OR "**function**" OR "**identifier**") AND ("**name**") AND ("**suggestion**" OR "**recommendation**" OR "**prediction**"). Subsequently, we conduct a total of $3 \times 1 \times 3 \times 4 = 36$ searches across four popular electronic databases in SE: ACM Digital Library, IEEE Xplore Digital Library, Scopus Digital Library, and Google Scholar. Following this, the first two authors manually inspect each paper to determine its relevance to our research scope. To quantify the consistency of our inclusion decisions, we computed Cohen's κ on a random subset of 50 papers. We obtained $\kappa = 0.84$, indicating "almost perfect" agreement between the two authors [18]. Disagreements were resolved by discussion until full consensus was reached.

The search was conducted on June 1st, 2024, encompassing studies published until that date. After completing automated searches on four electronic databases, 281 relevant studies were obtained, as shown in Table 1. Following the removal of duplicate findings, 196 studies on MNP are retained. Fig. 3 illustrates the systematic methodology employed for collecting papers in the survey. In the initial stage of the process, the papers are categorized into 9 groups based on several criteria, including method name suggestion, function name suggestion, and other related categories. A manual review of the initial 500 records for each group is conducted to ensure the inclusion of relevant papers for further analysis. Subsequently, the papers are sourced from four major digital libraries. The following digital libraries are consulted: the ACM Digital Library (42 papers), the IEEE Xplore Digital Library (59 papers), the Scopus Digital Library (25 papers), and Google Scholar (155 papers). An automated search identifies 281 papers, which are then filtered to remove duplicates, yielding 196 papers. The selection process continues with the application of filters to the titles and abstracts of the papers, resulting in 51 papers. These are then subjected to a detailed full-text review, which narrows the number down to 34 papers. The snowballing technique is employed to identify additional relevant papers through the references of the selected papers, expanding the pool to 1,660 papers. During this phase, citations that are not initially identified are incorporated. A

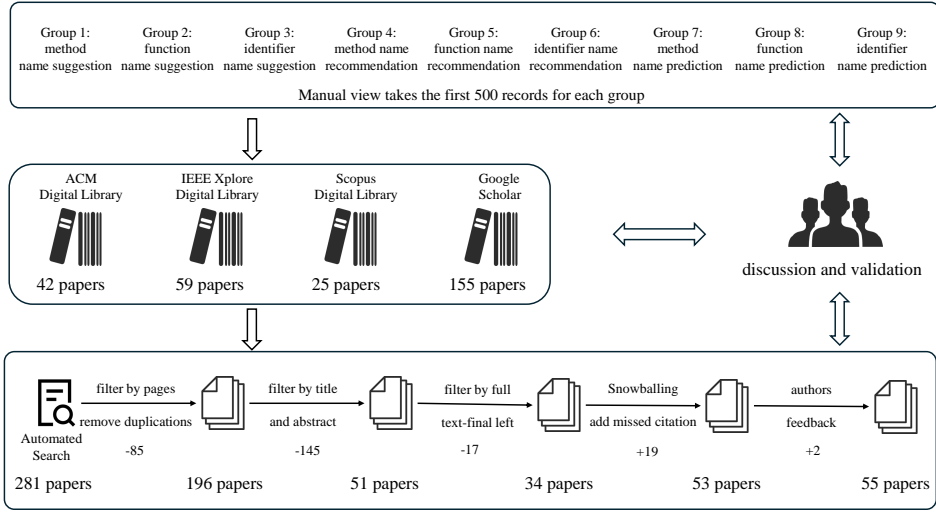


Fig. 3. General workflow of the paper collection

Table 1. The results of the retrieval and screening of papers related to MNP

Process	Studies
ACM Digital Library	42
IEEE Xplore Digital Library	59
Scopus Digital Library	25
Google Scholar	155
Automatic search from four electronic databases.	281
Removing duplicated studies.	196
Excluding primary studies based on title and abstract.	51
Excluding primary studies based on full text - final left	34
Adding missed citations after snowballing	53
After collecting authors feedback	55

comprehensive and meticulous discussion and selection process ultimately results in the inclusion of 53 papers that meet all predefined criteria.

3.3 Study Selection

The preliminary selection process of the study involved summarizing all research published in SE related to MNP. We select the top 500 search results from each database and screen suitable studies by reviewing each study's title, keywords, and abstract. Subsequently, these studies were further evaluated based on the detailed inclusion and exclusion criteria outlined in Table 2. Inclusion criterion I4 about minimum quality threshold will be discussed in detail in Section 3.5. Each study was evaluated independently by at least two authors. In cases of disagreement, senior researchers facilitated discussions to achieve consensus.

To ensure comprehensive coverage of the field in our investigation, we employ a snowballing approach to gather further papers that our keyword searches might have missed. The snowballing method identifies citation-dependent papers to expand our paper collection. We utilize both

Table 2. The inclusion and exclusion criteria

ID	Inclusion criteria
I1	The paper must be written in English.
I2	The paper must involve at least one ML/DL technique.
I3	The paper must be a peer-reviewed published full paper.
I4	The paper must pass the minimum quality threshold.
ID	Exclusion criteria
E1	The paper is a form of grey literature, e.g., a technical report or dissertation.
E2	The paper is explicitly a short paper, position paper, or editorial.
E3	After being extended into journal versions, conference papers are excluded.

Table 3. Data extraction form

Item	Description	Association
Title	The title of the literature.	Publication Trend
Publication year	The publication year of the literature.	Publication Trend
Publication type	The type of the literature, e.g., new technique and empirical study.	Publication Trend
Prediction approach	Methods/techniques used for context extraction, context preprocessing, context classification, context encoding/decoding	RQ1
Datasets	Focus on the dataset's type, language, scale, and source	RQ2
Performance metrics	Measurement for evaluating a technique.	RQ2
Replication package	Focus on availability, dependencies, and documentation.	RQ2
Programming languages	The programming languages used for performing MNP tasks	RQ3
Research questions	The key inquiries or investigation objectives posed in research	RQ3

backward and forward snowballing techniques. In the backward snowballing, we scrutinize the references of each collected paper and identify papers within our scope. In the forward snowballing, we use Google Scholar to find papers of interest that cited already collected papers. We iteratively repeat the snowballing process until no new relevant papers are added, reaching a stable state. Through this process, we retrieve an additional 19 papers.

To further ensure the comprehensiveness and accuracy of our survey, we reach out to the authors of the 53 papers we collect. We provide them with our manuscript and ask them to verify the accuracy of our descriptions of their work. This allows us to understand their contributions better and make necessary revisions to our descriptions. Additionally, during these communications, the authors bring to our attention 3 papers that were initially not included in our collection. Among the suggested papers, 2 meet our inclusion criteria and are deemed relevant to our investigation. These 2 papers were subsequently added to our literature database, further enhancing our coverage of the literature on MNP and ultimately bringing the total to 55 papers.

3.4 Data Extraction

The purpose of data extraction is to collect data items that can address the research questions. Our data collection primarily focuses on publication information, study background, technical details, and experimental settings, as detailed in Table 3 of this paper. The data extraction process is divided among the seven authors of this study, with each paper being independently extracted by at least two authors and reviewed by their principal supervisor. Data items relevant to answering the three research questions are meticulously recorded, along with any information that aids in analyzing MNP techniques. After completing data extraction, we compare the extracted data and assess their consistency using Cohen's kappa coefficient (κ) [18]. If the two authors achieve a high

Table 4. Checklist of questions to assess the quality of studies

ID	QA criteria
QA1	Does the predicted method names generated through code snippets?
QA2	Does the paper describe evaluation metrics?
QA3	Does the paper provide analysis after the performance evaluation?
QA4	Is the contribution of the research clearly stated?
QA5	Does the paper provide a publicly accessible link to the code?
QA6	Are all the links provided in the paper available?
QA7	Is the raw dataset retrieved from open source?
QA8	Are all the necessary components or environments for replication provided?
QA9	Does the paper provide sufficient detail to enable replication?
QA10	Does the paper provide all necessary details for reproducibility?

agreement in extracting methods for predicting methods, their data extraction is considered reliable and consistent. If the coefficient score is low, relevant concepts and definitions are further discussed or adjusted until a consensus is reached. Other senior researchers randomly check the extraction results to mitigate bias. We also assessed agreement on all extracted data items. On a sample of 20 studies, Cohen's κ was 0.78, reflecting "substantial" agreement. Any remaining discrepancies were adjudicated by the principal supervisor to ensure consistency.

3.5 Quality Assessment (QA)

QA plays a role in selecting studies and interpreting results in systematic literature reviews [45]. By providing broader inclusion and exclusion criteria, QA helps to screen and identify appropriate studies. These assessment criteria are formulated following one of the most widely used QA tools proposed by Dybå and Dingsøyr [23]. According to their approach, the assessment of research quality primarily focuses on rigor, credibility, and relevance. We develop four QA criteria, detailed in Table 4 as QA1 to QA4.

Detailed QA information is crucial for data synthesis and result interpretation. In SE, replicability and reproducibility are essential for determining the quality of original studies and testing their reliability. We design a quality checklist with QA5 to QA10 to generate quality data. This data is then used as evidence to support parts of the answers to RQ1 and RQ2. In Section 5, we provide a more detailed discussion of the answers to the checklist questions and their analysis.

Overall, these ten criteria provide a metric to assess whether the findings of a specific study can contribute valuable insights to the MNP field. Meeting the requirements of criteria QA1, QA2, QA3, and QA4 indicates passing the minimum quality threshold. In the QA process, each study is independently evaluated by two authors based on ten criteria and undergoes validation.

3.6 Publication Trends Analysis

We analyze the publication information from the 55 MNP studies retrieved from Section 3 and discuss the main emerging publication trends.

Fig. 4 illustrates the yearly publication count for research papers. Fig. 5 shows the cumulative number of publications for the corresponding year. As displayed in Fig. 4 and 5, it is observed that the first MNP study was published in 2013. The studies on MNP gradually gained popularity after 2019, peaking in 2021. The papers published between 2019 and 2023 represent 87% of the total.

The 55 reviewed papers are published in different conferences and journals. From Fig. 6(a), it is observed that 80% of the papers are published in conference proceedings and 20% in journals.

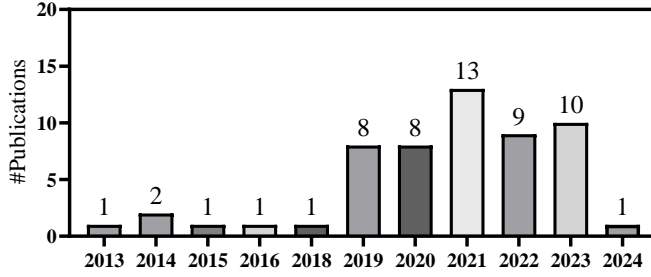


Fig. 4. Number of publications per year

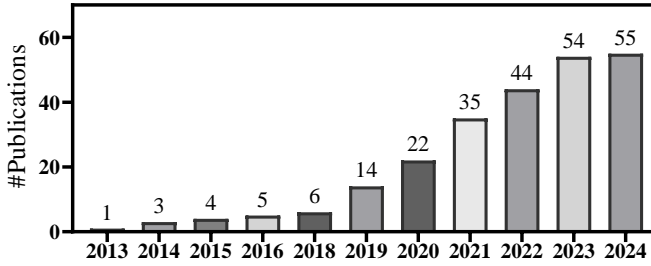


Fig. 5. Cumulative number of publications per year

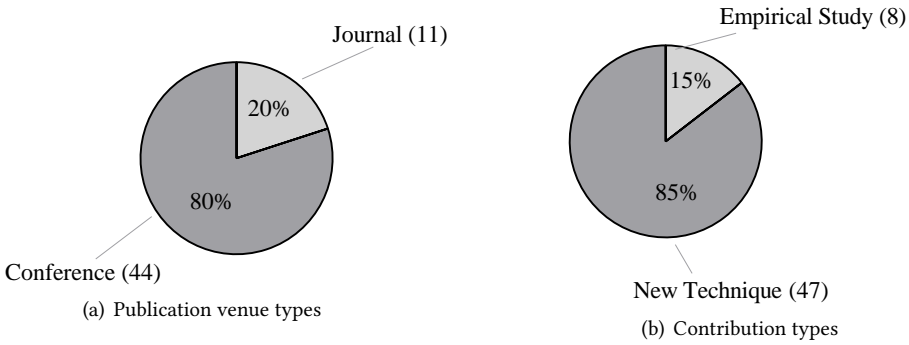


Fig. 6. Publication venues and contribution types

Additionally, Fig. 6(b) shows that out of the 55 articles, 47 belong to the category of **New Techniques (NT)**, 8 are categorized as **Empirical Studies (ES)** [51].

Table 5 lists the detailed publication venues of reviewed studies, covering various types of studies. Among the 32 publication venues, the most popular conferences for publishing these papers are ICSE, ICPC, MSR, PLDI, PACMPL, and ICLR. ICSE has the most significant number of presented papers among these.

Table 5. Publication venues for MNP studies (NT: New Technique; ES: Empirical Study; TS: Total Studies)

Short Name	Full Name	NT	ES	TS
MSR	International Conference on Mining Software Repositories	4	0	4
ICPC	International Conference on Program Comprehension	5	0	5
PLDI	ACM-SIGPLAN Symposium on Programming Language Design and Implementation	3	0	3
ESEC/FSE	ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering	2	0	2
ICML	International Conference on Machine Learning	1	0	1
PACMPL	Proceedings of the ACM on Programming Languages	3	0	3
ICLR	International Conference on Learning Representations	3	0	3
ISSTA	International Symposium on Software Testing and Analysis	0	1	1
APSEC	Asia-Pacific Software Engineering Conference	1	0	1
PEPM	ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation	1	0	1
ICSE	International Conference on Software Engineering	7	0	7
SANER	IEEE International Conference on Software Analysis, Evolution, and Reengineering	2	0	2
STC	Annual Software Technology Conference	0	1	1
RL+SE&PL	International Workshop on Representation Learning for Software Engineering and Program Languages	0	1	1
QUATIC	Quality of Information and Communications Technology	0	1	1
COLING	International Conference on Computational Linguistics	1	0	1
ASE	International Conference on Automated Software Engineering	1	0	1
JSS	Journal of Systems and Software	1	0	1
PAKDD	Pacific-Asia Conference on Knowledge Discovery and Data Mining	1	0	1
NeurIPS	Neural Information Processing Systems	1	0	1
Inf Softw Technol	Information and Software Technology	0	1	1
AAAI	Association for the Advancement of Artificial Intelligence	1	0	1
ICQ	International Conference on Code Quality	1	0	1
ISSRE	International Symposium on Software Reliability Engineering	1	0	1
Empir. Softw. Eng.	Empirical Software Engineering	1	0	1
COLA	Journal of Computer Languages	1	0	1
TOSEM	ACM Transactions on Software Engineering and Methodology	2	0	2
SSE	IEEE International Conference on Software Services Engineering	0	1	1
QRS	International Conference on Software Quality, Reliability and Security	1	1	2
IJCSIT	International Journal of Computer Science and Information Technologies	0	1	1
IJCNN	International Joint Conference on Neural Networks	1	0	1
TSE	Transactions on Software Engineering	1	0	1

3.7 Search Limitations and Methodological Reflections

While constructing our corpus for this survey on MNP, we initially employed a carefully designed set of keywords to retrieve relevant papers from major academic databases. However, through snowballing techniques, including both backward and forward citation tracing, we identified a set of relevant papers that were missed during the initial keyword-based search. This prompted a closer analysis of why these papers were not captured and how future survey efforts can be improved.

Our investigation revealed several key factors contributing to the omissions:

- **Task Overlap and Implicit Relevance:** Many papers in adjacent or overlapping tasks, such as code summarization, code completion, code representation, and source code processing, contain relevant insights or evaluations on MNP, yet do not explicitly frame their contributions in this context. In some cases, MNP appears only as an auxiliary evaluation task rather than the main focus, making these papers less discoverable through direct keyword queries.
- **Terminology Evolution:** The terminology around MNP has evolved. In earlier work, the task was sometimes referred to using alternative terms such as extreme code summarization. These older terms were not part of our initial keyword set, leading to several important but lexically divergent papers being missed.
- **Lack of Explicit Keywords:** Several relevant papers did not mention "method name prediction" or similar terms in the title, abstract, or metadata, despite proposing models or datasets directly applicable to the task. For instance, papers focused on code representation often contribute

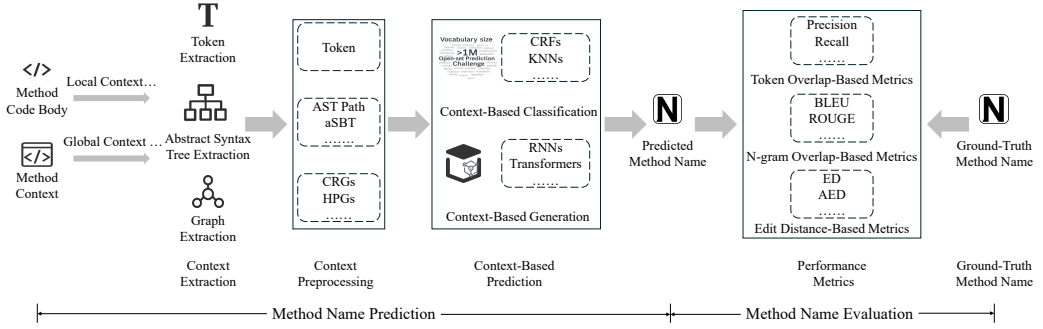


Fig. 7. Workflow of learning-based MNP

methods that are later adopted or benchmarked for MNP, even if the task is not explicitly discussed.

- **Hierarchical Task Relationships:** Notably, MNP can be seen as a specialized sub-task of code completion. Some papers addressing code completion cover method names implicitly but at a coarser granularity, and thus avoid using our target keywords. This hierarchical relationship further complicates clean keyword separation.

To address these challenges, we employed a citation-based snowballing strategy starting from a carefully selected set of seed papers. This allowed us to uncover hidden yet relevant works, particularly those buried in broader or differently labeled research contexts.

For future surveys in this and similar areas, it is recommended to adopt hybrid search strategies that combine keyword-based searches with citation tracing to enhance coverage and reduce reliance on specific terminologies. Researchers should remain attentive to task redefinitions and the evolving language, particularly in rapidly developing fields. Additionally, incorporating task-taxonomy analysis can help map related tasks and sub-tasks, potentially uncovering relevant literature that may not be explicitly named. This approach underscores the importance of flexible and comprehensive methods when navigating overlapping research areas and shifting terminologies.

Summary ► MNP gradually gained attention in SE research literature in 2019, with its popularity peaking in 2021. Of the 55 papers, 44 (representing 80% of the total) were published in conferences, while the remaining ones are in journals. Additionally, 47 papers, accounting for 85% of the total, focus on new MNP techniques. ◀

4 ANSWER TO RQ1: WHAT ARE THE KEY FACTORS THAT CONTRIBUTE TO EXISTING LEARNING-BASED MNP TECHNIQUES?

The task of MNP plays a critical role in enhancing code readability, maintainability, and overall development efficiency. In recent years, data-driven approaches, particularly those based on ML and DL, have shown remarkable progress and promising potential. Fig. 7 illustrates the overall workflow of learning-based MNP, which consists of two main phases: method name prediction and method name evaluation.

In the prediction phase, the process begins with extracting both local and global semantic contexts from the method body and its surrounding code. Three primary forms of representation are employed: token sequences, **Abstract Syntax Tree (AST)**, and graph-based structures. Token-based extraction captures lexical semantics; AST-based approaches, such as AST paths and **Advanced Structure-Based Traversal (aSBT)**, preserve syntactic information; and graph-based

representations, such as **Code Relation Graphs (CRGs)** and **Heterogeneous Program Graphs (HPGs)**, model structural and semantic dependencies within code. Based on these representations, models predict method names using either classification-based or generation-based approaches. The former includes techniques such as **Conditional Random Fields (CRFs)** and **K-Nearest Neighbors (KNNs)**, while the latter utilizes sequence generation models including **Recurrent Neural Networks (RNNs)** and Transformer architectures.

In the evaluation phase, the predicted method names are assessed against ground-truth references using a variety of metrics. These include token overlap-based metrics such as Precision and Recall, n-gram overlap metrics such as BLEU and ROUGE, and edit distance-based metrics such as **Edit Distance (ED)** and **Average Edit Distance (AED)**. Each metric provides a distinct perspective on prediction quality, encompassing both lexical accuracy and linguistic fidelity.

To support a structured comparison of existing methods while offering insights into the distinctive aspects of the MNP task, we organize the literature into three main categories: token-based approaches (Section 4.1), AST-based approaches (Section 4.2), and graph-based approaches (Section 4.3). Within each category, we analyze representative methods in terms of how they extract and preprocess context, encode structural and semantic information, and ultimately perform prediction through classification or generation. This unified, process-oriented framework facilitates direct comparison of design motivations, technical strengths, and inherent limitations across approaches.

Building on this foundation, we further examine recent advances and emerging trends (Section 4.4), such as the integration of multi-source context and the application of pre-trained language models, which are significantly reshaping the MNP landscape. Finally, we present a comparative analysis of the reviewed methods (Section 4.5), highlighting their relative performance, application scenarios, and future research directions.

4.1 Token-based MNP Approaches

Token-based approaches are among the earliest technical paradigms applied to the MNP task, treating source code primarily as a sequence or set of lexical units, known as tokens.

4.1.1 Background and Motivation. Token-based approaches operate on the principle of decomposing source code, particularly method bodies, into individual lexical units such as identifiers, keywords, operators, and literals. These approaches aim to learn patterns from the sequence or set of such tokens to predict method names that align with the method's intended functionality. The core assumption is that method names are often composed of, or strongly suggested by, tokens that appear frequently within the method body or in semantically important positions (e.g., near return statements or within key API arguments), especially verbs and noun phrases.

When implementing specific functionality, developers tend to select and combine relevant identifier tokens. A well-chosen method name typically reflects this functionality abstractly and concisely. The motivation behind token-based approaches includes leveraging successful techniques from **Natural Language Processing (NLP)** by treating code as a specialized form of language [72], as well as benefiting from the simplicity and interpretability of implementation compared to structurally intensive methods. Empirical evidence from Nguyen et al. [59] supports this rationale, showing that 65.0% of the sub-tokens in a method name also appear in the surrounding identifier tokens within its context, which is also mentioned in Section 2.2.

4.1.2 Token Extraction and Representation. In token-based approaches, context extraction primarily targets lexical units from various code components. The main source is the local context, particularly the method body, which provides tokens such as identifiers like variable names, method calls, and parameters, alongside keywords and selected literals [4, 52, 59, 71, 77, 92, 104, 104, 115, 115, 127].

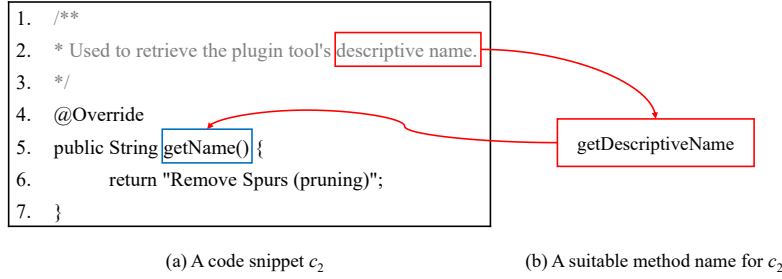


Fig. 8. An example where documentation can facilitate MNP

Tokens from the method signature, including parameter types, names, and return types, are also commonly used [50, 59].

To enrich contextual representation, some studies incorporate tokens from the global context, such as the enclosing class name and its attributes [4, 28, 59, 104, 115, 120], names or content of callers and callees [104], sibling methods within the same class [50]. Furthermore, document context, such as natural language comments or Javadoc summaries, is treated as a valuable source of tokens. The work [52] provides a definition of documentation context, that is, the first sentence of the code documentation is informative, and many code summarization approaches use it as a code summary [41, 47, 86]. Fig. 8 shows an example of using the documentation to facilitate MNP. It is observed that the body code of the method c_2 cannot provide enough information to suggest a suitable name, while the documentation of c_2 (lines 1–3 of Fig. 8(a)) can provide a useful indication for predicting the method name `getDescriptiveName` shown in Fig. 8(b).

Token sequences are typically constructed in their original order to preserve contextual relationships. A key preprocessing step is subtokenization of compound identifiers based on naming conventions. For example, `getMethodName` is split into `get`, `Method`, and `Name`, while `calculate_array_sum` becomes `calculate`, `array`, and `sum`. Lowercasing all tokens or subtokens is also a common practice to reduce vocabulary size and unify representation [59, 104, 115, 127].

4.1.3 Token Preprocessing Techniques. To prepare raw token sequences for model learning, common preprocessing steps are applied to improve data quality. A key step is token concatenation. When tokens are extracted from multiple sources such as the method body, parameters, class name, and comments, they are merged into a single input sequence. Special separators such as periods or dedicated tokens are often used to indicate source boundaries. For example, Nguyen et al. [59] concatenate tokens from the **Implementation (IMP)**, **Interface (INF)**, and **Enclosing (ENC)** contexts using periods.

Additional NLP preprocessing techniques are also widely adopted. Stop-word removal filters out frequent programming tokens such as `public` and `void` that provide limited semantic value for MNP tasks. The stop-word list must be carefully designed to avoid discarding informative tokens. Stemming or lemmatization reduces tokens to their root forms, helping unify semantically related variants, but must be applied cautiously in code contexts.

Preprocessing also includes vocabulary construction and **Out-of-Vocabulary (OOV)** handling, where high-frequency tokens are retained and low-frequency ones are replaced with `<UNK>`. Subtokenization remains the primary strategy to mitigate OOV issues. Lastly, sequence padding and truncation are used to conform to fixed input length requirements in neural models. For instance, AUMENA [127] limits input sequences to a maximum of 512 tokens.

4.1.4 Prediction Models and Techniques. For the token-based method MNP, prediction techniques can be broadly categorized into classification and generation approaches.

In classification-based approaches, the goal is to associate a token context representation with a predefined name or subtoken category. This typically involves two components: an encoder that transforms the input token sequence into a vector representation, and a classifier that predicts the name based on this representation. Early methods relied on statistical models such as N-grams [77, 92], followed by neural models including the **Logbilinear Context Model (LCM)**, **Subtoken Context Model (SCM)** [4], and the Copy Convolutional Attentional Model [6], which integrates **Convolutional Neural Networks (CNNs)**, attention mechanisms, and copy operations. In [53], Liu et al. treat method bodies as documents to learn embeddings, while ensemble classifiers like Random Forest [120] have also been explored. More general DL models—such as RNNs, **Long Short-Term Memory (LSTM)**, **Gated Recurrent Units (GRUs)**, and Transformers—are widely adopted for encoding. The classifier stage includes techniques like CRFs [9], normalized dot product with Softmax (e.g., used in code2vec) [4, 10, 13, 14, 84], **Support Vector Machines (SVMs)** via the WEKA toolkit [19], and **Multi-Layer Perceptrons (MLPs)** [12]. Other methods apply KNN [65], conditional probability estimation, ranked-list recommendations, or cross-entropy-based classification [7, 53, 54, 77, 92, 120].

In generation-based approaches, MNP is treated as a sequence-to-sequence learning task, where the model generates the method name as a sequence of subtokens. The architecture consists of an encoder to process the input and a decoder to generate the output. Common encoders include RNNs, LSTMs, and GRUs [31, 59, 71, 104, 115], which capture sequential dependencies. Transformer encoders [52, 128] have demonstrated strong capabilities in modeling long-range dependencies through self-attention. More recent methods incorporate large pre-trained code models such as CodeBERT and CodeT5 [27, 106], which provide rich contextual representations. Some studies also explore **Sequence Graph Neural Networks (Sequence GNNs)** [28, 33] to integrate structural information. On the decoding side, both RNN/LSTM/GRU-based decoders [31, 59, 82, 104, 110] and Transformer decoders [3, 32, 66, 119, 128] are commonly used, often with attention mechanisms to focus on relevant parts of the input. The CodeT5 decoder [106], as part of a full encoder-decoder architecture, is particularly suited for generation tasks. To address the OOV problem, many models incorporate copy mechanisms or pointer networks [28, 128], allowing direct copying of tokens from the input to the output. In some cases, simpler decoders [68] or fully connected layers [109] are employed for generation.

4.1.5 Advantages, Limitations, and Applicability. Token-based approaches offer several distinct advantages. They are relatively simple and intuitive to implement, supported by well-established technical frameworks. These approaches can directly leverage lexical information from source code, making them effective in cases where method names closely align with identifiers or API calls within the method body. Moreover, token sequences are naturally compatible with large-scale pre-trained language models such as CodeBERT and CodeT5, which enhances their performance in various downstream tasks. Compared to structure-based approaches, token-based approaches typically require less computational overhead.

However, these approaches also face several limitations. They struggle to capture deep structural information, including hierarchical syntax, control flow, and data dependencies [95]. This reliance on a linear sequence of tokens means they can be sensitive to semantically-equivalent code transformations that alter the token sequence. For example, a simple refactoring of a for loop into an equivalent while loop results in a substantially different input sequence, which can be challenging for a model to recognize as functionally identical without extensive training on such examples. Similarly, their performance can be affected by common stylistic variations in

Table 6. AST parsers and AST-based tools used in existing MNP studies

Parser Year	JavaParser	Spoon	srcML	Joern	PSIMiner	Tree-sitter	ANTLR	Gumtree	Javalang	JDT	Semantic	Astminer	fAST	Total
2013	0	0	0	0	0	0	0	0	0	[7]	0	0	0	1
2018	[9]	0	0	0	0	0	0	0	0	0	0	0	0	1
2019	[10] [8, 43]	0	0	0	0	0	0	[53]	0	0	0	0	0	4
2020	[101, 105] [29, 74]	[19]	0	0	0	0	0	0	0	0	0	0	0	5
2021	[33]	0	[13]	[95]	[82]	[66]	0	0	0	0	[128]	0	[14]	7
2022	[68]	0	0	0	0	0	0	0	[52]	0	0	[109]	0	3
2023	[12, 51] [11, 69, 120]	0	0	0	0	[51, 69] [32]	[51]	[51]	[51, 69] [35]	[69]	0	0	0	7
Total	15	1	1	1	1	4	1	2	4	2	1	1	1	-

expressions; for instance, expressing a boolean check as `if(!flag)` versus the semantically identical `if(flag==false)` produces different token sequences that a model might treat as distinct. As shown by Rabin et al. [73], such semantic-preserving transformations significantly reduce the performance of neural program models like `code2vec` [10] and `code2seq` [8]. Promisingly, this limitation is increasingly being addressed by the sheer scale of modern pre-trained code models, which learn to recognize many of these patterns from vast code corpora implicitly.

Considering these characteristics, token-based approaches are most applicable in scenarios where method functionality is reflected by key operations or explicit vocabulary in the method body. They are also suitable as lightweight baseline models or rapid prototypes, particularly when support for complex program analysis is unavailable. The use of pre-trained code models can alleviate structural limitations by implicitly incorporating abstract syntax knowledge. Furthermore, token-based approaches can complement structural models as part of multi-modal approaches, contributing lexical-level insights to enhance overall prediction quality.

4.2 AST-based MNP Approaches

The AST, as a direct representation of the syntactic structure of source code, provides richer structural information for MNP than pure token sequences and constitutes an important technical paradigm in MNP research.

4.2.1 Background and Motivation. AST-based approaches typically begin by parsing source code into an AST, from which structural features, such as paths, subtrees, and node sequences, are extracted to capture the syntactic and hierarchical relationships within the code. These structured representations are then used to learn and predict method names. The motivation for using ASTs lies primarily in their ability to preserve structural information that is lost in linear token sequences. Unlike flat token streams, ASTs explicitly encode the nested composition of expressions, statements, and declaration-use relationships, offering a more informative and unambiguous context for model learning [8].

Moreover, AST-based representations improve robustness to superficial code transformations. Refactorings such as safe variable renaming, formatting changes, or stylistic variations typically alter tokens but leave the underlying structure intact; ASTs mitigate the impact of such changes by abstracting away surface-level noise. Furthermore, ASTs enable fine-grained localization of semantic units and clearer modeling of syntactic dependencies. For instance, specific nodes representing API calls, control structures, or variable declarations can be directly identified and analyzed in relation to surrounding constructs, thereby supporting more precise and semantically aware predictions.

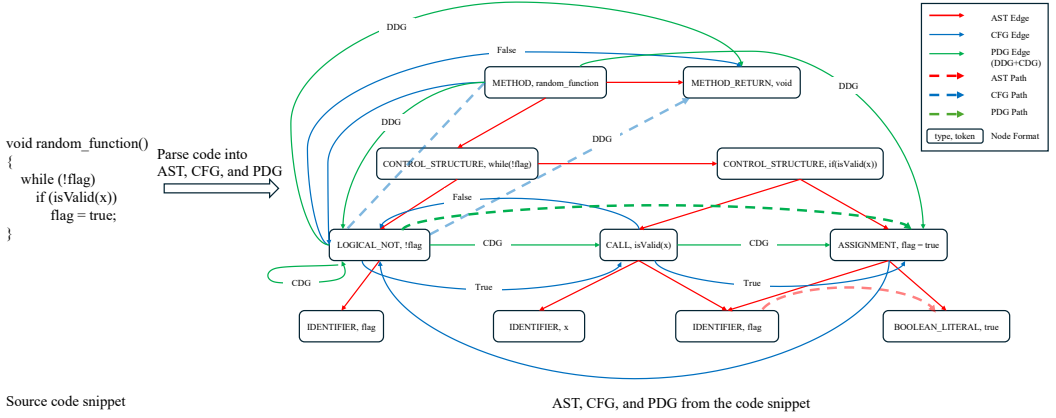


Fig. 9. Example of a code snippet along with its corresponding AST, CFG, and PDG provided in [95]

4.2.2 AST Extraction and Representation. In AST-based approaches, generating the AST is the initial step. An AST is a hierarchical tree representation that captures the syntactic structure of source code. It omits superficial elements such as parentheses, semicolons, whitespace, and some comments, and instead emphasizes the logical structure and key components of the code. Internal nodes typically represent syntactic constructs, while leaf nodes correspond to identifiers or literals. The parent and child relationships in the tree reflect syntactic subordination among code elements [8, 10].

AST construction depends on language-specific parsers. Table 6 summarizes the widely used parsers and AST-based tools in MNP research. These include JavaParser [8–10, 43, 101], Spoon [19], srcML [13], Tree-sitter [66], and others such as ANTLR [51], Javalang [52], GumTree (AST-based tree differencing tool) [51], Eclipse JDT [69], fAST (flattened AST representation for fast traversal and differencing) [117], and PSIMiner (IntelliJ PSI-based AST miner with type, reference, and symbol resolution) [82]. Furthermore, we count all these parsers and tools used in MNP studies, and the results are shown in Table 6. It is observed that there are various AST parsers used in learning-based MNP research, among which JavaParser is used most frequently. Fig. 9 illustrates a code snippet along with its corresponding AST, **Control Flow Graph (CFG)**, and **Program Dependence Graph (PDG)**, adapted from [95]. In this figure, red, blue, and green solid edges represent the structural connections in the AST, CFG, and PDG, respectively, while red, blue, and green dashed paths highlight example AST, CFG, and PDG paths. This subsection focuses on the AST, whose structural edges are highlighted in red.

The **Program Structure Interface (PSI)** tree extends the AST by combining syntactic structure with enhanced granularity and interactive access to code elements. Unlike standard ASTs, PSI trees preserve additional syntactic details such as punctuation, braces, and formatting. Each PSI tree contains all AST nodes and further augments them with additional structural elements, including modifier nodes, internal structural constructs, and other Java-specific syntactic components, even if they are not explicitly present in the code. As illustrated in Fig. 10, red nodes represent original AST elements, green nodes denote literal tokens (e.g., variable names and type keywords), and blue nodes correspond to supplementary syntactic structures introduced by PSI. The root node *PsiMethod* encapsulates the full method structure, including modifiers, return type, method name, parameter list, reference list, and method body. PSI further enriches the representation through

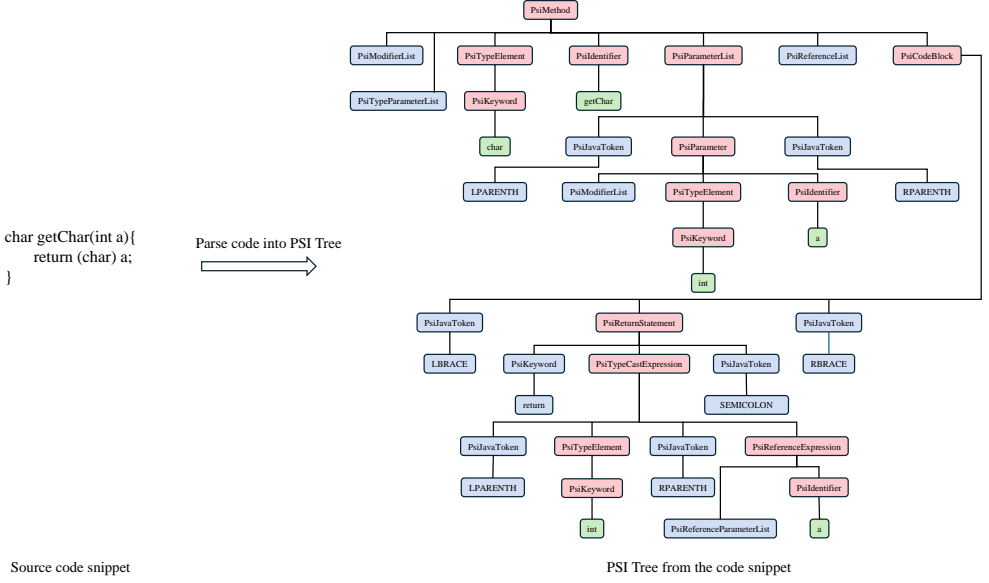


Fig. 10. Example of a code snippet and corresponding PSI tree provided in [82]

intermediate nodes such as *PsiJavaToken*, *PsiKeyword*, and *PsiTypeCastExpression*, which make syntactic features like keywords and type casts explicit.

The choice of parser can significantly impact the structure of the generated AST and, consequently, the performance of AST-based models. As discussed in Section 6.2, empirical studies by Li et al. [51] and Qian et al. [69] demonstrate that different parsers often produce ASTs with substantial structural variation. These discrepancies affect feature extraction and model learning. Qian et al. [69] further found that parser selection has a direct impact on MNP effectiveness, with Eclipse JDT achieving superior results in their experiments.

4.2.3 AST Preprocessing Techniques. Raw ASTs are often too large and complex to be directly used as input for learning models. To address this, various preprocessing techniques have been proposed to extract informative structural features or transform ASTs into more tractable forms. One of the most widely adopted approaches is AST path extraction, introduced by Alon et al. [9], and successfully applied in models such as code2vec [10] and code2seq [8]. This technique involves extracting paths between terminal nodes (typically identifiers or literals), with each path encoded as a sequence of AST nodes and their traversal directions. A path context is commonly defined as a triplet: (start_token, path_nodes, end_token). For example, as shown in the AST portion of Fig. 9 shows, the statement *flag = true* can be represented as the following AST path [95]: (*IDENTIFIER, flag* ↑ *ASSIGNMENT, flag = true* ↓ *IDENTIFIER, true*). The {↑, ↓} are the directions of movements between path nodes in the AST. In this representation, each element denotes an AST node consisting of two fields separated by a comma. The first field indicates the syntactic category of the node, while the second field specifies the actual code token associated with that node. To manage complexity, parameters such as maximum path length, width, and sampling rate are used, with models like code2seq employing iterative resampling for robustness.

Another key method is subtree extraction, which decomposes ASTs into compact, meaningful substructures that reflect common local patterns or reusable code fragments. These subtrees serve

as a "structural vocabulary" for learning [13]. For instance, InferCode [13] uses subtree prediction as a pretext task in self-supervised learning.

To make ASTs compatible with sequential models, linearization strategies have been developed. These include depth-first traversal, aSBT, and other syntactic encoding schemes. Using aSBT, Xie et al. [110] transformed the AST into two sequences: one sequence represents the code tokens, while the other captures the type information from the AST. These sequences are then embedded together into a single aSBT token embedded sequence using an embedding layer. This enhanced traversal technique allows for a richer representation of the code, capturing both syntactic and structural features more effectively compared to traditional AST traversal methods, thus improving performance in tasks such as code understanding and prediction.

Statement-Centric AST (SC-AST) is a refined AST representation focused on program statements. It partitions code into individual statements, constructs a separate AST for each, and preserves their sequential order. Originally proposed in [109], SC-AST is built by extracting statement nodes from a standard AST, treating each as the root of a subtree, and reversing edge directions to emphasize inter-statement relationships. This structure effectively captures fine-grained, statement-level syntactic dependencies.

Blended Trace is a sequential representation that integrates symbolic and concrete execution information to capture both the structural and behavioral aspects of a program. As introduced in [101], each element in a Blended Trace pairs a symbolic statement with the corresponding concrete program state observed during execution. This dual representation enables a more comprehensive modeling of program behavior by reflecting both abstract control-flow logic and actual runtime semantics, thereby enhancing downstream tasks such as MNP through enriched execution-aware context.

An AST-based Test Graph is a specialized graph representation designed to capture both the structural and semantic relationships between unit test methods and their corresponding **Methods Under tests (MUTs)**. Petukhov et al. [68] propose this approach to enhance MNP, particularly for unit test naming tasks. The construction begins with extracting the AST of a unit test method from the source code, followed by the augmentation of semantic edges that reflect data-flow and control-flow dependencies between variables. To localize relevant behavior, a subgraph corresponding to the test method body is isolated and further enriched with control-flow edges that model execution order. MUTs are identified via static call graph analysis, and their ASTs are incorporated into the graph. These are then connected to the unit test graph using additional control-flow edges to form a unified representation. This integrated structure enables the model to leverage both syntactic constructs and execution semantics, thereby improving the precision and contextual relevance of generated method names.

To make PSI representations compatible with downstream models, PSI Tree Path extraction is applied as a preprocessing step. This process transforms PSI trees into path-based contexts, similar to AST path methods. As demonstrated by Spirin et al. [82], this involves converting the PSI tree into a set of paths, where type names are split into subwords and treated in the same manner as other tokens in the path context. By leveraging PSI's fine-grained structure, this method facilitates more detailed and expressive representations of code semantics, while maintaining compatibility with existing path-based neural architectures [8].

An abstract alternative to syntax tree-based representations is the Metrics Vector Representation, which summarizes source code through quantitative software quality metrics. As proposed in SENSE [65], this approach encodes code snippets into vectors based on properties such as size, cyclomatic complexity, cohesion, and coupling. By focusing on functional and structural characteristics rather than lexical content, it provides a lightweight, interpretable, and computationally

efficient representation of code, making it particularly useful in scenarios where fine-grained syntax is less critical.

Each method presents trade-offs. AST paths capture local structural dependencies effectively but are computationally intensive and may overlook global context. Subtrees provide reusable representations but require careful selection and integration. Linearized ASTs support sequential modeling but lose two-dimensional structure. SC-AST strikes a balance between global and local code representation. PSI-based methods offer fine granularity but may introduce noise. The optimal choice of preprocessing strategy depends on the downstream encoding model, task requirements, and available resources. Empirical findings by Qian et al. [69] further confirm the significant influence of preprocessing techniques on MNP performance

4.2.4 Prediction Models and Techniques. For MNP methods leveraging ASTs and their derived representations, prediction models are generally categorized into classification-based and generation-based approaches.

In classification-based methods, encoders are designed to capture structural information from the AST and transform it into fixed-size vector representations. For instance, code2vec [10] aggregates multiple AST paths using attention mechanisms to produce a global code embedding. **Tree-Based Convolutional Neural Networks (TBCNN)** [13] apply convolution operations directly over tree structures, while SENSE [65] likely employs attention to highlight salient AST regions. Serialized ASTs have been modeled with n-gram language models [7], and linearized AST statements have been encoded using **Bidirectional Long Short-Term Memory (Bi-LSTM)** with multi-head attention in Fold2Vec [12]. The resulting embeddings are typically passed to classifiers such as fully connected layers with Softmax, or to SVMs [19] and CRFs [9].

Generation-based methods use more diverse encoders to accommodate sequence output. In [8], code2seq encodes each AST path independently via Bi-LSTMs. Tree-LSTM architectures [101] directly process hierarchical AST structures, while GNNs [68] treat ASTs as graphs to capture richer dependencies. For linearized ASTs or path sequences, Transformer-based encoders [11] offer strong performance, with variants optimized for efficiency. Self-attention mechanisms have also been applied to SC-ASTs [109]. Additionally, Bi-LSTMs [35] and RNNs [110] are employed for sequence encoding of flattened ASTs. These encodings are then fed into decoders, typically LSTM/GRU or Transformer-based, that autoregressively generate method names, often incorporating attention and copy mechanisms for improved accuracy.

4.2.5 Advantages, Limitations, and Applicability. AST-based approaches offer distinct value in MNP by effectively capturing the syntactic structural information of code, enabling a deeper understanding of code composition compared to purely token-based approaches. Their emphasis on structure over specific identifiers also provides robustness to certain refactorings, such as safe variable renaming or formatting changes. In addition, AST node types, hierarchical relations, and traversal paths constitute a richer feature space than raw tokens.

However, AST-based approaches face several limitations. First, ASTs can become large and complex for long methods, introducing significant computational overhead for preprocessing and feature extraction. Second, variations in AST parsing tools and preprocessing techniques can substantially impact model performance, making reproducibility and fair comparison across studies more difficult [51, 69]. Third, as ASTs primarily encode static syntax, they are less effective at capturing semantic properties such as data flow, control dependencies, or runtime behavior, which are more naturally expressed through program graphs like CFG or PDG. Finally, while ASTs do capture identifiers in their terminal nodes, the way models process this information can lead to a reduced sensitivity to the lexical semantics of key identifiers. This issue arises when models become overly reliant on structural features [43]. In many AST-based models, such as those

using path-based representations (e.g., `code2vec`), the final code embedding is an aggregation of information from numerous structural paths. If a particular syntactic structure is very common and strongly correlated with certain names in the training data, the model may learn to prioritize this recurring structural signal. In the aggregation process, the semantic contribution of a single, crucial but perhaps less common identifier can be overshadowed or diluted by the weight of the more dominant structural patterns. For instance, consider two distinct methods: one that reads from a file and a buffer, and another that gets data from a network and a cache. Both might share an identical AST structure (e.g., two variable assignments from method calls, followed by a return statement). A model that over-relies on this common structure might struggle to fully differentiate between the critical lexical cues, `read` versus `get`, and thus may generate a similar or incorrect name for both.

Accordingly, AST-based approaches are particularly applicable in scenarios where method names correlate strongly with structural patterns, such as specific control constructs, API usage, or syntactic templates, or when robustness to surface-level changes is important. They also serve as a valuable component in multi-modal architectures, complementing token-based, graph-based, or contextual signals (e.g., comments) to support more comprehensive code understanding.

4.3 Graph-based MNP approaches

To capture program semantics beyond what ASTs provide, particularly aspects such as control flow and data dependencies, researchers have increasingly adopted more expressive graph-based representations. These representations model execution dynamics and variable interactions, enabling the application of graph learning techniques to enhance MNP.

4.3.1 Background and Motivation. The central idea of these approaches is to convert the source code into one or more graph representations of structured programs. In such graphs, nodes typically correspond to statements, basic blocks, or higher-level program units, while edges capture control flow, data flow, call relationships, or other semantic dependencies. Graph learning methods, particularly GNNs, are then applied to learn vector representations from these graphs, which are subsequently used for MNP. The key motivations include capturing control flow information through CFGs to model execution paths; representing data flow and control dependencies via PDGs to reflect variable lifecycles and computational logic; extending beyond syntactic structure to better model program semantics; achieving robustness against syntactic variations that preserve core semantics; and integrating diverse relations through advanced graph formats such as CRGs and HPGs to construct more expressive representations.

4.3.2 Graph Extraction and Representation. Representative graph types include the CFG [56, 68, 95, 105, 113], where nodes represent basic blocks or statements and edges denote control transfers. CFGs are typically extracted using tools such as Spoon [105, 113], Joern [95], and Soot [56, 68]. A CFG is a directed graph in which nodes represent predicates and executable statements, and edges indicate the control flow between them, with optional labels describing execution conditions for each path. Fig. 9 also includes the CFG corresponding to the same code snippet, with control flow edges highlighted in blue.

The PDG [54, 56, 95, 119] captures both data and control dependencies among statements or predicates. PDG nodes represent statements or conditions, while edges reflect control or data dependencies. Multiple MNP approaches incorporate PDGs into model design [54, 56, 95, 119]. Zhang et al. [119] use Soot [96] to compute control and data flows at the Jimple level and construct PDGs. Liu et al. [54] apply Joern to generate PDGs, while Ma et al. [56] also use Soot, first converting Java code into Jimple before PDG extraction.

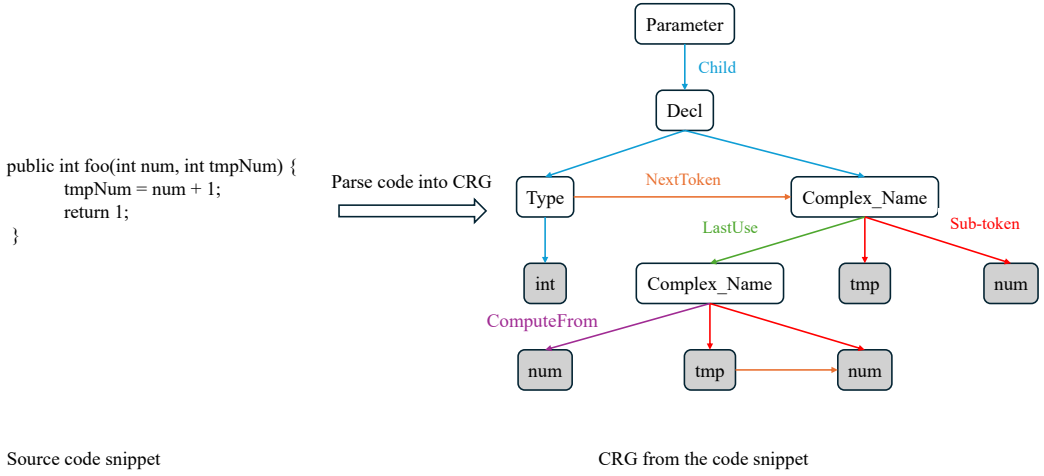


Fig. 11. Example of a code snippet corresponding CRG provided in [71]

PDG edges are of two main types: control dependencies, which describe how a predicate influences the control flow, and data dependencies, which indicate how the value of one variable affects another [95]. These properties make PDG particularly effective in capturing core computational logic and execution dependencies beyond syntax.

The CRG introduced by Qu et al. [71], captures multiple facets of code structure by integrating AST hierarchy, token order, and variable define-use relations. As a multi-relational, multi-edge graph, the CRG encodes both semantic and structural relationships among program elements through diverse node and edge types, unifying information from AST, control flow, and data flow into a comprehensive representation. Fig. 11 presents the CRG of the code snippet. They use the graph output format [5] from the code analysis tool fastast [117] to construct a CRG for each method, representing the structural context. The nodes in the graph represent different types of code elements, including syntactic constructs (e.g., parameter declarations *Decl*, type *int*) and lexical identifiers (e.g., variable name *tmpNum*).

HPG is proposed by Zhang et al. [121], which uses type information in the chart to represent code and can explicitly provide the types of nodes and edges. An HPG of a certain code snippet is generated based on its AST according to the **Abstract Syntax Description Language (ASDL)** [100]. HPG can relieve the type missing issue in the homogeneous graph [121]. Fig. 12 shows the HPG of the code snippet. Each node is assigned a type (left part) and a value (right part). Each edge has a type label. HPG is a format of heterogeneous graphs. A heterogeneous graph is a graph consisting of different types of entities, i.e., nodes, and different types of relations, i.e., edges. Zhang et al. [121] verify the promoting effect of HPG on MNP tasks. To generate HPG, they implement HPG parsers for two popular programming languages, including Python and Java. The Python parser conforms to the official Python 3.7 ASDL grammar¹. The implemented Python parser can extract up to 23 types of nodes and 71 types of edges (forward). As for Java, they implement an HPG parser based

¹<https://docs.python.org/3.7/library/ast.html#abstract-grammar>

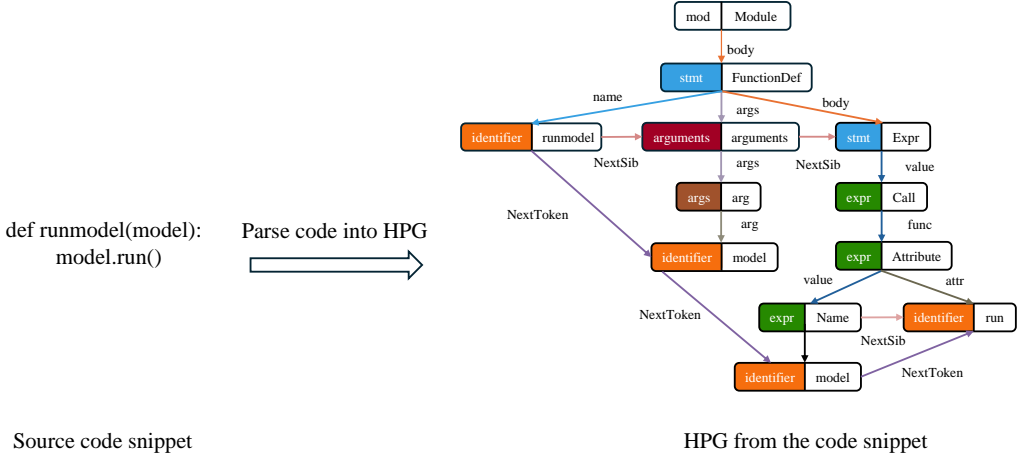


Fig. 12. Example of a code snippet and corresponding HPG provided in [121]

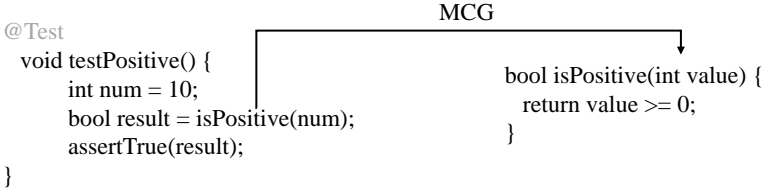


Fig. 13. An example of MCG provided in [68]

on the AST parser Tree-sitter. They manually define the field type for Java and assign types to the edges in the AST.

The **Method Call Graph (MCG)** [56, 68] is a high-level representation of inter-method invocations, where nodes represent methods and directed edges denote call relationships. It is typically constructed using tools such as Soot. MCGs enable the identification of methods invoked by a given target method, providing a concise abstraction of method-level dependencies within a program. Fig. 13 presents an example of the MCG for the given code snippet. The unit test method `testPositive()` calls the method `isPositive(intvalue)`, and this invocation is represented by a directed edge from the node corresponding to `testPositive()` to the node corresponding to `isPositive()`. The MCG abstracts away internal implementation details such as variable declarations or control flow, focusing solely on inter-method interactions. This graph is widely used in static analysis, program comprehension, and test impact analysis, as it captures dependencies between program components and helps identify how changes in one method may affect others.

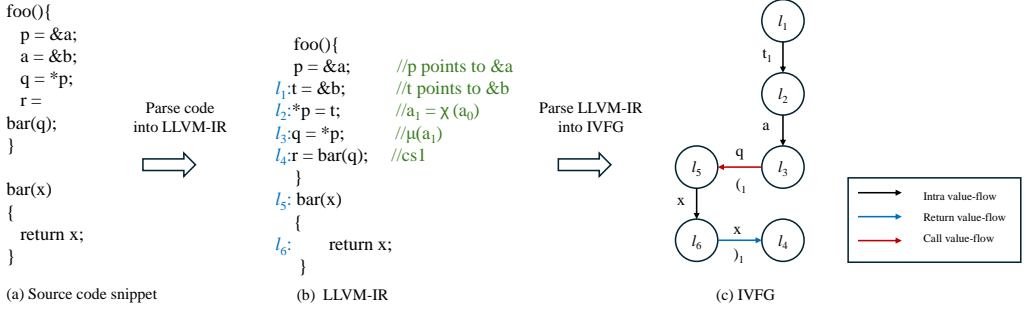


Fig. 14. Example of a code snippet and corresponding LLVM-IR and IVFG provided in [84]

The **Aggregated Call Graph (ACG)** is used by Yonai et al. [116] to improve MNP via graph embeddings. It captures functional similarity through embeddings learned from caller-callee relationships. In the ACG, each node represents a set of methods with the same name, and edges denote calls between these sets. Compared to traditional call graphs, the ACG reduces complexity by identifying nodes solely by method names, which simplifies structure and lowers the computational cost of embedding. Unlike call graphs, the ACG is constructed directly from source code, avoiding the overhead of explicit call graph construction. The proposed algorithm extracts each method's name and the names of all methods it calls, then creates nodes and edges accordingly. This process requires no semantic analysis, enabling efficient computation. Furthermore, because each source file is processed independently, the construction is easily parallelizable.

The **Interprocedural Value-Flow Graph (IVFG)**, built upon **LLVM-Intermediate Representation (LLVM-IR)** and introduced by Sui et al. [84], captures value flow across procedures for deep semantic modeling. LLVM-IR is a uniform code representation generated by modern compilers like LLVM, which supports various programming languages through multiple front-ends. It is widely used for program analysis and transformation tasks, representing a program's code in a structured and language-agnostic manner. Fig. 14 presents a C code snippet alongside its corresponding LLVM-IR and illustrates how an IVFG is constructed from it. Fig. 14(a) presents the original C source code, where the function *foo* performs a sequence of pointer assignments and invokes the function *bar*, which returns its argument. Fig. 14(b) shows the corresponding LLVM-IR, where each instruction is labeled from l_1 to l_6 , capturing low-level memory operations such as loading, storing, and calling functions. Annotations in green provide additional semantic interpretations, such as value definitions. Fig. 14(c) depicts the IVFG, where nodes represent LLVM instructions and directed edges represent the flow of values. Black edges denote intra-procedural value flows within a function, red edges capture call-site value propagation from caller to callee, and blue edges represent return-site value flows from callee back to caller.

4.3.3 Graph Preprocessing Techniques. Raw program graphs, particularly complex ones such as PDG and HPG, often require dedicated preprocessing to be effectively utilized by graph learning models like GNNs. Common preprocessing techniques include: path extraction from CFGs or PDGs [54, 105], which partially linearizes graph structure into sequences but may sacrifice global context. Similar to ASTs, CFG paths can be constructed, for example, a CFG path associated with the node *METHOD*, representing the method *random_function*. This CFG path can be expressed as: (*METHOD*, *random_function* \uparrow *LOGICAL_NOT*, *!flag* \downarrow *METHOD_RETURN*, *void*), where each tuple consists of a node type and the associated code token. The PDG illustrated in Fig. 9

shows a path corresponding to the node *LOGICAL_NOT*, associated with the expression *!flag*. This PDG path can be represented as: (*LOGICAL_NOT*, *!flag* \uparrow *METHOD*, *random_function* \downarrow *ASSIGNMENT* *flag = true*). In this representation, each tuple denotes a PDG node, where the first element indicates the syntactic or semantic category of the node, and the second represents the actual code token or expression linked to it.

The abstraction cycle in the **Graph Interval Neural Network (GINN)**, proposed by Wang et al. [105], employs three operators: partitioning, heightening, and lowering. These operators iteratively abstract and recover the CFG, enabling effective semantic embedding learning by ensuring sufficient and structured message propagation across the graph. This facilitates the capture of both local and global contextual information.

The σ Graph introduced in [54] facilitates detailed program analysis, particularly in identifying security vulnerabilities and concurrency issues. It includes two main variants. The σ_0 graph captures control and data flow, while the σ_1 graph builds on this foundation by incorporating additional semantic information. These representations support fine-grained method-level analysis by modeling both structural and semantic characteristics of program behavior.

Subtokenization of token-containing graph nodes is applied in HPG by Zhang et al. [121] to improve semantic representation and address the OOV problem resulting from large vocabularies. Terminal identifier nodes are split into subtoken nodes based on camel case or snake case conventions [4]. For instance, the identifier *train_model* is divided into *train* and *model*, with subtoken_of edges linking each subtoken to its original node. Reverse edges are also added to ensure bidirectional connectivity. Zhang et al. [121] propose two subtokenization strategies. The shared subtoken strategy reuses identical subtoken nodes across different identifiers. For example, the *model* node is shared between *train_model* and *test_model*, which helps reduce graph size. The independent subtoken strategy treats the subtokens of each identifier separately. Although this enables context-sensitive representations, it increases the overall size of the graph.

In approaches based on the IVFG, high-order proximity embeddings have been introduced to capture more comprehensive program semantics. Sui et al. [84] propose Flow2vec, which compiles source code into LLVM-IR, performs pointer analysis to generate a memory **Static Single Assignment (SSA)** form and IVFG, and resolves indirect function calls. The method then applies matrix decomposition techniques, including the Katz Index and **Singular Value Decomposition (SVD)**, to approximate embedding vectors that preserve context-sensitive value flows efficiently, even in sparse graphs.

In summary, path-based methods offer computational simplicity at the cost of losing global structural information, while full-graph models capture richer semantic details with higher computational overhead. Effective preprocessing, including feature engineering and graph sampling, is crucial for striking a balance between expressiveness and efficiency in graph-based program representations.

4.3.4 Prediction Models and Techniques. Graph-based approaches for MNP primarily employ GNNs as encoders to capture both structural and semantic information from program graphs.

For classification-based approaches, common encoders include the **Relational Graph Convolutional Network (R-GCN)** [71], which is particularly effective for heterogeneous graphs such as PDGs or CRGs. R-GCN models multi-relational data by integrating edge-type-specific transformations, while employing weight regularization techniques (e.g., basis or block diagonal decomposition) to reduce overfitting. R-GCN extracts relational features, while **Gated Graph Sequence Neural Network (GGNN)** introduces time-step updates to capture sequential context during message passing, enhancing the integration of structural and semantic features. Beyond relational GNNs, high-order proximity embedding methods such as Flow2vec [84] are used to

encode IVFGs, capturing long-range data dependencies. When graphs are preprocessed into path sets, path-based models like code2vec [10] can be applied to encode semantic paths. The resulting graph or node representations are typically fed into classifiers—commonly fully connected layers followed by a Softmax function—for label prediction.

For generation-based approaches, encoder architectures are more diverse. Xu et al. [113] proposed a **Hierarchical Attention Network (HAN)** to encode CFGs represented as sequences of basic blocks. Tokens within blocks are embedded using GRUs, and attention mechanisms at both the token and block levels are applied to learn context-aware representations with adaptive weighting. Wang et al. [105] introduced the GINN for CFGs, incorporating control interval features. In addition, general GNNs such as GGNN have been widely adopted for various graph structures. Zhang et al. [121] proposed an encoder based on the **Heterogeneous Graph Transformer (HGT)**, which leverages mutual attention mechanisms across node and edge types, enriched with positional encodings (e.g., from depth-first AST traversal) and attention-based message passing to generate expressive multi-layer representations.

In scenarios where graphs are linearized into path sequences, Transformer encoders provide an alternative for learning long-range dependencies. Encoder outputs, regardless of architecture, are typically coupled with RNN or Transformer-based decoders, often augmented with attention and copy mechanisms, to generate target method names in sequence.

4.3.5 Advantages, Limitations, and Applicability. The primary advantage of graph-based approaches lies in their capacity to capture complex program dependencies and deeper semantic relationships beyond what is accessible through ASTs or token-based representations. By explicitly modeling control flow, data flow, and heterogeneous interactions between program entities, these approaches enable more accurate interpretation of code behavior and intent. In theory, they also offer greater robustness to semantically equivalent code transformations, as long as core structural dependencies are preserved. Moreover, graph-based representations introduce richer structural priors for code representation learning, potentially improving generalization in downstream tasks.

Despite these strengths, graph-based approaches also present notable limitations. Constructing and processing program graphs often involves significant computational overhead and complexity, requiring sophisticated static analysis tools. Graph sizes can be large and variable, posing scalability challenges. Training GNNs on such structures may suffer from issues such as over-smoothing and inefficient information propagation. The design of initial node and edge features is critical, as irrelevant or noisy graph information can degrade model performance, highlighting the importance of effective signal filtering.

Consequently, graph-based approaches are most suitable in scenarios where method names are closely tied to intricate program logic, such as data flow, control decisions, or interprocedural dependencies, and where a nuanced semantic understanding is essential to distinguish between methods with similar surface forms but divergent internal behaviors. These methods are particularly advantageous when sufficient computational resources and reliable program analysis tools are available to construct high-quality graph representations. For example, when methods share similar AST structures but differ in internal data processing or control logic, graph-based models can provide meaningful performance gains.

4.4 Advances and Trends in MNP

To address the limitations of single-paradigm information representations and fully exploit the complementary strengths of various contextual sources, hybrid approaches and emerging trends have garnered increasing attention in the field of MNP.

4.4.1 Approaches Combining Multiple Contextual Sources. These approaches aim to overcome the limitations of relying on a single type of program context by integrating multiple heterogeneous sources. To comprehensively understand a method's functionality, MNP techniques draw upon three primary categories of contextual information: local context, global context, and document context.

- **Local Context** refers to information contained entirely within the target method's signature and body. As introduced in Section 2.2 and illustrated in Fig. 2, this includes the sequence of statements, control flow structures, variable declarations, and parameters that define the method's direct implementation.
- **Global Context** encompasses information from the surrounding codebase, outside the method itself. This can include the name of the enclosing class, its attributes and superclasses, sibling methods, and the callers or callees of the target method. This context provides broader architectural and relational cues about the method's role in the larger system.
- **Document Context** consists of natural language artifacts associated with the code, most commonly Javadoc or other forms of comments. As shown in the example in Fig. 8, this documentation often contains a high-level, human-readable summary of the method's purpose, which can provide powerful semantic clues not always present in the code itself.

A common implementation strategy, often employed by hybrid models for tasks like MNP, is feature-level fusion. The fundamental approach involves combining different facets of a program to capture implementation details (local), relational information (global), and human-authored intent (document). Specifically, representations are extracted from sources like code tokens and ASTs to capture local implementation details and syntactic structure, from PDGs for global relations, and documentation for human-authored intent. These diverse representations are then merged, typically via concatenation, weighted summation, or bilinear pooling, into a unified vector, which is fed into the prediction model to achieve a more comprehensive, robust representation and enhance overall performance. For example, Meth2Seq [119] combines PDG paths, typed IR statements, and comments, while AUMENA [127], integrates local implementation features with the surrounding class context. Another strategy is model-level fusion or ensemble learning, in which separate models are trained on different contextual inputs and their predictions are aggregated. In addition, some works introduce multi-modal attention mechanisms that jointly attend to different sources of information, allowing the model to dynamically learn the relevance and interactions across modalities, such as between structural code elements and natural language comments.

While these hybrid approaches show promising improvements, they also introduce new challenges. Effectively integrating heterogeneous information requires careful design to avoid signal conflicts or the dominance of one modality. Moreover, the increased model complexity and computational overhead may pose scalability issues, particularly when dealing with large-scale datasets or limited hardware resources.

4.4.2 Use of Pre-trained Models. The advent of large Transformer-based pre-trained language models, especially those trained on large-scale code corpora, such as CodeBERT [27], CodeT5 [106] has significantly advanced various SE tasks, including MNP. These models are typically applied either as feature extractors or through end-to-end fine-tuning on MNP-specific datasets. For very large-scale **Large Language Models (LLMs)**, prompt learning or in-context learning offers an alternative to full fine-tuning. AUMENA [127], for instance, applies prompt tuning to CodeT5 for MNP.

The benefits of LLMs include strong representation capabilities, effective knowledge transfer, reduced need for manual feature engineering, and the ability to model long-range dependencies. They also offer potential for multi-task and cross-lingual applications. However, their deployment

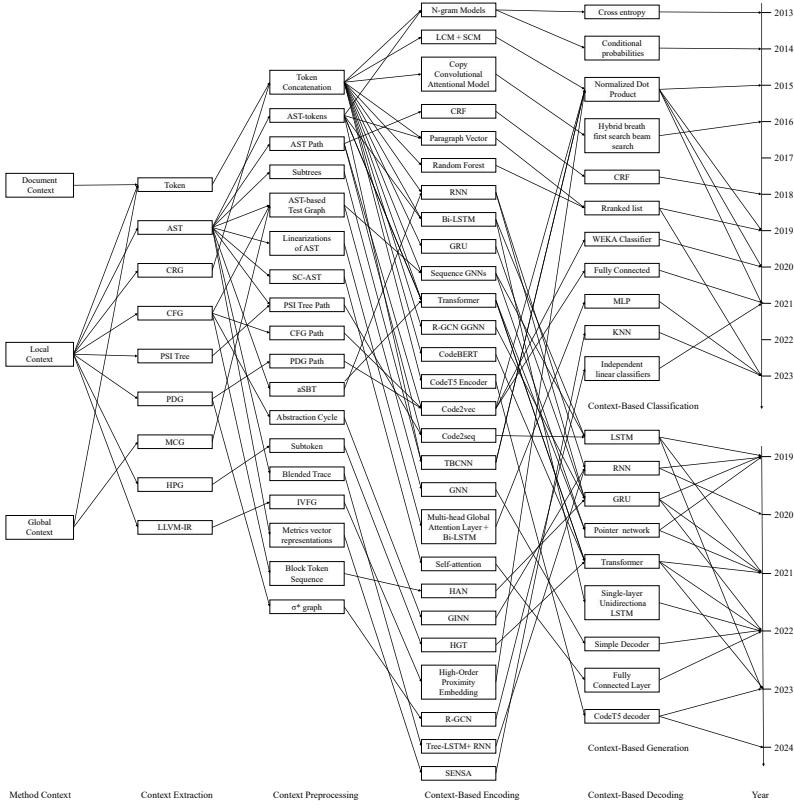


Fig. 15. Components of learning-based MNP

faces several challenges: high computational resource demands, generalization and domain adaptation limitations, interpretability concerns, restricted handling of extremely long contexts, and risks related to data contamination or bias from pre-training corpora.

4.4.3 Evolution of MNP Techniques. To contextualize the evolution of MNP techniques, Fig. 15 outlines the complete workflow of learning-based MNP, highlighting the core components, context extraction, pre-processing, encoding, and prediction, and their interdependencies. This end-to-end view helps clarify how innovations at each stage have shaped the development of increasingly effective MNP models. The evolution of MNP techniques reflects a broader trend in software intelligence research: the progression from shallow lexical models to structurally enriched and semantically expressive representations. Early token-based approaches relied on statistical language models such as n-grams or log-bilinear classifiers, which primarily captured token co-occurrence and local context features. With the rise of DL, models like RNNs, LSTMs, and GRUs were introduced to better model sequential dependencies in token streams. Attention mechanisms, including HANs, further improved focus on salient tokens. The adoption of Transformer architectures marked a turning point, offering improved handling of long-range dependencies and parallel computation. Most recently, pre-trained code models such as CodeBERT, CodeT5, and StarCoder have significantly advanced MNP performance by learning general-purpose code representations from large corpora, reducing the need for handcrafted features and improving generalization.

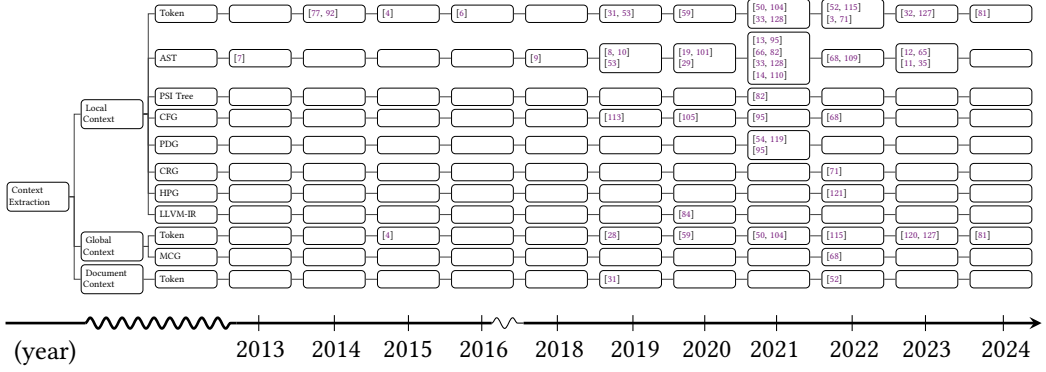


Fig. 16. Evolution of context extraction

In parallel, AST-based methods have explored the use of syntactic structure for capturing program intent. Early work included incorporating AST-derived features into token-based models, but a breakthrough came with `code2vec` and `code2seq`, which leveraged large sets of AST paths to learn distributed code representations via neural encoders with attention. Subsequent research advanced this line by introducing self-supervised objectives over subtrees (e.g., `InferCode`), structure-aware architectures like `Tree-LSTMs` and `GNNs` on ASTs, and linearization techniques (e.g., `SBT`, `aSBT`) for compatibility with sequence models. More recent approaches focus on hybridizing AST representations with token-level information, using multimodal fusion techniques and attention mechanisms to capture both structural and lexical cues. Empirical studies have also highlighted the importance of parser choice and AST preprocessing on downstream MNP performance.

Graph-based MNP methods represent a further shift toward richer semantic modeling. Early techniques extracted features from control-flow paths, but with the rise of Graph Neural Networks, researchers began constructing and leveraging various program graphs such as CFGs, PDGs, and more recently, heterogeneous graphs like CRGs and HPGs. These methods have employed increasingly sophisticated GNN variants—including GGNNs, R-GCN, and HGT, to capture both intra- and inter-procedural relationships. Recent work also explores combining GNN-derived features with token-based or pre-trained representations, enabling better integration of structural and lexical information. The overall trend points toward deeper graph modeling, the use of domain-adapted neural architectures, and a fusion of graph-based insights with other contextual signals to enhance semantic representation and prediction accuracy.

To make the historical trajectory of MNP research more explicit, Fig. 16 to Fig. 21 systematically trace the technical evolution of MNP approaches across four key stages: context extraction, preprocessing, encoding, and decoding. These diagrams reveal how advances in upstream components, such as token-based and structure-based representations (e.g., ASTs, CFGs, PDGs), have progressively enabled more sophisticated downstream techniques, including neural encoding schemes and classifier architectures. By mapping each technique to its origin paper and year, the charts clarify the dependencies and cumulative advancements that have shaped the MNP landscape.

4.5 Comparison of MNP Techniques

This section aims to provide a more direct and systematic comparison of the aforementioned primary MNP technical paradigms, token-based, AST-based, and graph-based, and to discuss their characteristics, trade-offs, and applicability across different dimensions.

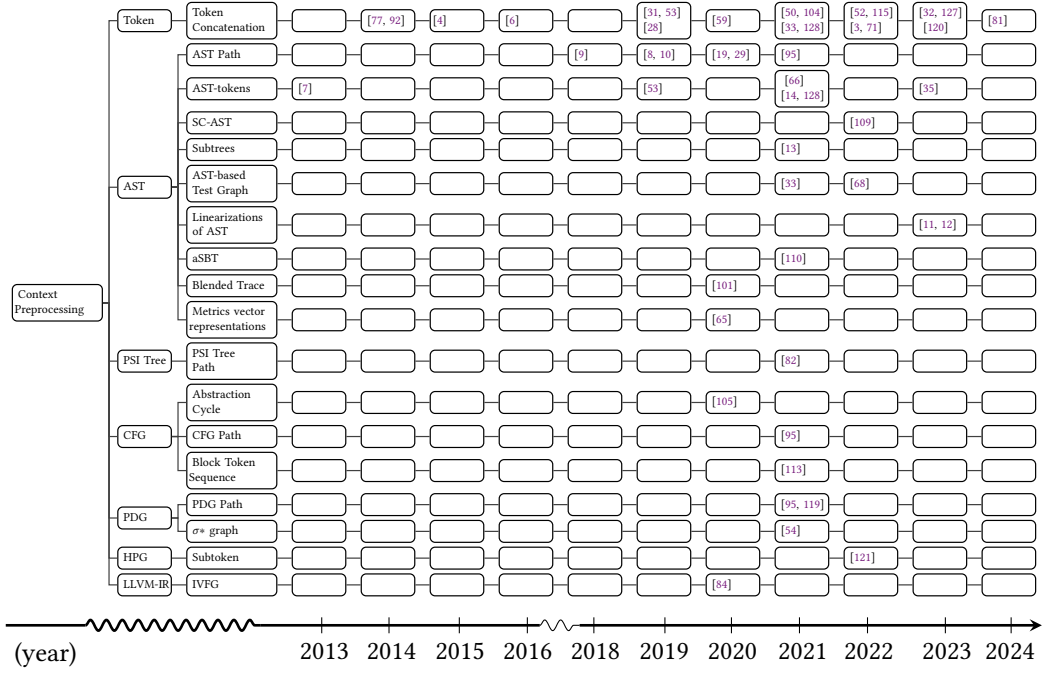


Fig. 17. Evolution of context preprocessing

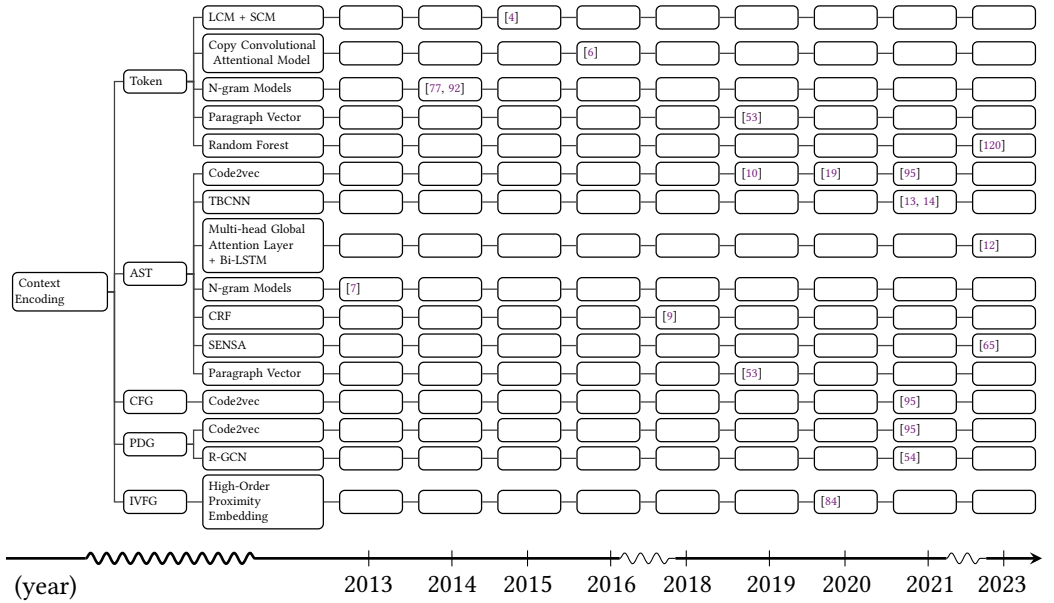


Fig. 18. Evolution of context encoding used in context-based classification techniques

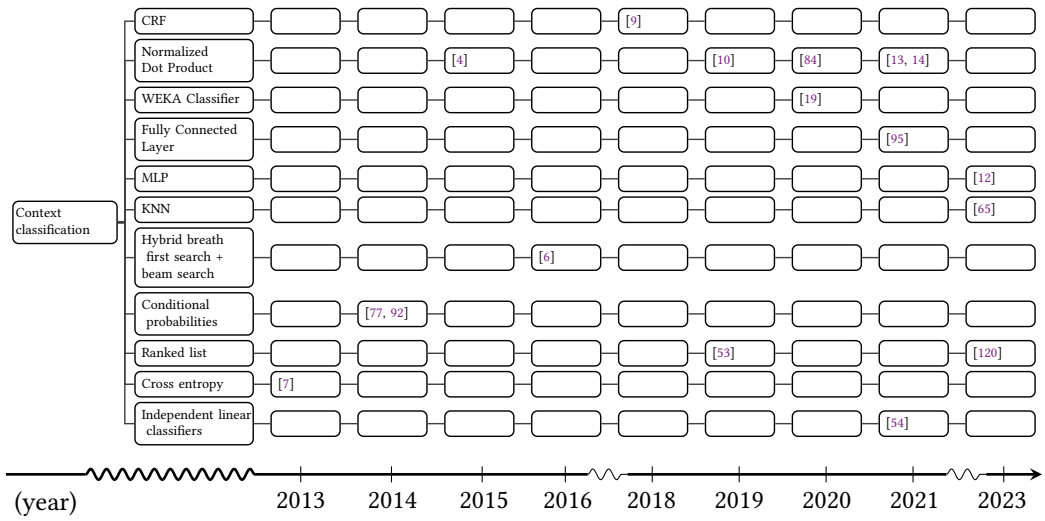


Fig. 19. Evolution of context classification used in context-based classification techniques

4.5.1 Comparison of Representational Capabilities. To compare the representational capabilities of different approach types in MNP, we examine token-based, AST-based, and graph-based approaches across several key dimensions.

Token-based approaches focus on lexical elements such as identifiers, keywords, and literals, offering strong access to surface-level code information. They are easy to implement and leverage well-established sequence modeling techniques from natural language processing. However, they lack structural awareness and are sensitive to minor variations in naming, formatting, and token order. Their ability to capture control or data flow is very limited, resulting in shallow semantic understanding.

AST-based approaches introduce syntactic structure by representing code as abstract syntax trees. This enables accurate modeling of hierarchical code organization and improves robustness to structural code transformations. As detailed in Section 4.2.5, while ASTs preserve lexical information through leaf nodes, their strength lies in capturing syntactic dependencies. However, an over-reliance on these structural features can sometimes lead to the dilution of key lexical signals during model training. Control flow and data dependencies are not explicitly modeled, though basic constructs like loops and conditionals can be indirectly inferred.

Graph-based approaches, including those based on CFGs and PDGs, go further by encoding execution semantics. CFGs explicitly model possible execution paths, while PDGs capture both data and control dependencies between code elements. These representations allow for a deeper understanding of how information flows through the program and how variables interact, offering robustness to semantically equivalent but syntactically different code variants. However, fine-grained lexical details may need to be reintroduced through node features.

Each approaches type was developed to address specific limitations of earlier approaches. Token-based approaches aim for simplicity and fast lexical modeling. AST-based approaches introduce structural understanding. Graph-based approaches incorporate semantic relationships such as control and data flow. More recent hybrid techniques and pretrained models attempt to integrate the strengths of multiple approaches or leverage large-scale data to improve generalization and reduce reliance on task-specific annotations.

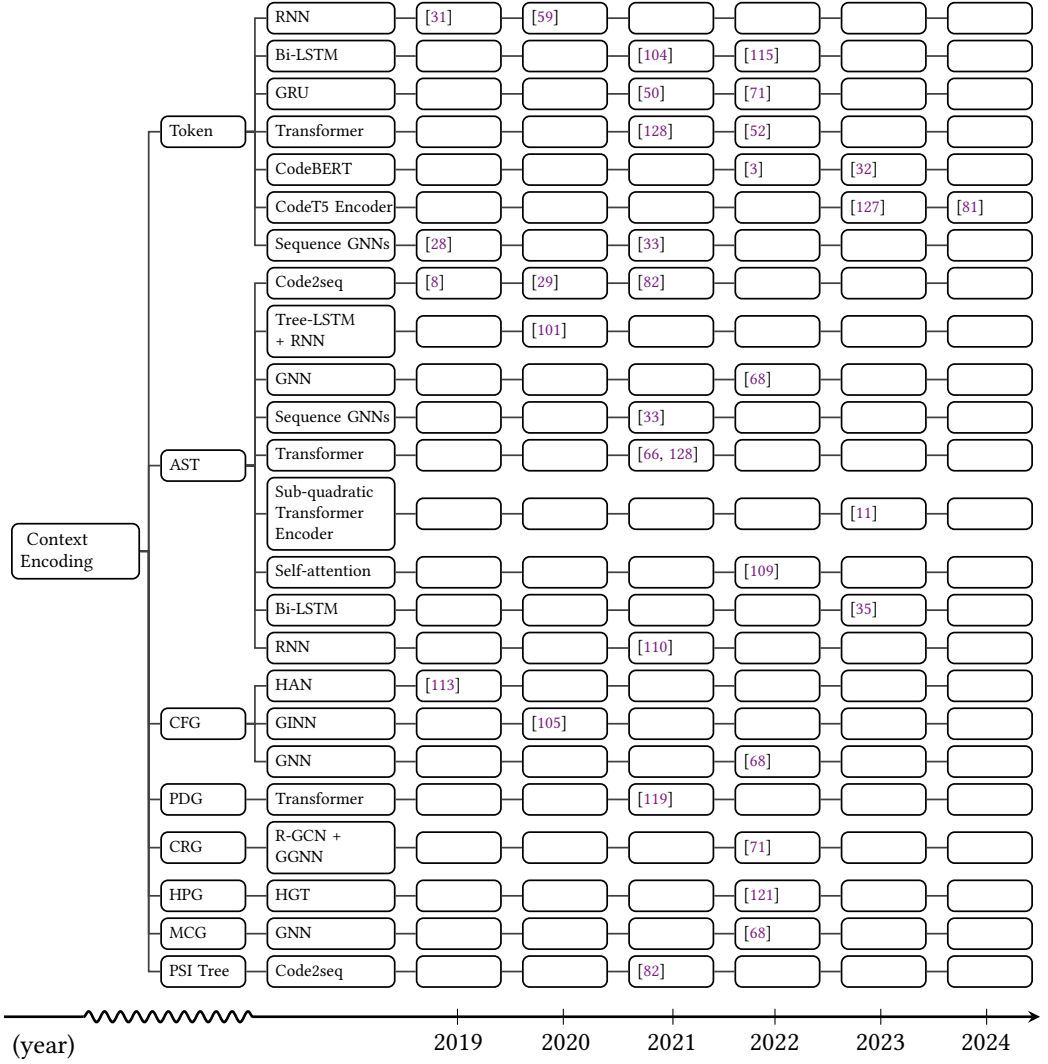


Fig. 20. Evolution of context encoding used in context-based generation techniques

4.5.2 Comparison of Performance and Cost Trade-offs. As part of our comparative analysis of MNP techniques, it is essential to examine how different approaches balance predictive capability with implementation and computational costs. This section provides a structured comparison across token-based, AST-based, graph-based, and pretrained approaches, focusing on trade-offs in performance, complexity, and data requirements.

Token-based approaches offer relatively low implementation complexity, especially when using standard tokenizers and simple sequence models. Their computational cost is generally modest, although it increases with model complexity (e.g., basic RNNs vs. large Transformers). While their representational power is limited, their performance can be substantially enhanced by incorporating large pretrained models that implicitly capture structural and semantic patterns. These methods typically require moderate to large labeled datasets for effective training.

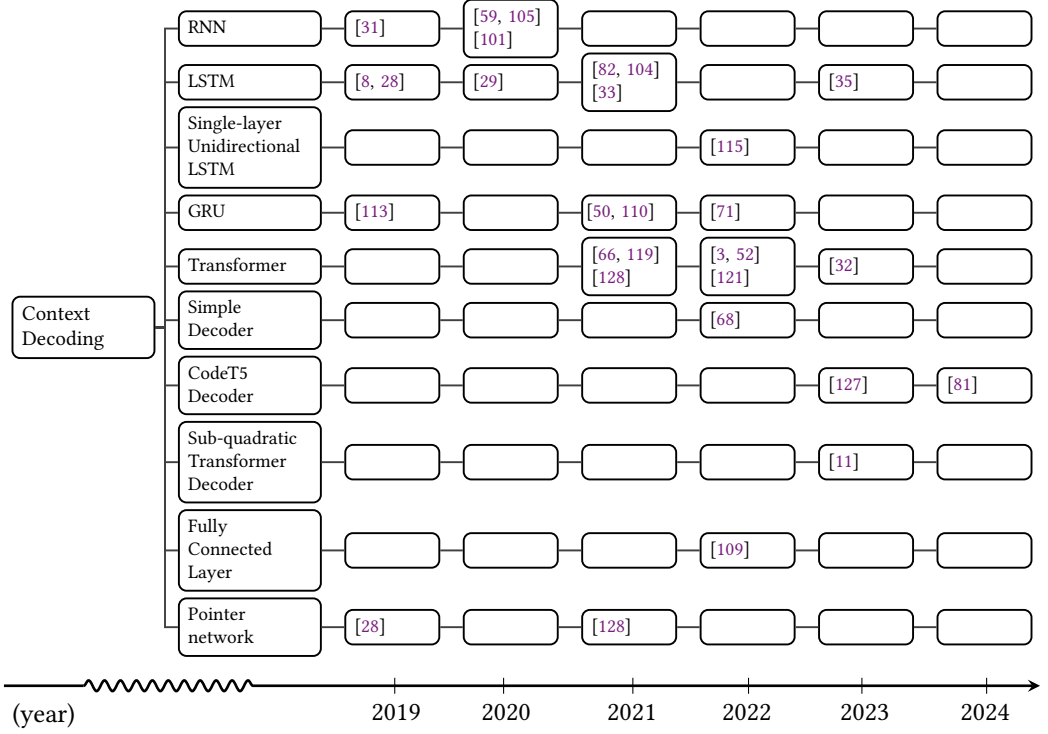


Fig. 21. Evolution of context decoding used in context-based generation techniques

AST-based approaches improve structural understanding through syntactic parsing and tree-based modeling. They require moderate implementation effort, including AST extraction and pre-processing (e.g., path or subtree sampling). Computational cost is higher than token-based methods due to tree traversal and more complex encoders. Data requirements remain similar, with gains in performance often justifying the added overhead.

Graph-based approaches offer the highest potential for semantic understanding by explicitly modeling control flow and data dependencies. However, they also incur the highest cost: graph construction requires advanced static program analysis, and GNN-based models are typically more expensive to train and deploy. Despite this, their ability to capture deep semantic relationships can lead to superior performance, especially in challenging scenarios.

Large pretrained models provide state-of-the-art results by leveraging massive-scale unsupervised training. Although the fine-tuning phase can be relatively straightforward, the overall computational cost, particularly in pre-training, is substantial. These models also require enormous amounts of unlabeled code for pretraining and still depend on labeled data for task-specific fine-tuning.

In general, approaches that capture deeper structural and semantic information tend to achieve higher predictive accuracy, but at the cost of increased complexity and resource demands.

4.5.3 Context-Specific Effectiveness of MNP Techniques. The effectiveness of MNP techniques varies with the input characteristics and the application scenarios, reflecting differences in generality, scalability, and alignment of the language tasks.


Token-based methods, especially at the subtoken level, are relatively language-agnostic. In contrast, AST-based and graph-based approaches rely on language-specific parsers, which limits

their portability unless intermediate representations, such as Tree-sitter or LLVM-IR, are used. Pre-trained models trained on multilingual corpora typically offer better cross-language adaptability.

Code scale and complexity significantly influence model suitability. Lexical and shallow AST-based methods suffice for small projects, while large, complex systems benefit from graph-based or pre-trained models. However, the overhead of AST and graph construction grows with project size, making scalability a key concern addressed by models like GINN [105].

Model performance is also sensitive to code quality and naming consistency. High-quality code supports better generalization, while noisy or inconsistent naming increases prediction difficulty.

Task-specific needs further affect model choice. Generative models are suited for creating novel names, whereas classification or ranking models perform better for consistency checking or candidate selection. Models like AUMENA [127] differentiate between these subtasks, and approaches such as code2vec [10] perform particularly well on methods with regular naming patterns, such as getters and setters.

 **Summary** ▶ A comprehensive review of MNP techniques is provided, covering token-based, AST-based, graph-based, hybrid methods, and recent advances with pre-trained models. These approaches are examined along the MNP workflow: context extraction, preprocessing, encoding, and prediction. The evolution of techniques reflects a shift from shallow statistical models to structurally enriched and multi-modal representations, highlighting trade-offs in representational power, model complexity, efficiency, and suitability for practical naming tasks.

5 ANSWER TO RQ2: WHAT ARE THE METHODS EMPLOYED TO EVALUATE LEARNING-BASED MNP TECHNIQUES?

In this section, we investigate and summarize the key approaches used to evaluate the 55 MNP techniques, including aspects of the datasets, performance metrics, and replication packages.

5.1 Datasets

5.1.1 Overview of Datasets. Different datasets pose different challenges and tasks for MNP. Fig. 22(a) shows the distribution of programming languages in the dataset. It is observed that Java is the most popular language with 51 studies using Java. After Java, Python is the next most popular language with 8 studies using it. Fig. 22(b) displays the distribution of the number of study instances (i.e., methods or files) in the dataset. Most researchers explicitly count the number of methods or files in the dataset. Counting the number of methods is the dominant approach (50 studies provide the number of methods in the dataset). Some studies provide details on the number of methods or files used for each stage of model training.

Quantitatively, the number of methods used in MNP is concentrated in the interval of 100k~100m. Among them, there are 17 studies in the range of 100k~1m and 20 studies in the range of 10m~100m. These two intervals account for 67% of the total. Table 7 shows the dataset size for different study instances. It can be observed that most of the MNP studies have constructed larger datasets (> 1m).

Table 7. Dataset scale in different code granularities (i.e., method and file)

Scale	[10k,100k)	[100k,1m)	[1m,10m)	[10m,100m)	Unkown	Total
Method	4	17	9	20	0	50
File	1	0	1	0	3	5
Total	5	17	10	20	3	55

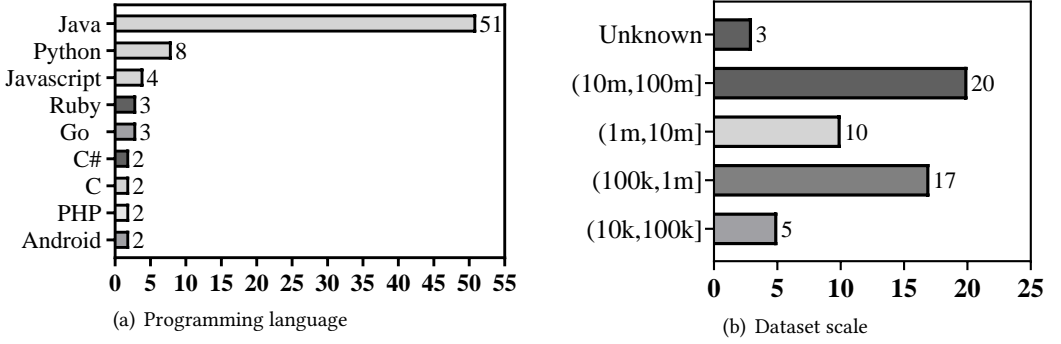


Fig. 22. Dataset language and scale

Table 8 overviews the different data sources. It is divided into several different categories: mainly from GitHub [8], Apache [101], Stanford **Open Graph Benchmark (OGB)** [54], CodeXGLUE[3], Google [101], and CodeSearchNet [121], etc. Among them, GitHub is the most popular, with 43 MNP papers using methods collected from GitHub, from which they mainly select some of the top-ranked and most popular projects. Additionally, Java-large, Java-small, and Java-med are used in many studies and are representative to a certain extent.

5.1.2 Dataset Comparison and Challenges. Table 9 presents a comprehensive comparison of existing datasets used for MNP, detailing their size, construction methodology, annotation strategies, and the key challenges they pose. Together, these tables cover both foundational benchmarks, such as Java-small, Java-med, and Java-large, and more recent datasets that incorporate enhanced structural, semantic, or contextual modeling. While Java-small, Java-med, and Java-large constitute the core benchmarks widely adopted in MNP research, ranging from small-scale validation to large-scale industrial evaluation, they are uniformly built on AST path representations and primarily capture intra-method syntactic information. For instance, Java-small contains approximately 0.7 million methods and serves to test model effectiveness on limited data, whereas Java-med and Java-large scale up to 4 million and 16 million methods respectively, providing broader evaluations of model scalability and generalization under project-level data splits. However, these datasets are inherently limited in their ability to capture cross-method dependencies, semantic abstractions, and interprocedural program behavior. To address these shortcomings, newer datasets have been proposed. CodeAttention employs MCGs to represent inter-method relationships. Flow2vec leverages LLVM-based pointer analysis to construct IVFG, thereby enhancing semantic flow representation. CoderPat integrates graph neural networks and additional structural edge types to capture long-range dependencies in source code. Moreover, concerns over the consistency and reliability of method name annotations in earlier datasets have led to the development of Mnire, which provides human-curated labels emphasizing naming consistency, and JavaRepos, which extracts real-world renaming instances from commit histories to better reflect practical development contexts. Other datasets such as CodeXGLUE and CodeSearchNet further extend the scope of evaluation by introducing multi-language, multi-task benchmarks. The datasets illustrate the evolution of MNP benchmarks from syntax-driven, AST-based modeling toward more semantically grounded, structurally diverse, and context-aware representations—ultimately supporting the development of models capable of deeper code understanding and more realistic naming prediction.

Table 8. A summary overview of the different data sources

Year	Name	Language	URL	Source	Number
2019	Java-small	Java	https://github.com/tech-srl/code2vec http://github.com/tech-srl/code2seq	GitHub	[8, 10, 13, 56, 82, 105] [50, 52, 71, 104, 121] [29, 51, 69, 97, 127] [14, 33, 81, 109]
2019	Java-med	Java		GitHub	[8, 10, 43, 101] [14, 50, 65, 81, 104, 127]
2019	Java-large	Java		GitHub	[8, 10, 19, 74, 101] [12, 50, 82, 104, 127] [11, 14, 29, 81]
2019	codeattention	Java	https://groups.inf.ed.ac.uk/cup/codeattention/	GitHub	[116]
2019	CodeSum2	Java	https://github.com/XuSihan/CodeSum2	GitHub	[113]
2019	CoderPat	Java,C#	https://github.com/CoderPat/structured-neural-summarization	GitHub	[28]
2019	—	Java	https://drive.google.com/file/d/1fNSmPluXbhQ9cfcAHkTtNrHID8Jkh2XD	GitHub	[31]
2020	Mnire	Java	https://sonvnguyen.github.io/mnire/#datasets	GitHub	[50, 52, 59, 104] [81, 115, 127]
2020	Flow2vec	C/C++	https://github.com/artifact4oopsla20/Flow2Vec?tab=readme-ov-file#21-code-classification-and-summarization-table-2	GitHub	[84]
2021	OGB	Python	https://ogb.stanford.edu/	Stanford OGB	[54]
2021	Meth2Seq	Java	https://meth2seq.github.io/meth2seq/	GitHub	[119]
2021	Mocktail Dataset	C	https://archive.org/download/mocktail-dataset-method-naming-tse	GitHub	[95]
2021	Python Summary Dataset	Python	https://github.com/EdinburghNLP/code-docstring-corpus	GitHub	[110]
2022	CodeSearchNet	Go, Java, JavaScript, PHP, Python, Ruby	https://github.com/github/CodeSearchNet	GitHub	[32, 66, 121, 128]
2022	Java2Graph	Java	https://github.com/kk-arman/graph_names/	GitHub	[68]
2022	CodeXGLUE	Go, Java, JavaScript, PHP, Python, Ruby	https://microsoft.github.io/CodeXGLUE/	GitHub	[3]
2023	JavaRepos	Java	https://github.com/TruX-DTF/debug-method-name	Apache, Spring, Hibernate, and Google	[53, 103]
2023	150k Python Dataset	Python	https://www.sri.inf.ethz.ch/py150	GitHub	[69]
2023	GitHub Java corpus	Java	https://groups.inf.ed.ac.uk/cup/javaGithub/	GitHub	[120]

5.2 Performance Metrics

In this Section, we will provide an overview of the metrics used in the MNP works and clarify the relevant definitions. Additionally, we will illustrate the differences between various metrics with concrete examples.

5.2.1 Overview of Metrics. Existing MNP studies employ various performance evaluation metrics as listed in Table 10 to measure the effectiveness of an MNP technique. In terms of performance evaluation, most of the studies mainly measure precision, recall, and F1, and some studies creatively use exact matching accuracy and so on.

- **Precision.** Precision refers to the proportion of correctly predicted method name tokens from the total number of predicted method name tokens. It measures the accuracy of the MNP technique.

- **Recall.** Recall is the proportion of correctly predicted method name tokens to the total number of tokens in the target method name. It measures the recall ability of the MNP technique.

- **F1.** The F1 is the harmonic mean of precision and recall and provides a comprehensive measure of system performance. A high F1 indicates a good balance between precision and recall.

Table 9. Dataset comparison and challenges

Dataset	Size	Motivation	Constructed Manner	Annotation	Challenges
Java-small	≈0.7M methods (≈11 projects)	Benchmark for code2vec/code2seq; validate AST-path learning on small data.	Method bodies from GitHub Java projects; converted to ASTs; extracted fixed-length AST-path-context triplets.	Original method names (tokenized).	Generalization on small, project-split data; potential over-reliance on lexical cues.
Java-med	≈4M methods (≈1000 projects)	Evaluate model scalability/performance on medium-scale data.	Method bodies from ≈1000 highly-starred GitHub Java projects; AST-path processing.	Original method names.	Scalability and generalization on medium-scale, project-split data.
Java-large	≈16M methods (≈9500 projects)	Evaluate model scalability/performance on very large-scale data for real-world applications.	Method bodies from ≈9500 highly-starred GitHub Java projects; AST-path processing.	Original method names.	High demands for scalability; rigorous project-split generalization; high computational resources.
codeattention	Unknown	Improve code comprehensibility via identifier recommendation; leverage inter-method call graph relationships.	Based on method call graphs. Limited construction details.	Original method names.	Models need to utilize inter-method call relationships effectively.
CodeSum2	≈200K	Improve method name suggestion by capturing code's hierarchical structure (tokens, basic blocks, snippets).	Collected from ≈10 open-source Java repositories; uses two-level attention for method representations.	Original method names.	Effectively capturing and utilizing hierarchical code structure.
CoderPat	≈1M	Overcome pure sequence model limitations in long-range dependencies by integrating GNNs.	Uses Java-small and new dataset from ≈23 C# projects. Represents code with subtokens, identifier/parse tree nodes, and LASTEXIC/ALUSE edges.	Original method names.	Effectively processing graph-structured data and capturing long-range dependencies.
MnIre	≈14.5M	Benchmark for MNP consistency/recommendation based on human-reviewed names; addresses imprecision.	Based on "naturalness" studies; generative approach treating MNP as abstract summarization of token contexts. Includes MCC/MNR corpora.	Emphasizes human-validated consistency.	Predicting unseen words (neologisms); focuses on "consistency"; reducing automated false positives.
Flow2vec	≈17K	Overcome limitations of existing code embeddings by preserving interprocedural program dependencies (value-flows).	Programs compiled to LLVM-IR; builds IVFGs using Andersen's pointer analysis; transformsto adjacency matrices.	Primarily for unsupervised code representation; MNP is a downstream task.	Capturing interprocedural semantics; handling alias problems; achieving context-sensitive representations.
OCB (ogbg-code2)	≈450K	Robust benchmark for ML on code; addresses duplication, low project count, unrealistic splitting of prior datasets.	Collects ASTs from ≈450K Python methods. Adds AST node features and "next-token" edges.	Original method names (tokenized).	Significant generalization gap from strict "project split"; effectively leveraging AST graph structure; need for better GNNs.
Meth2Seq	≈280K	Bridge code syntax and semantics gap by scalably capturing semantics.	Enodes methods as distributed vector sequences by combining PDG paths, typed IR statements, and natural language comments.	Original method names.	Learning from and fusing heterogeneous code representations (syntax semantics, natural language).
Mocktail Dataset	≈730K	Extends code2vec by including semantic graphs (CFG, PDG alongside ASTs for comprehensive feature capture.	Integrates CFG and PDG path info on top of ASTs. Evaluated on custom C dataset (≈730K methods, ≈16 C projects).	Original method names.	Integrating multi-modal code representations (AST, CFG, PDG); capturing more comprehensive features.
Python Summary Dataset	Unknown	For automated code documentation/generation; parallel corpus of code/natural language.	Scraped GitHub; preprocessed Python code to extract function declarations, docstrings, and bodies. Provides, "repo_split" version.	Function names and docstrings.	Adapting code-to-text mapping for MNP; leveraging docstrings as rich semantic context.
CodeSearchNet	≈700K	Foster ML/NLP research on code-NL relationships; provide datasets, tools, benchmarks for semantic code retrieval.	Collects comment, code pairs (top-level functions); strict repository-based splitting. *AdvTest normalizes names.	Function/method comments and corresponding code.	High semantic understanding; large-scale, multi-language data processing.
Java2Graph	≈16K	Implied: graph-based naming for Java code.	URL, inaccessible; unknown methods.	Unknown.	Data quality/details unknown; restricted access; graph representation learning challenges.
CodeXGLUE	≈1M	Foster ML research in program understanding/generation; diversified benchmark suite with standardized protocols.	Include a collection of code intelligence tasks and a platform for model evaluation and comparison; ≈10 tasks.	Varies by sub-task.	Task/language diversity requires adaptable generalizable, models.
JavaRepos	≈2M	Support automated method name consistency/refactoring; address impracticability of pre-implementation naming.	Clones GitHub Java repos; collects renamed methods from commit history. Resource-intensive data prep.	Based on method renaming events in commit history.	Supports pre-implementation naming; handling real-world codebase inconsistencies.
150K Python Dataset	≈150K files	Part of "ML for Programming"; create new tools via ML from large codebases.	Collects GitHub Python programs; rigorous cleaning (dedupe, forks, obfuscation); retains parsed programs ≈30K AST nodes.	Original method names; primarily provides AST structure.	Effectively leveraging structured AST for MNP; importance of high-quality non-redundant data.
GitHub Java corpus	≈14K projects	Large, high-quality corpus for studying large-scale Java coding practices.	Projects filtered by fork count for quality; strict deduplication; 75%/25% train/test split by lines of code.	Original method names.	Immense scale demands high model scalability/generalization; meticulous deduplication/project-splitting ensure realistic evaluation.

Table 10. Performance metrics used by MNP studies

Metric	Range	Study	Number
Precision Recall F1	[0, 1]	[8, 10, 13, 19, 43, 59, 74, 95, 101, 105, 113] [50, 52, 54, 56, 68, 82, 104, 115, 119, 121] [3, 4, 12, 14, 35, 51, 65, 69, 72, 103, 109, 127] [6, 11, 28, 29, 32, 53, 66, 81, 84, 92, 97, 118, 120, 128]	47
EM	0,1	[6, 9, 31, 50, 52, 59, 81, 97, 113, 115, 118, 119, 127]	13
ROUGE	[0,1]	[28, 33, 71, 105, 110]	5
BLEU	[0,1]	[31, 110]	2
ED	[0, +∞)	[31, 97]	2
Accuracy	[0,1]	[29, 35, 53]	3
AED RED	[0, +∞)	[35]	1
METEOR	[0,1]	[110]	1

In the evaluation process, the method name ground-truth (N_g) and the predicted method name (N_p) are treated as a pair, and the following calculations are performed:

$$Precision = \frac{|token(N_g) \cap token(N_p)|}{|token(N_p)|} \quad (1)$$

$$Recall = \frac{|token(N_g) \cap token(N_p)|}{|token(N_g)|} \quad (2)$$

$$F1 = 2 \times \frac{Precision \times Recall}{Precision + Recall} \quad (3)$$

where $token(\cdot)$ is a function that returns the (sub)tokens in the method name \cdot , and the absolute value symbol $|\cdot|$ returns the number of tokens. Similar to F1, the Modified-F1 (F1**) score is also used to evaluate method names. The Modified-F1 (F1**) score is calculated from the modified unigram precision and unigram recall. This prevents models that repeatedly output subwords in the gold function name from receiving unreasonably high scores.

• **Exact match (EM).** The EM is whether the model's predictions exactly match the true results. For tasks such as sequence annotation, each annotation is required to be correct to be considered a match. Specifically, if the method name predicts N_p matches the method name ground-truth N_g exactly, the exact matching degree is 1, otherwise 0.

$$EM = \begin{cases} 1, & \text{if } N_g == N_p \\ 0, & \text{otherwise} \end{cases} \quad (4)$$

• **ROUGE.** ROUGE is a set of metrics for evaluating text summarization systems. ROUGE scores typically include variants such as ROUGE-N and ROUGE-L, where N is the size of the n-gram (n consecutive words), and L is the size of the **Longest Common Subsequence(LCS)**. ROUGE-N evaluates the performance of a model by comparing the similarity of the generated summaries to a reference summary.

$$ROUGE-N = \frac{|token_n(N_g) \cap token_n(N_p)|}{|token_n(N_g)|} \quad (5)$$

where n stands for the length of the n-gram, n -gram and $|token_n(N_g) \cap token_n(N_p)|$ is the maximum number of n-grams co-occurring in a candidate summary and a set of reference summaries.

ROUGE-L ROUGE-N focuses on the overlap of n-grams to measure the similarity between generated text and reference text at the phrase or n-gram level, while ROUGE-L emphasizes the LCS to evaluate the coherence and order of the text. Its calculation formula is as follows:

$$R_{lcs} = \frac{LCS(N_g, N_p)}{|token(N_g)|} \quad (6)$$

$$P_{lcs} = \frac{LCS(N_g, N_p)}{|token(N_p)|} \quad (7)$$

$$F_{lcs} = \frac{(1 + \beta^2)R_{lcs}P_{lcs}}{R_{lcs} + \beta^2P_{lcs}} \quad (8)$$

where $LCS(N_g, N_p)$ is the length of the LCS of N_g and N_p . R_{lcs} and P_{lcs} denote recall and precision. The last F_{lcs} is ROUGE-L. β is a parameter that adjusts the relative importance of precision and recall. If β is set to 1, the F_{lcs} represents the harmonic mean of precision and recall, and its calculation is the same as that of the F1.

• **BLEU**. BLEU is a commonly used metric in machine translation tasks to measure the accuracy of translated candidate sentences against reference sentences. It compares how many words in the target sequence appear in the decoded sequence. In the MNP task, the predicted method names and the actual method names are treated as candidate sentences and reference sentences, respectively. The score is calculated as follows:

$$BLEU = BP \cdot \exp \left(\sum_{n=1}^N w_n \log(N_p(n)) \right) \quad (9)$$

where $N_p(n)$ is the modified n-gram precision calculated by dividing the clipped count of n-grams in the candidate sentence by the count of n-grams in the reference sentence.

To reduce the bias for short candidate sentences, a brevity penalty (BP) is introduced:

$$BP = \begin{cases} 1 & \text{if } |N_p| > |N_g| \\ e^{(1 - |N_g|/|N_p|)} & \text{if } |N_p| \leq |N_g| \end{cases} \quad (10)$$

It calculates the 1-gram, 2-gram, 3-gram, and 4-gram matching accuracy between candidate and reference sequences.

• **ED**. ED is a commonly used metric for measuring the difference between two strings, calculating the minimum number of operations required to transform the generated string into the target string. In the MNP task, ED is used to compare the differences between the predicted and ground truth method names, counting the necessary insertions, deletions, and single-character substitutions.

• **Metric for Evaluation of Translation with Explicit Ordering (METEOR)**. METEOR is a commonly used metric for evaluating machine translation quality. It measures translation accuracy and completeness by combining the harmonic mean of precision and recall. METEOR not only focuses on word-level matches but also includes phrase matching and accounts for variations in word forms (such as stemming and lemmatization) and word order issues, providing a more comprehensive evaluation of translation quality compared to traditional metrics like BLEU. In the MNP task, the predicted method names and ground truth names correspond to the translated output and reference answers, respectively.

$$F_{mean} = \frac{10 \cdot Precision \cdot Recall}{Recall + 9 \cdot Precision} \quad (11)$$

$$Penalty = 0.5 \left(\frac{\#chunks}{\#unigrams_matched} \right)^3 \quad (12)$$

$$METEOR_Score = F_{mean} \cdot (1 - Penalty) \quad (13)$$

where *unigram* refers to the smallest unit of a single word and typically does not require special configuration. METEOR first calculates the harmonic mean F_{mean} of precision and recall. It then adjusts this value based on the number of matched *unigrams* and *chunks*, resulting in a final score that reflects both precision and recall. In practical applications, the length of *chunks* is usually set between 1 and 4 words.

- **Accuracy.** Accuracy is a metric used to assess the performance of classification models, indicating the proportion of correct predictions among all predictions made. In the MNP task, accuracy measures the proportion of generated method names that are identical to the ground truth method names. It is calculated by dividing the number of correctly generated method names by the total number of test instances.

$$Accuracy = \frac{Number\ of\ EM\ Predictions}{Total\ Number\ of\ Predictions} \quad (14)$$

- **AED and Relative Edit Distance (RED).** AED is a metric for measuring edit operations, used to calculate the average number of edits required to transform the generated name into the target name. RED is used in method name consistency detection scenarios, and, compared to absolute edit distance, RED better reflects the relative differences and improvements between the newly generated name and the original name.

Among the metrics mentioned earlier, AED and RED are similar to the Accuracy metric in that they are calculated over the entire test set. In contrast, other metrics are computed for individual method names, with the dataset-level metrics obtained by averaging the results for each method name. Specifically, for a given test set with N samples, the AED and RED of an approach are defined as follows:

$$AED = \frac{1}{N} \sum_{k=1}^N ED(N_p^{(k)}, N_g^{(k)}) \quad (15)$$

$$RED = \frac{1}{N} \sum_{k=1}^N \frac{ED(N_p^{(k)}, N_g^{(k)})}{ED(N_{orig}^{(k)}, N_g^{(k)})} \quad (16)$$

where $ED(\cdot)$ is the edit distance function, N_p is the generated method name, N_{orig} is the original method name that has not been modified before consistency checking, and N_g is the ground truth method name.

5.2.2 Metric Limitations and Developer Alignment. Many standard metrics in MNP fall into three families: token overlap, ngram overlap, and edit-based, and each family has characteristic blind spots regarding semantic clarity, naming conventions, and readability.

Token-Overlap Metrics (Precision, Recall, F1, EM, Accuracy). These metrics quantify subtoken-level matches but overlook critical aspects such as order, structure, and semantics. For instance, a prediction like “removeFile” compared to the reference “deleteFile” receives a score of 0 from EM and Accuracy, while Precision, Recall, and F1 yield low scores, even though the two are semantically equivalent. EM tends to significantly undervalue predictions that differ in surface form but convey the same meaning [98], and F1 has been shown to penalize the use of synonyms, which is conceptually flawed. Moreover, these metrics fail to account for naming conventions such

as the verb-first pattern. For example, “fileRemove” receives the same F1 as “removeFile” even though only the latter follows standard naming practices.

N-gram Overlap Metrics (BLEU, ROUGE, METEOR). Originally developed for machine translation and summarization tasks, n-gram overlap metrics such as BLEU and ROUGE capture partial word order but still fall short of true semantic understanding. For instance, they penalize token order variations like “parseJson” and “jsonParse”, even though both are semantically equivalent to developers. Moreover, BLEU and ROUGE are unstable for short sequences such as method names, where score differences often fail to reflect meaningful qualitative distinctions [78]. METEOR attempts to mitigate some of these issues by incorporating synonym and stem matching, for example, recognizing “delete” and “remove” as similar terms. However, its effectiveness is limited when applied to code, as its dictionary often lacks API-specific vocabulary such as “getUserData” or “fetchUserInfo”, which can be semantically equivalent but lexically different.

Edit-Based Metrics (ED, AED, RED). These metrics measure minimal edits but fail to account for semantic equivalence and naming conventions. For example, “countLines” and “linesCount” receive the same penalty as “countLines” and “countCats”, even though only the latter represents a true change in meaning. ED tends to over-penalize lexically different but semantically equivalent names, and even RED still ignores important naming conventions such as the verb-first pattern.

All of these metrics offer value but exhibit biases that prevent a comprehensive assessment of developers’ perceived method-name quality. We therefore suggest exploring a multi-faceted evaluation strategy that combines F1, METEOR, and RED to cover lexical, semantic, and structural dimensions. This strategy could be augmented with an embedding-based semantic similarity metric to better capture clarity of intent and further extended to analyze naming-style consistency by measuring verb-first rate and camelCase adherence, thereby reflecting readability and convention compliance. Additionally, incorporating developer judgments on a sample of predictions, by having them rate clarity, conformity, and acceptability, can help align automated metrics with human perception. To complement these methods, an LLM may be employed as an auxiliary evaluator to generate natural-language scores and critiques that emulate developer assessments of semantic precision and naming intent, and to flag candidates for focused review. By integrating automated metrics, LLM-based evaluation, and human cognitive factors, future approaches may more accurately capture a model’s practical utility and acceptance in real-world coding scenarios.

5.2.3 Metrics Examples.

Providing examples is very helpful in clarifying the metrics. We will present examples to visually demonstrate the calculation process and practical application of each metric, thereby aiding in the understanding of their actual significance and use.

Assuming the dataset contains two records: for the first record, the ground truth method name $N_g^{(1)}$ is “calculateArraySum” and our predicted method name $N_p^{(1)}$ is “calculateSum”; for the second record, the ground truth method name $N_g^{(2)}$ is “calculateTotalSumforArrayElements” and our predicted method name $N_p^{(2)}$ is “calculateSumforArrayElements.” We evaluate the performance of the predictions by calculating various metrics.

To calculate the metrics, method names need to be split into individual tokens and converted to lowercase. For names using snake case, underscores should be removed. Table 11 presents some important intermediate results in the calculation of the metrics. For example, from the table, we see that $|token(N_g^{(1)})| = 3$, $|token(N_p^{(1)})| = 2$, and $|token(N_p^{(1)}) \cap token(N_g^{(1)})| = 2$. Using Formulas 1 and 2, precision $P = 1$ and recall $R = 0.66$ can be easily calculated, leading to an F1 of 0.8.

Table 11. Metric calculation details for examples

$Record_1$	$Record_2$
$N_g^{(1)} = \text{"calculateArraySum"}$	$N_g^{(2)} = \text{"calculateTotalSumforArrayElements"}$
$N_p^{(1)} = \text{"calculateSum"}$	$N_p^{(2)} = \text{"calculateSumforArrayElements"}$
$token(N_g^{(1)}) = \{\text{'calculate'}, \text{'Array'}, \text{'Sum'}\}$	$token(N_g^{(2)}) = \{\text{'calculate'}, \text{'total'}, \text{'sum'}, \text{'for'}, \text{'array'}, \text{'elements'}\}$
$ token(N_g^{(1)}) = 3$	$ token(N_g^{(2)}) = 6$
$token(N_p^{(1)}) = \{\text{'calculate'}, \text{'sum'}\}$	$token(N_p^{(2)}) = \{\text{'calculate'}, \text{'Sum'}, \text{'for'}, \text{'array'}, \text{'elements'}\}$
$ token(N_p^{(1)}) = 2$	$ token(N_p^{(2)}) = 5$
$token(N_p^{(1)}) \cap token(N_g^{(1)}) = \{\text{'calculate'}, \text{'sum'}\}$	$token(N_p^{(2)}) \cap token(N_g^{(2)}) = \{\text{'calculate'}, \text{'sum'}, \text{'for'}, \text{'array'}, \text{'elements'}\}$
$ token(N_p^{(1)}) \cap token(N_g^{(1)}) = 2$	$ token(N_p^{(2)}) \cap token(N_g^{(2)}) = 5$
$token_2(N_g^{(1)}) = \{\text{'calculate array'}, \text{'array sum'}\}$	$token_2(N_g^{(2)}) = \{\text{'calculate total'}, \text{'total sum'}, \text{'sum for'}, \text{'for array'}, \text{'array elements'}\}$
$ token_2(N_g^{(1)}) = 2$	$ token_2(N_g^{(2)}) = 5$
$token_2(N_p^{(1)}) = \{\text{'calculate sum'}\}$	$token_2(N_p^{(2)}) = \{\text{'calculate sum'}, \text{'sum for'}, \text{'for array'}, \text{'array elements'}\}$
$ token_2(N_p^{(1)}) = 1$	$ token_2(N_p^{(2)}) = 4$
$LCS(N_p^{(1)}, N_g^{(1)}) = \text{'calculate sum'}$	$LCS(N_p^{(2)}, N_g^{(2)}) = \text{'calculate sum for array elements'}$
$ LCS(N_p^{(1)}, N_g^{(1)}) = 2$	$ LCS(N_p^{(2)}, N_g^{(2)}) = 5$
$ N_g^{(1)} = 2$	$ N_g^{(2)} = 6$

Table 12. Metric values for two examples

Metric	Precision	Recall	F1	EM	ROUGE-L	BLEU	ED	Accuracy	AED	RED	METEOR
Range	[0,1]	[0,1]	[0,1]	{0,1}	[0,1]	[0,1]	[0, +∞)	[0,1]	[0, +∞)	[0, +∞)	[0,1]
$Record_1$	1	0.66	0.80	0	0.80	0.61	5	0	5	0.42	0.34
$Record_2$	1	0.83	0.91	0	0.91	0.58	5	0	5	0.42	0.82

In other examples, such as "calculateSum" and "calculateArraySum", the names are not identical, resulting in an EM score of 0. The method name "calculateSum" requires the insertion of 5 characters, "Array" to transform into the target name "calculateArraySum", so the ED between them is 5. Similarly, the edit distance between "calculateSumforArrayElements" and "calculateTotalSumforArrayElements" is also 5, so the AED of these two records is 5. In [35], RED is used for checking name consistency. Assuming the original name is "calculate", the edit distance between "calculate" and "calculateArraySum" is 5, while the edit distance between "calculate" and "calculateSumforArrayElements" is 24, resulting in a RED value of approximately 0.42. We present the calculated results of the examples in Table 12. To provide a clear visual representation of each metric, we have listed the value ranges for each metric in the first row to facilitate intuitive comparison.

5.3 Replication Packages

In this section, we explore how often the 55 reviewed MNP techniques share replication packages in their papers. Replicability refers to the ability of other researchers to obtain the same results using the artifacts provided by the authors. Reproducibility, however, involves obtaining the same results using generally the same methods, but not necessarily the original artifacts. Both replicability and reproducibility are crucial for assessing the quality of original research and validating its credibility, as they enhance our confidence in the results and help us distinguish between reliable and unreliable findings. This is particularly important for learning-based methods, whose performance often heavily depends on factors such as datasets, data processing methods, and hyperparameter settings.

Table 13. Replication package links provided in MNP studies

Year	Technique	URL	Study
2015	-	https://groups.inf.ed.ac.uk/cup/naturalize/#team	[4]
2016	-	https://groups.inf.ed.ac.uk/cup/codeattention/	[6]
2019	-	https://github.com/SerVal-DTF/debug-method-name	[53]
2019	-	https://github.com/CoderPat/structured-neural-summarization	[28]
2019	Code2vec	https://github.com/tech-srl/code2vec	[10, 101]
		https://github.com/basedrhys/obfuscated-code2vec	[19]
		https://github.com/mdrafiqulrabin/handcrafted-embeddings	[74]
		https://cazzola.di.unimi.it/fold2vec.html	[51]
		https://github.com/NobleMathews/mocktail-blend	[95]
2019	Code2seq	https://github.com/tech-srl/code2seq	[8]
2019	Mercem	https://groups.inf.ed.ac.uk/cup/codeattention	[116]
2019	HeMA	https://github.com/Method-Name-Recommendation/HeMa	[43]
2019	HAN	https://github.com/XuSihan/CodeSum2	[113]
2020	GINN	https://github.com/GINN-lmp/GINN	[105]
2020	Mnire	https://doubledoubleblind.github.io/mnire	[59]
2020	Flow2vec	https://github.com/artifact4oops/la20/Flow2Vec?tab=readme-ov-file#21-code-classification-and-summarization-table-2	[84]
2021	TPTrans	https://github.com/AwdHanPeng/TPTrans	[66]
2021	InferCode	https://github.com/bdqngghi/infercode	[13]
2021	-	https://github.com/css518/Keywords-Guided-Method-Name-Generation	[33]
2021	-	https://www.daml.in.tum.de/code-transformer	[128]
2021	-	https://github.com/mdrafiqulrabin/tnpa-generalizability	[72]
2021	TreeCaps	https://github.com/bdqngghi/treecaps	[14]
2021	PSIMiner	https://github.com/JetBrains-Research/code2seq	[82]
2021	Meth2Seq	https://meth2seq.github.io/meth2seq/	[119]
2021	UniCoRN	https://github.com/dmlc/dgl/tree/master/examples/pytorch/rgcn-hetero	[54]
2021	Cognac	https://github.com/ShangwenWang/Cognac	[104]
2021	DeepName	https://github.com/deepname2021icse/DeepName-2021-ICSE	[50]
2022	GraphCode2Vec	https://github.com/graphcode2vec/graphcode2vec	[56]
2022	HGT	https://github.com/IBM/Project_CodeNet/issues/29	[121]
2022	GNNs	https://github.com/kk-arman/graph_names	[68]
2022	NamPat	https://github.com/cqu-isse/NamPat	[115]
2022	SGMNG	https://github.com/QZH-eng/SGMNG	[71]
2022	CodeBERT	https://github.com/microsoft/CodeBERT/tree/master/GraphCodeBERT/code2search	[3]
2022	GTNM	https://github.com/LiuFang816/GTNM	[52]
2022	PCAN	https://github.com/HongliangLiang/pcan	[109]
2022	-	https://github.com/ceragoguztuzun/MethodNameGeneration	[97]
2023	SENSA	https://github.com/mzakeri/SENSA	[65]
2023	Fold2Vec	https://cazzola.di.unimi.it/fold2vec.html	[12]
2023	Mario	https://github.com/ShangwenWang/Mario	[103]
2023	AUMENA	https://figshare.com/s/0382ba979d970b4c2b2	[127]
2023	-	https://github.com/software-theorem/mnp	[69]
2023	Sub-quadratic Transformer	https://cazzola.di.unimi.it/comb-transformer.html	[11]
2023	-	https://github.com/APIDocEnrich/Renaming	[120]
2023	CREAM	https://github.com/ReliableCoding/CREAM	[32]
2024	DMNA	https://github.com/software-theorem/mnp	[81]

As mentioned in Section 3.5, we consider replicability and reproducibility as significant factors in evaluating research quality. We begin by systematically reviewing each paper to ensure that the provided code links are publicly accessible and valid, and we record the availability of each link. Next, we verify whether the datasets used in the papers are sourced from open repositories; if they are private or require special permissions, we mark these as exceptions. We then assess whether the papers include all necessary components and environments for replication, such as experimental setups, dependencies, and configuration files. Additionally, we evaluate whether the papers provide sufficient details to support replication, including experimental methods, parameter settings, and data processing steps. Finally, we compile all this information, calculate the compliance of each paper with the standards, and summarize the results. Based on the statistical and summary results, we have created Fig. 23 to present the data visually. To facilitate future MNP studies, we provide the usable replication packages in Table 13.

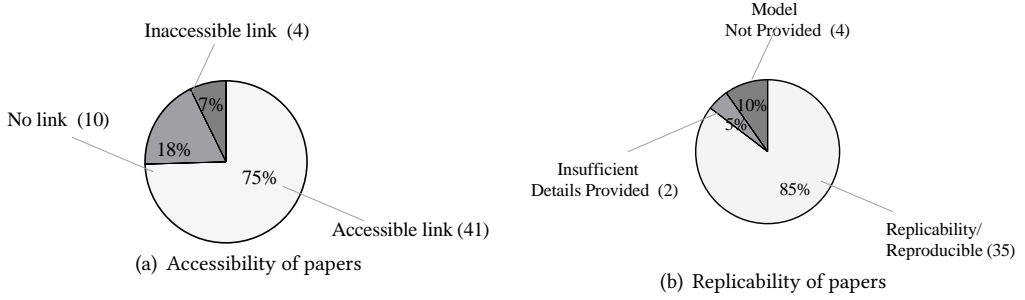


Fig. 23. Analysis of paper accessibility and replicability

As shown in Fig. 23(a), 10 papers, accounting for approximately 18%, did not provide links to the research results. Among the 41 papers that did provide links, the links to 4 papers are no longer valid, representing about 7%. The remaining 41 papers offered valid links, and we further analyzed their replicability. As depicted in Fig. 23(b), all 41 papers explicitly provided specific datasets. However, 4 papers did not provide model files, and 2 papers lacked sufficient details, corresponding to approximately 10% and 5%, respectively. The remaining 35 papers also included detailed documentation.

Summary ► The evaluation of MNP techniques has evolved from reliance on syntax-driven benchmarks toward semantically enriched assessments, underpinned by large-scale Java datasets sourced from public repositories such as GitHub. Despite the continued use of standard metrics like Precision, Recall, and F1, their limitations in reflecting semantic equivalence have catalyzed a shift toward more comprehensive, developer-oriented evaluation paradigms. Reproducibility is showing signs of improvement, with 35 of the 55 surveyed studies (approximately 64%) providing complete and publicly available replication packages. ◀

6 ANSWER TO RQ3: WHAT DO EXISTING EMPIRICAL STUDIES ON LEARNING-BASED MNP CONCERN?

Although MNP is an emerging research area in SE, a variety of learning-based MNP techniques have been successively proposed and continuously achieved promising results in terms of the accuracy of predicted names over the past six years. In addition to developing new MNP techniques that address technical challenges, the learning-based MNP research field is benefiting from several empirical studies. These empirical studies systematically explore the impact of various components (e.g., AST representation), providing insights into future learning-based MNP work. We summarize existing empirical studies in Table 14 and discuss them in detail as follows.

6.1 Generalization Performance of Models

Jiang et al. [43] conduct an ES on code2vec, exploring its performance across various datasets and investigating the underlying reasons for both its successes and failures. Specifically, they investigate the following research questions: 1) How well does code2vec work on datasets other than the one employed by the original evaluation conducted by the authors of code2vec? They evaluate code2vec on a new dataset as well as its original dataset employed in paper [10]. The results show that code2vec overall is accurate, and switching to a new large-scale dataset does not result in a significant reduction in the performance of code2vec. 2) How well does code2vec

Table 14. A summary and comparison of empirical studies in learning-based MNP

Year	Study	Language	Main Research Content (Research Questions)
2019	Jiang et al. [43]	Java	1) How well does code2vec work on datasets other than the one employed by the original evaluation conducted by the authors of code2vec? 2) How well does code2vec work with more realistic settings? 3) Can code2vec generate method names correctly when the given method bodies do not contain method name tokens? 4) Where and why does code2vec work? 5) Where and why does code2vec fail? 6) Is code2vec useful for developers? How often does code2vec recommend correctly for methods that are challenging to name manually?
2020	Rabin et al. [74]	Java	1)How does the performance of SVM models built using code2vec embeddings compare to SVM models trained on handcrafted features for MNP?
2020	Zaitsev et al. [118]	Pharo	1)Which model performs better in the MNP task, the extractive model based on Term Frequency Inverse Document Frequency (TF-IDF) with n-grams or the abstractive model based on sequence-to-sequence neural networks?
2021	Rabin et al. [72]	Java	1) How do the transformations impact the predictions of neural program models in the single-place transformed dataset? 2) When do the transformations affect neural program models the most? 3) How does the method length impact the generalizability of neural program models? 4) What are the trends in types of changes? 5) How do the transformations affect the precision, recall and F1 of the neural program models?
2022	Varner et al. [97]	Java	1)Which model performs better in the MNP, the RNN encoder-decoder model with attention or the Transformer model?
2023	Wang et al. [103]	Java	1) How pervasive are proximate class pairs? 2) How is the relation between the field-irrelevant method names of a class and those of its proximate classes? 3) How is the similarity between field-relevant method names of a class and those of its proximate classes with respect to its non-unique fields? 4) How is the effectiveness of Mario? 5) How is the effectiveness of Mario on field-relevant method names and field-irrelevant method names respectively?
2023	Li et al. [51]	Java	1)How ASTs generated from different parsing tools could affect the performance of ML models depending on an input of code structure representation.
2023	Qian et al. [69]	Java, Python	1) Does AST improve the performance of MNP? 2) How do AST parsing methods affect the performance of MNP? 3) How do AST preprocessing methods affect the performance of MNP? 4) How do AST encoding methods affect the performance of MNP?

work with more realistic settings? They evaluate code2vec on a new dataset with different settings, i.e., file-based validation, project-based validation, and project-based non-overriding validation. The results demonstrate that the performance of code2vec decreases significantly in more realistic settings. However, its overall performance is still promising. 3) Can code2vec generate method names correctly when the given method bodies do not contain method name tokens? They evaluate the performance of code2vec in generating seen and unseen method name tokens, respectively. For each of the method names in the new dataset, they split it into tokens according to the Camel-Case naming convention. A token t from method name mn is a seen token if t appears (case insensitive) in the body named by mn . Otherwise, it is an unseen token. They assess how often code2vec can successfully recommend seen/unseen method name tokens during project-based non-overriding validation. They conclude that code2vec works well in generating unseen method name tokens. 4) Where and why does code2vec work? They conduct another manual analysis on successful cases.

They observe that code2vec works well on getter/setter methods and delegations. The rationale for the success is that such methods have common structures in their ASTs. 5) Where and why does code2vec fail? They conduct another manual analysis on one thousand failed cases. They identify the main reasons for the failure as follows: First, OOV method names are the primary for the failure. A method name is OOV if and only if none of the methods in the training set is named with it. As a result, code2vec often fails. Using open vocabularies or minting method names by combining tokens may help to further improve performance. The second reason is that the exploited method features are often insufficient to infer method names. To further improve performance, additional information should be utilized, e.g., their enclosing classes. The third reason is improper method names in the testing set. During the evaluation, all recommended names are compared against the original names associated with testing methods, i.e., the code2vec fails if the recommended name is different from the original one.

Rabin et al. [74] conduct an ES to have a better understanding of the contents of code2vec neural source code embeddings. They use code2vec embeddings to create binary SVM classifiers and compare their performance with the handcrafted features on the MNP task. The results of a small case study suggest that the handcrafted features can perform very close to the highly-dimensional code2vec embeddings, and the information gains are more evenly distributed in the code2vec embeddings compared to the handcrafted features. They also find that the code2vec embeddings are more resilient to the removal of dimensions with low information gains than the handcrafted features.

In [72], Rabin et al. explore how semantic-preserving transformations affect the generalizability of neural program models. They assess the performance of code2vec, code2seq, and GGNN on three Java datasets, evaluating model predictions before and after transformations. The study reveals that GGNN is more sensitive to changes caused by semantic-preserving transformations compared to code2vec and code2seq. The latter models exhibit less variation in prediction outcomes with increased dataset sizes. Additionally, all-place transformations tend to affect code2vec and code2seq more than single-place transformations, while GGNN remains relatively stable. The study also finds that code2vec and code2seq perform better with longer methods, whereas GGNN's performance is less influenced by method length. Transformations generally reduce model performance, with code2vec and code2seq being more affected but compensating with larger datasets. Although semantic-preserving transformations impact classic metrics such as precision, recall, and F1, these changes do not correlate with the new metric, **Prediction Change Percentage (PCP)**.

6.2 Role and Impact of Components

AST. Li et al. [51] investigate the source code engineering impacts (mainly AST parser) towards ML-based software services (mainly MNP task). They evaluate AST parser generators towards the impacts on the prediction model of code2vec for the prediction task of the method name in Java language. Like Utkin et al. [94], they also utilize five metrics: TS, TD, BF, UTP, and UTK, to compare the ASTs generated by five AST parsers: JavaParser, ANTLR, Tree-sitter, Guntree, and Javalang. Their experimental results on the Java-small dataset show that ASTs generated using different parsing parsers differ greatly in terms of source code structures and contents. This difference could significantly influence the performance of the trained model code2vec.

Qian et al. [69] also believe that it is a complex process to manipulate AST, including AST parsing, AST preprocessing, and AST encoding, of which a change in the scheme may change the AST embeddings and thus affect the performance of MNP. Therefore, they aim to dive deep into the problem and answer: *Does AST help to improve MNP? How do AST parsing/preprocessing/encoding methods affect MNP?* To achieve this, they conduct a comprehensive ES to systematically investigate the impact of the sub-processes of AST usage on MNP performance. Specifically, they first depict

the workflow of the AST processing and usage in MNP. The utilization of AST involves intricacies encompassing three crucial sub-processes: AST parsing, AST preprocessing, and AST encoding. Then, they carry out experiments on two popular programming datasets, i.e., Java-small [4] and 150k Python Dataset [76]. The experiments involving four AST parsing methods: JDT, Javaparser, Javalang, and Tree-sitter; three AST preprocessing methods: original AST, AST path [9], and program graph [5]; and five AST encoding methods: BiLSTM, code2vec, code2seq, Tree-LSTM, and GGNN [5]. Finally, they obtain the following major findings about the current AST for MNP: 1) AST can indeed improve MNP; The strategic combination of path information within AST and token information from the code can notably influence MNP performance. 2) The selection of AST parsing methods bears a considerable influence on the overall performance of MNP. Currently, JDT stands out as the superior choice, outperforming Javaparser, Javalang, and Tree-sitter in promoting MNP. 3) AST preprocessing plays a pivotal role in the usage of AST, and a well-designed preprocessing scheme significantly influences MNP's efficacy. Currently, AST Path emerges as a particularly promising choice for MNP. 4) AST encoding undoubtedly warrants greater attention as it plays a crucial role in effectively translating ASTs into meaningful representations for MNP. Different AST preprocessing results necessitate distinct AST encoding networks or models to achieve optimal outcomes. Currently, code2seq performs best in promoting MNP for AST Path.

Model. In [97], Varner et al evaluate the Transformer and RNN model architectures for three distinct input types. Subsequently, the performance is evaluated across the six distinct experiments. The first input types employ solely the documentation tokens. The second input types utilize the enclosing class tokens, input parameter tokens, return type tokens, and body tokens, all concatenated together as a single input. The third input type is concatenating the input sequences employed in the initial experiment with the corresponding input sequences utilized in the second experiment. It is anticipated that the third experiment will yield the most favorable results. All of the contexts employed in the input were combined using the period token. The results demonstrate that a model utilizing all of the aforementioned contexts will exhibit superior performance compared to a model employing any subset of the contexts. Moreover, the study demonstrates that the Transformer model outperforms the RNN model in this scenario.


Zaitsev et al. [118] explore two approaches to take the MNP task: one based on TF-IDF [75] and the other using a deep RNN [91]. The first approach combines TF-IDF with an n-gram language model to perform extractive text summarization. This method identifies keywords from the method's source code and arranges them in a meaningful sequence. The second approach employs an attention-based sequence-to-sequence neural network for abstractive summarization, generating method names from words that do not appear in the source code. In this context, the method name serves as the generated summary, while the method body is the original text. The effectiveness of these approaches is assessed by comparing the generated method names to the actual names assigned by programmers, which are used as the benchmark.

6.3 Performance in MNP Practice

Jiang et al. [43] investigate how often code2vec works when it is strongly needed, specifically assessing its practical utility for developers. The investigation is conducted as follows: First, they invite six developers involved in a commercial project for the evaluation. The commercial project has been released recently by a giant of IT industry to conduct large-scale software refactorings. Second, each of the participants randomly selected one hundred methods developed by himself/herself. Third, they request all participants to score the difficulty in naming sampled methods. The scores rank between one and five (i.e. 5-point scale [46, 55]) where one represents the least difficulty and five represents the highest difficulty. Fourth, they apply code2vec of project-based non-overriding validation to the scored methods and validate the recommendations against manually constructed

names. They find that code2vec rarely works when it is strongly needed. Consider code2vec requires a large number of high-quality datasets, while the computational process consumes time and resources. To further investigate the possibility of designing simple alternative approaches, they propose a heuristics-based approach to recommending method names according to given method bodies. And their evaluation results show that it outperforms code2vec significantly, and improves precision and recall by 65.25% and 22.45%, respectively.

Wang et al. [103] find that existing literature approaches assume the availability of method implementation to infer its name. However, methods are named before their implementations. Therefore, they first introduce a new MNP scenario where the developer requires an MNP tool to predict names for all methods that are likely to be implemented within a newly defined class. They observe that for classes whose names are similar, the names of their methods, no matter whether they are related to the fields, may be similar to some extent. To validate this observation, they conduct a large-scale empirical analysis on 258K+ classes from real-world projects to validate their findings. To perform the empirical analysis, they analyze 258,321 classes and investigate three research questions: 1) How pervasive are proximate class pairs? 2) How is the relation between the field-irrelevant method names of a class and those of its proximate classes? 3) How is the similarity between field-relevant method names of a class and those of its proximate classes with respect to its non-unique fields? They conclude the main findings obtained through the empirical analysis as follows: [F1] Proximate class pairs are pervasive among real-world projects. [F2] A considerable percentage of tokens composing the names of a class's field-irrelevant methods can be found in the names of its proximate classes' field-irrelevant methods. [F3] The similarity between a class's method names of its non-unique fields and those of its proximate classes is extremely high. Then, supported by the empirical findings, they propose a hybrid big code-driven approach, namely Mario, to predict method names based on the class name. Experiment results demonstrate that Mario is effective in predicting method names and offers comparable or higher performance to code2seq that leverages implementation details.

 **Summary** ► Existing empirical studies on MNP primarily focus on the following aspects: 1) **Generalization Performance of Models.** Code2vec performs well on large-scale datasets, with embeddings showing similar performance to handcrafted features and greater resilience; 2) **Role and Impact of Components.** AST parsing and encoding methods are crucial for MNP performance, with the JDT parser and AST Path preprocessing showing the best results, while the Transformer model outperforms RNNs; 3) **Performance in MNP Practice.** The practical effectiveness of code2vec is limited. ◀

7 CHALLENGE AND OPPORTUNITIES

7.1 Challenges

7.1.1 Accurate understanding of the target method. For any MNP model, whether classification-based or generation-based, an accurate understanding of the functionality and intent of the target method is the prerequisite for generating a high-quality name. As detailed in our analysis for RQ1, all stages of the MNP workflow, context extraction, preprocessing, and encoding, are designed to facilitate this understanding. Consequently, the overarching challenge of accurately understanding the target method can be attributed to the inherent difficulties within these three consecutive processes.

Accurate Context Extraction. One key challenge in context extraction is the lack of adequate understanding of how individual and combined context sources contribute to the performance of MNP. The effects of specific contexts, such as sibling methods, are often not quantified, and their interaction can lead to redundancy or interference. Additionally, nontraditional sources such as

design documents, issue trackers, or execution traces remain underexplored despite their potential to provide rich semantic signals beyond static code.

Accurate Context Preprocessing. Preprocessing aims to simplify and enhance input representations, yet overly aggressive transformations risk removing essential semantic cues, such as meaningful distinctions between identifiers or structural nuances in ASTs. The effectiveness of many preprocessing techniques remains unvalidated, and the lack of reproducible, well-documented implementations hinders rigorous benchmarking and comparison across studies.

Accurate Context Encoding. Designing appropriate encoders involves trade-offs between specialized architectures tailored to specific code structures and general-purpose models like Transformers. The added value of architectural specialization is not always clear, especially given the strength of pre-trained models. Furthermore, ensuring that embedding spaces reflect semantic relationships relevant to naming remains a difficult task. High computational costs of advanced encoders also pose challenges for practical deployment, especially in resource-limited settings.

7.1.2 Rapid construction of high-quality training data. It is well known that the performance of neural network models depends on both the scale and quality of the training data, and MNP models are no exception. With the popularity of the open-source development paradigm, numerous open-source platforms (e.g., GitHub) now contain vast amounts of open-source code, making the construction of large-scale training datasets for MNP models no longer out of reach. However, unfortunately, not all code in open-source projects is of high quality. This makes the rapid construction of large-scale and high-quality training datasets quite challenging. As mentioned in Section 5.1, there are currently several widely used large-scale datasets for training and evaluating MNP models. However, the quality and diversity of these datasets have not been evaluated and verified. This challenge can be further attributed to the difficulties in assessing data quality and diversity.

Data Quality Assessment. Based on our comprehensive review of 55 MNP studies, we find that the field still lacks practical metrics for assessing dataset quality. The design of such metrics is challenging because there is not a strict one-to-one relationship between the method name and the method code body (such as overloaded methods). Furthermore, in addition to the local context contained in the method body itself, numerous MNP techniques also rely on some global contexts, such as callee methods and the class in which the target method appears. There is also a lack of practical, available metrics for evaluating the quality of these global contexts. Last but not least, relying on manual inspection for large datasets is time-consuming and impractical. How to devising automated QA methods based on metrics is also a challenging task.

Data Diversity Assessment. Likewise, we find no existing MNP studies that take sample diversity into account when constructing datasets. Sample diversity is crucial for developing effective, fair, and robust MNP models that can perform well in a variety of real-world MNP situations. Certainly, it is undeniable that assessing method diversity in MNP datasets is a challenging task. Accomplishing such a task requires considering the characteristics of the programming language. For example, there are various types of methods that can be defined in Java, such as instance methods, static methods, getter and setter methods, constructor Methods, and varargs methods. In Python, developers can define instance methods, class methods, dunder methods, etc. Additionally, assessing the diversity of the global contexts also poses a challenge due to their complexity.

7.1.3 Practical applications of MNP techniques. Existing MNP studies primarily focus on improving context extraction and preprocessing, as well as designing advanced MNP models. There is less attention to the practical implementation and application of MNP techniques/models in production environments. Creating usable and user-friendly MNP tools or systems is a highly challenging task. This challenge can equally be attributed to the difficulties in capturing context and achieving user-friendly interactions in a production environment.

Context Extraction in Production Environment. The actual software development environment is complex, and context extraction may involve (dynamic/static) program analysis techniques, such as control flow and data flow analysis, and function call analysis. Relying on developers to manually provide/extract complete context information is not user-friendly, especially for novice programmers or developers newly joining a project. How to automatically capture context information (including global context) in a production environment has not yet received sufficient attention and research.

Implementation of User-friendly Interaction. While numerous MNP models have been proposed, there is still a considerable distance to cover before these models can be practically implemented in engineering applications. The development of excellent programming assistance tools, including method name recommendation tools, is a complex, systematic engineering task. It not only requires developers to have a full stack of programming skills but also necessitates considerations for the actual user experience. How to implement efficient method name recommendation, and how to make it easier for developers to use have not yet received sufficient attention and research.

7.2 Opportunities

7.2.1 Addressing challenges mentioned in Section 7.1.

Typically, research challenges can also be viewed as opportunities from another perspective. Focusing on the challenges outlined in Section 7.1 and designing corresponding solutions provide many excellent research directions. Research results in these directions will significantly enhance the performance and practical usability of MNP.

7.2.2 Devising semantic-based performance metrics.

In Section 5.2, we have compiled commonly used performance metrics (e.g., *precision* and *ROUGE*) in existing MNP research. While these metrics, overall, contribute to the automated assessment of MNP techniques, they fundamentally calculate the textual similarity between generated method names and reference (or ground-truth) method names, and cannot evaluate their semantic similarity. It should be noted that, in real-world practice, developers with diverse programming/linguistic backgrounds are likely to employ distinct terminologies to convey identical meanings. In addition, variations in programming conventions and practices may result in the utilization of dissimilar abbreviation forms for the same set of terms. In short, developers can name the same method with totally different words. For instance, Feitelson et al. [26] find that the probability of two developers selecting the same name for the same method is only 6.9%. Furthermore, morphological problems (e.g., synonyms, abbreviations, and misspellings) in method names will affect the model train and test result. Even if the model recommends a better method name than the ground truth, existing metrics may also consider its prediction incorrect. In summary, we observe that there is still a lack of semantics-based performance metrics in this field. Attempting to propose and validate such metrics is undoubtedly a promising research opportunity/direction. The research outcomes in this direction would contribute to a more accurate evaluation of MNP performance, fostering further development in this field.

7.2.3 Adapting LLMs to MNP.

Recently, with the success of LLMs in NLP [22, 70], an increasing number of SE researchers have started integrating them into the resolution process of various SE tasks [17, 24, 40, 123], such as code generation [16, 99, 107], program repair [15, 44, 124], and code summarization [2, 25, 88, 89]. To the best of our systematic review of existing research, there is currently no study that investigates the performance of LLMs on MNP tasks. Certainly, investigating the performance of LLMs in MNP tasks and exploring the factors that affect their performance presents a valuable research opportunity.

In addition, similar to LLMs for NLP (e.g., ChatGPT [63] and LLaMA [93]) tasks, there are many LLMs of code for SE tasks, e.g., Codex [64], StarCoder [48], CodeGen [60], and PolyCoder [112]. How to automatically adapt general-purpose LLMs or dedicated code LLMs to MNP tasks is also indeed a promising research opportunity.

8 THREATS TO VALIDITY

8.1 Publication Bias

Publication bias refers to the tendency in academic publishing for researchers to preferentially publish positive results while overlooking or suppressing negative ones. This can lead to a skewed presentation of research findings on a particular topic in the literature. Therefore, if there is publication bias in the primary literature, the arguments supported or refuted by the prior studies selected for this paper may also be biased. To mitigate this threat, we conduct extensive searches across multiple databases and literature repositories to ensure the comprehensive inclusion of all relevant studies and results. Additionally, we cross-validate the findings across different studies to enhance the reliability and validity of our analysis.

8.2 Search Terms

The selection and use of search terms are crucial for comprehensively retrieving all literature relevant to the research topic. We review titles, abstracts, and keywords from several known relevant papers to manually construct search strings and test their applicability. We also utilize popular search databases to retrieve all relevant papers and employ forward and backward snowballing methods. We also consult with authors of relevant papers to ensure any potentially missed articles are identified.

8.3 Study Selection Bias

Study selection bias arises from subjective or objective factors that may skew the selection process, potentially affecting the representativeness and comprehensiveness of the final chosen studies. To mitigate this validity threat, each selected paper undergoes scrutiny by at least two researchers. In cases where consensus cannot be reached between two researchers regarding a particular study, a third researcher is consulted for discussion. This selection process is iterated until researchers achieve a complete consensus on including relevant papers from all the retrieved literature.

8.4 Data Extraction

Data extraction is the process of retrieving relevant data and information from selected studies that align with the research objectives. Extracting data from studies requires manual effort and expertise, which carries the risk of human error. To standardize this process, we have established a clear procedure for identifying, recording, and summarizing data from the study. More than three researchers are involved in the data extraction process: each data item is initially extracted by one researcher from the relevant studies and then independently reviewed by two other researchers for accuracy and consistency.

8.5 Data Analysis

The validity threats in data analysis may arise during the process of handling, organizing, analyzing, and interpreting the collected data. In our survey, we included one of our own studies, and while we are confident in the accuracy of our analysis, there is a possibility of misunderstanding technical or methodological reports from other researchers or misinterpreting details of their datasets and evaluation metrics. To mitigate these risks, we actively engaged with the authors of the papers

we could contact, shared our research findings with them, and sought their feedback. We received confirmation and incorporated their suggestions to further refine our paper.

9 CONCLUSION

MNP tackle the critical challenge of automatically predicting suitable and accurate method names, significantly reducing the naming effort for developers and enhancing software development and maintenance. Recent advancements in learning-based MNP techniques have shown considerable promise, demonstrating the significant potential of ML/DL approaches in this area.

In this paper, we offer a comprehensive review of existing learning-based MNP studies, detailing the general workflow, including context extraction, context preprocessing, and context-based prediction. We summarize various strategies employed by existing techniques to optimize these key subprocesses, and we provide an overview of databases, metrics, datasets, and replication packages within the MNP research community.

Despite the progress made, several challenges remain. Accurately understanding and extracting context, preprocessing effectively, and encoding context remain significant hurdles. The construction of high-quality training data, assessing data quality and diversity, and the practical application of MNP techniques in real-world environments also pose ongoing challenges. Addressing these issues is crucial for advancing the field and improving the practical utility of MNP tools.

Future research should focus on overcoming these challenges by developing more robust methods for context extraction and preprocessing, improving data quality and diversity, and creating user-friendly applications. By addressing these areas, the field of learning-based MNP can continue to evolve and make meaningful contributions to SE.

10 ACKNOWLEDGMENTS

We would like to extend our sincere gratitude to the anonymous reviewers for their insightful feedback and constructive comments. We also express our heartfelt thanks to all the authors involved in the papers we reviewed for their valuable feedback and contributions.

This work is partially supported by the National Key Research and Development Program of China (2024YFF0908001), the National Natural Science Foundation of China (U24A20337, 62372228), and the Fundamental Research Funds for the Central Universities (14380029).

REFERENCES

- [1] Ibtissam Abnane, Ali Idri, Imane Chlioui, and Alain Abran. 2023. Evaluating ensemble imputation in software effort estimation. *Empir. Softw. Eng.* 28, 2 (2023), 56.
- [2] Toufique Ahmed and Premkumar Devanbu. 2022. Few-shot Training LLMs for Project-specific Code-Summarization. In *Proceedings of the 37th International Conference on Automated Software Engineering*. ACM, Rochester, MI, USA, 177:1–177:5.
- [3] Toufique Ahmed and Premkumar Devanbu. 2022. Multilingual Training for Software Engineering. In *Proceedings of the 44th International Conference on Software Engineering*. ACM, Pittsburgh, PA, USA, 1443–1455.
- [4] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles A. Sutton. 2015. Suggesting Accurate Method and Class Names. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*. ACM, Bergamo, Italy, 38–49.
- [5] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018. Learning to Represent Programs with Graphs. In *Proceedings of the 6th International Conference on Learning Representations*. OpenReview.net, Vancouver, BC, Canada, 1–17.
- [6] Miltiadis Allamanis, Hao Peng, and Charles Sutton. 2016. A Convolutional Attention Network for Extreme Summarization of Source Code. In *Proceedings of the 33rd International Conference on Machine Learning*. ICML, New York City, NY, USA, 2091–2100.
- [7] Miltiadis Allamanis and Charles Sutton. 2013. Mining source code repositories at massive scale using language modeling. In *Proceedings of the 10th Working Conference on Mining Software Repositories*. MSR, San Francisco, CA, USA, 207–216.

- [8] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2019. code2seq: Generating Sequences from Structured Representations of Code. In *Proceedings of the 7th International Conference on Learning Representations*. OpenReview.net, New Orleans, LA, USA, 1–13.
- [9] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2018. A General Path-based Representation for Predicting Program Properties. In *Proceedings of the 39th Conference on Programming Language Design and Implementation*. ACM, Philadelphia, PA, USA, 404–419.
- [10] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: Learning Distributed Representations of Code. *Proceedings of the ACM on Programming Languages* 3, 40 (2019), 40:1–40:29.
- [11] Francesco Bertolotti and Walter Cazzola. 2023. CombTransformers: Statement-Wise Transformers for Statement-Wise Representations. *IEEE Trans. Software Eng.* 49, 10 (2023), 4677–4690.
- [12] Francesco Bertolotti and Walter Cazzola. 2023. Fold2Vec: Towards a Statement-Based Representation of Code for Code Comprehension. *ACM Transactions on Software Engineering and Methodology* 32, 1 (2023), 6:1–6:31.
- [13] Nghi D. Q. Bui, Yijun Yu, and Lingxiao Jiang. 2021. InferCode: Self-Supervised Learning of Code Representations by Predicting Subtrees. In *Proceedings of 43rd International Conference on Software Engineering*. IEEE, Madrid, Spain, 1186–1197.
- [14] Nghi D. Q. Bui, Yijun Yu, and Lingxiao Jiang. 2021. TreeCaps: Tree-Based Capsule Networks for Source Code Processing. *Proceedings of the AAAI Conference on Artificial Intelligence* 35, 1 (2021), 30–38.
- [15] Jialun Cao, Meiziniu Li, Ming Wen, and Shing chi Cheung. 2023. A Study on Prompt Design, Advantages and Limitations of ChatGPT for Deep Learning Program Repair. *CoRR abs/2304.08191*, 1 (2023), 1–12.
- [16] Hailin Chen, Amrita Saha, Steven Chu-Hong Hoi, and Shafiq Joty. 2023. Personalized Distillation: Empowering Open-Sourced LLMs with Adaptive Learning for Code Generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Singapore, 6737–6749.
- [17] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. *CoRR abs/2107.03374*, 1 (2021), 1–19.
- [18] Jacob Cohen. 1960. A coefficient of agreement for nominal scales. *Educational and psychological measurement* 20, 1 (1960), 37–46.
- [19] Rhys Compton, Eibe Frank, Panos Patros, and Abigail Koay. 2020. Embedding Java Classes with code2vec: Improvements from Variable Obfuscation. In *Proceedings of the 17th International Conference on Mining Software Repositories*. ACM, Seoul, Republic of Korea, 243–253.
- [20] Prem Devanbu, Matthew B. Dwyer, Sebastian G. Elbaum, Michael Lowry, Kevin Moran, Denys Poshyvanyk, Baishakhi Ray, Rishabh Singh, and Xiangyu Zhang. 2020. Deep Learning & Software Engineering: State of Research and Future Directions. *CoRR abs/2009.08525*, 1 (2020), 1–37.
- [21] Junwei Du, Xinshuang Ren, Haojie Li, Feng Jiang, and Xu Yu. 2023. Prediction of bug-fixing time based on distinguishable sequences fusion in open source software. *J. Softw. Evol. Process.* 35, 11 (2023), 1–20.
- [22] Mengnan Du, Fengxiang He, Na Zou, Dacheng Tao, and Xia Hu. 2022. Shortcut Learning of Large Language Models in Natural Language Understanding: A Survey. *CoRR abs/2208.11857*, 1 (2022), 1–10.
- [23] Tore Dybå and Torgeir Dingsøy. 2008. Empirical studies of agile software development: A systematic review. *Information & Software Technology* 50, 9–10 (2008), 833–859.
- [24] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M. Zhang. 2023. Large Language Models for Software Engineering: Survey and Open Problems. *CoRR abs/2310.03533*, 1 (2023), 1–23.
- [25] Chunrong Fang, Weisong Sun, Yuchen Chen, Xiao Chen, Zhao Wei, Qunjun Zhang, Yudu You, Bin Luo, Yang Liu, and Zhenyu Chen. 2024. ESALE: Enhancing Code-Summary Alignment Learning for Source Code Summarization. *IEEE Transactions on Software Engineering* 50, 8 (2024), 2077–2095.
- [26] Dror G. Feitelson, Ayelet Mizrahi, Nofar Noy, Aviad Ben Shabat, Or Eliyahu, and Roy Sheffer. 2022. How Developers Choose Names. *IEEE Transactions on Software Engineering* 48, 2 (2022), 37–52.
- [27] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Proceedings of the 25th Conference on Empirical Methods in Natural Language Processing: Findings*. Association for Computational Linguistics, Online Event, 1536–1547.

- [28] Patrick Fernandes, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Structured neural summarization. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, New Orleans, LA, USA, 1–12.
- [29] Shogo Fujita, Hidetaka Kamigaito, Hiroya Takamura, and Manabu Okumura. 2020. Pointing to Subwords for Generating Function Names in Source Code. In *Proceedings of the 28th International Conference on Computational Linguistics*. COLING, Barcelona, Spain (Online), 316–327.
- [30] Han Gao, Shaoyin Cheng, Yinxing Xue, and Weiming Zhang. 2021. A Lightweight Framework for Function Name Reassignment Based on Large-Scale Stripped Binaries. In *Proceedings of the 30th International Symposium on Software Testing and Analysis*. ACM, Virtual Event, Denmark, 607–619.
- [31] Sa Gao, Chunyang Chen, Zhenchang Xing, Yukun Ma, Wen Song, and Shang-Wei Lin. 2019. A Neural Model for Method Name Generation from Functional Description. In *Proceedings of the 26th International Conference on Software Analysis, Evolution and Reengineering*. IEEE, Hangzhou, China, 411–421.
- [32] Shuzheng Gao, Cuiyun Gao, Chaozheng Wang, Jun Sun, David Lo, and Yue Yu. 2023. Two Sides of the Same Coin: Exploiting the Impact of Identifiers in Neural Code Comprehension. In *Proceedings of the IEEE/ACM 45th International Conference on Software Engineering*. IEEE, Melbourne, Australia, 1933–1945.
- [33] Fan Ge and Li Kuang. 2021. Keywords Guided Method Name Generation. In *Proceedings of the 29th International Conference on Program Comprehension*. IEEE, Madrid, Spain, 196–206.
- [34] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep Code Search. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, Gothenburg, Sweden, 933–944.
- [35] Zhenting Guo, Meng Yan, Hongyan Li, Zhezhe Chen, and Weifeng Sun. 2023. Just-In-Time Method Name Updating With Heuristics and Neural Model. In *Proceedings of the 23rd International Conference on Software Quality, Reliability, and Security*. IEEE, Chiang Mai, Thailand, 707–718.
- [36] Mayy Habayeb, Syed Shariyar Murtaza, Andriy V. Miranskyy, and Ayse Basar Bener. 2018. On the Use of Hidden Markov Model to Predict the Time to Fix Bugs. *IEEE Trans. Software Eng.* 44, 12 (2018), 1224–1244.
- [37] Ruidong Han, Siqi Ma, Juanru Li, Surya Nepal, David Lo, Zhuo Ma, and Jianfeng Ma. 2024. Range Specification Bug Detection in Flight Control System Through Fuzzing. *IEEE Trans. Software Eng.* 50, 3 (2024), 461–473.
- [38] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar T. Devanbu. 2012. On the naturalness of software. In *Proceedings of the 34th International Conference on Software Engineering*. IEEE Computer Society, Zurich, Switzerland, 837–847.
- [39] Einar W. Høst and Bjarte M. Østvold. 2009. Debugging Method Names. In *Proceedings of the 23rd European Conference on Object-Oriented Programming*. Springer, Genoa, Italy, 294–317.
- [40] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John C. Grundy, and Haoyu Wang. 2023. Large Language Models for Software Engineering: A Systematic Literature Review. *CoRR abs/2308.10620*, 1 (2023), 1–62.
- [41] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep Code Comment Generation. In *Proceedings of the 26th International Conference on Program Comprehension*. ACM, Gothenburg, Sweden, 200–210.
- [42] Jianjun Huang, Jianglei Nie, Yuanjun Gong, Wei You, Bin Liang, and Pan Bian. 2024. Raisin: Identifying Rare Sensitive Functions for Bug Detection. In *Proceedings of the 46th International Conference on Software Engineering*. ACM, Lisbon, Portugal, 175:1–175:12.
- [43] Lin Jiang, Hui Liu, and He Jiang. 2019. Machine Learning Based Recommendation of Method Names: How Far are We. In *Proceedings of the 34th International Conference on Automated Software Engineering*. IEEE, San Diego, CA, USA, 602–614.
- [44] Matthew Jin, Syed Shahriar, Michele Tufano, Xin Shi, Shuai Lu, Neel Sundaresan, and Alexey Svyatkovskiy. 2023. InferFix: End-to-End Program Repair with LLMs over Retrieval-Augmented Prompts. *CoRR abs/2303.07263*, 1 (2023), 1–11.
- [45] Barbara Kitchenham. 2004. Procedures for performing systematic reviews. *Keele, UK, Keele University* 33, 2004 (2004), 1–26.
- [46] Pavneet Singh Kochhar, Xin Xia, David Lo, and Shanping Li. 2016. Practitioners’ Expectations on Automated Fault Localization. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, Saarbrücken, Germany, 165–176.
- [47] Alexander LeClair, Siyuan Jiang, and Collin McMillan. 2019. A Neural Model for Generating Natural Language Summaries of Program Subroutines. In *Proceedings of the 41st International Conference on Software Engineering*. IEEE / ACM, Montreal, QC, Canada, 795–806.
- [48] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Arnel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo

Wang, Rudra Murthy V, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Moustafa-Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2023. StarCoder: may the source be with you! *CoRR abs/2305.06161*, 1 (2023), 1–44.

- [49] Yue Li, Zhong Ren, Zhiqi Wang, Lanxin Yang, Liming Dong, Chenxing Zhong, and He Zhang. 2024. Fine-SE: Integrating Semantic Features and Expert Features for Software Effort Estimation. In *Proceedings of the 46th International Conference on Software Engineering*. ACM, Lisbon, Portugal, 27:1–27:12.
- [50] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2021. A Context-based Automated Approach for Method Name Consistency Checking and Suggestion. In *Proceedings of the 43rd International Conference on Software Engineering*. IEEE, Madrid, Spain, 574–586.
- [51] Yanli Li, Chongbin Ye, Huaming Chen, Shiping Chen, Minhui Xue, and Jun Shen. 2023. Towards Better ML-Based Software Services: An Investigation of Source Code Engineering Impact. In *Proceedings of the 2023 International Conference on Software Services Engineering*. IEEE, Chicago, IL, USA, 1–10.
- [52] Fang Liu, Ge Li, Zhiyi Fu, Shuai Lu, Yiyang Hao, and Zhi Jin. 2022. Learning to Recommend Method Names with Global Context. In *Proceedings of the 44th International Conference on Software Engineering*. ACM, Pittsburgh, PA, USA, 1294–1306.
- [53] Kui Liu, Dongsun Kim, Tegawendé F Bisseyandé, Taeyoung Kim, Kisub Kim, Anil Koyuncu, Suntae Kim, and Yves Le Traon. 2019. Learning to spot and refactor inconsistent method names. In *Proceedings of the 41st International Conference on Software Engineering*. ICSE, Montreal, QC, Canada, 1–12.
- [54] Linfeng Liu, Hoan Nguyen, George Karypis, and Srinivasan Sengamedu. 2021. Universal Representation for Code. In *Proceedings of the 25th Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining*. Springer, Virtual Event, 16–28.
- [55] Zhongxin Liu, Xin Xia, Ahmed E. Hassan, David Lo, Zhenchang Xing, and Xinyu Wang. 2018. Neural Machine Translation-based Commit Message Generation: How Far Are We?. In *Proceedings of the 33rd International Conference on Automated Software Engineering*. ACM, Montpellier, France, 373–384.
- [56] Wei Ma, Mengjie Zhao, Ezekiel O. Soremekun, Qiang Hu, Jie M. Zhang, Mike Papadakis, Maxime Cordy, Xiaofei Xie, and Yves Le Traon. 2022. GraphCode2Vec: Generic Code Embedding via Lexical and Program Dependence Analyses. In *Proceedings of the 19th International Conference on Mining Software Repositories*. ACM, Pittsburgh, PA, USA, 524–536.
- [57] Yi-Fan Ma, Yali Du, and Ming Li. 2023. Capturing the Long-Distance Dependency in the Control Flow Graph via Structural-Guided Attention for Bug Localization. In *Proceedings of the 32nd International Joint Conference on Artificial Intelligence*. ijcai.org, Macao, SAR, China, 2242–2250.
- [58] Junayed Mahmud, Nadeeshan De Silva, Safwat Ali Khan, Seyed Hooman Mostafavi, S. M. Hasan Mansur, Oscar Chaparro, Andrian Marcus, and Kevin Moran. 2024. On Using GUI Interaction Data to Improve Text Retrieval-based Bug Localization. In *Proceedings of the 46th International Conference on Software Engineering*. ACM, Lisbon, Portugal, 40:1–40:13.
- [59] Son Nguyen, Hung Phan, Trinh Le, and Tien N. Nguyen. 2020. Suggesting Natural Method Names to Check Name Consistencies. In *Proceedings of the 42nd International Conference on Software Engineering*. ACM, Seoul, South Korea, 1372–1384.
- [60] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. In *Proceedings of the 11th International Conference on Learning Representations*. OpenReview.net, Kigali, Rwanda, 1–13.
- [61] Changan Niu, Chuanyi Li, Bin Luo, and Vincent Ng. 2022. Deep Learning Meets Software Engineering: A Survey on Pre-Trained Models of Source Code. In *Proceedings of the 31st International Joint Conference on Artificial Intelligence*. IJCAI, Vienna, Austria, 5546–5555.
- [62] Safa Omri and Carsten Sinz. 2020. Deep Learning for Software Defect Prediction: A Survey. In *Proceedings of the 42nd International Conference on Software Engineering Workshops*. ACM, Seoul, Republic of Korea, 209–214.
- [63] OpenAI. 2022. ChatGPT. site: <https://openai.com/blog/chatgpt>. Accessed December, 2023.
- [64] OpenAI. 2023. Codex. site: <https://openai.com/blog/openai-codex>. Accessed December, 2023.
- [65] Saeed Parsa, Morteza Zakeri Nasrabadi, Masoud Ekhtiarzadeh, and Mohammad Ramezani. 2023. Method Name Recommendation Based on Source Code Metrics. *Journal of Computer Languages* 74, 1 (2023), 101177.
- [66] Han Peng, Ge Li, Wenhan Wang, Yunfei Zhao, and Zhi Jin. 2021. Integrating Tree Path in Transformer for Code Representation. In *Proceedings of the 35th International Conference on Neural Information Processing Systems*. NeurIPS, virtual, 9343–9354.

- [67] Kai Petersen, Sairam Vakkalanka, and Ludwik Kuzniarz. 2015. Guidelines for Conducting Systematic Mapping Studies in Software Engineering: An Update. *Information and Software Technology* 64, 1 (2015), 1–18.
- [68] Maxim Petukhov, Evelina Gudauskayte, Arman Kaliyev, Mikhail Oskin, Dmitry Ivanov, and Qianxiang Wang. 2022. Method Name Prediction for Automatically Generated Unit Tests. In *Proceedings of the 2nd International Conference on Code Quality*. IEEE, Innopolis, Russian, 29–38.
- [69] Hanwei Qian, Wei Liu, Ziqi Ding, Weisong Sun, and Chunrong Fang. 2023. Abstract Syntax Tree for Method Name Prediction: How Far Are We?. In *Proceedings of the 23rd International Conference on Software Quality, Reliability, and Security*. IEEE, Chiang Mai, Thailand, 464–475.
- [70] Chengwei Qin, Aston Zhang, Zhuosheng Zhang, Jiaao Chen, Michihiro Yasunaga, and Diyi Yang. 2023. Is ChatGPT a General-Purpose Natural Language Processing Task Solver?. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Singapore, 1339–1384.
- [71] Zhiheng Qu, Yi Hu, Jianhui Zeng, Bo Cai, and Shun Yang. 2022. Method Name Generation Based on Code Structure Guidance. In *Proceedings of the 29th IEEE International Conference on Software Analysis, Evolution and Reengineering*. IEEE, Honolulu, HI, USA, 1101–1110.
- [72] Md. Rafiqul Islam Rabin, Nghi D. Q. Bui, Ke Wang, Yijun Yu, Lingxiao Jiang, and Mohammad Amin Alipour. 2021. On the generalizability of Neural Program Models with respect to semantic-preserving program transformations. *Inf. Softw. Technol.* 135 (2021), 106552.
- [73] Md. Rafiqul Islam Rabin, Nghi D. Q. Bui, Ke Wang, Yijun Yu, Lingxiao Jiang, and Mohammad Amin Alipour. 2021. On the generalizability of Neural Program Models with respect to semantic-preserving program transformations. *Inf. Softw. Technol.* 135 (2021), 106552.
- [74] Md. Rafiqul Islam Rabin, Arjun Mukherjee, Omprakash Gnawali, and Mohammad Amin Alipour. 2020. Towards Demystifying Dimensions of Source Code Embeddings. *CoRR* abs/2008.13064, 1 (2020), 1–10.
- [75] Juan Ramos. 2003. Using TF-IDF to determine word relevance in document queries. In *Proceedings of the 1st Instructional Conference on Machine Learning*, Vol. 242. Citeseer, 133–142.
- [76] Veselin Raychev, Pavol Bielik, and Martin T. Vechev. 2016. Probabilistic Model for Code with Decision Trees. In *Proceedings of the 31st International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, Amsterdam, The Netherlands, 731–747.
- [77] Veselin Raychev, Martin T. Vechev, and Eran Yahav. 2014. Code completion with statistical language models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI, Edinburgh, United Kingdom, 419–428.
- [78] Devjeet Roy, Sarah Fakhoury, and Venera Arnaoudova. 2021. Reassessing automatic evaluation metrics for code summarization tasks. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, New York, NY, USA, 1105–1116.
- [79] Felice Salviulo and Giuseppe Scanniello. 2014. Dealing with Identifiers and Comments in Source Code Comprehension and Maintenance: Results from an Ethnographically-informed Study with Students and Professionals. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*. ACM, London, England, United Kingdom, 48:1–48:10.
- [80] Hazem Peter Samoa, Firas Bayram, Pasquale Salza, and Philipp Leitner. 2022. A systematic mapping study of source code representation for deep learning in software engineering. *IET Softw.* 16, 4 (2022), 351–385.
- [81] Chenjie Shen, Jie Zhu, Lei Yu, Li Yang, and Chun Zuo. 2024. Dependency-Aware Method Naming Framework with Generative Adversarial Sampling. In *Proceedings of the 34th International Joint Conference on Neural Networks*. IEEE, Yokohama, Japan, 1–8.
- [82] Egor Spirin, Egor Bogomolov, Vladimir Kovalenko, and Timofey Bryksin. 2021. PSIMiner: A Tool for Mining Rich Abstract Syntax Trees from Code. In *Proceedings of the 18th International Conference on Mining Software Repositories*. IEEE, Madrid, Spain, 13–17.
- [83] Keele Staffs. 2007. Guidelines for Performing Systematic Literature Reviews in Software Engineering. site: http://robertfeldt.net/advice/kitchenham_2007_systematic_reviews_report_updated.pdf. Accessed: December 31, 2023.
- [84] Yulei Sui, Xiao Cheng, Guanqin Zhang, and Haoyu Wang. 2020. Flow2Vec: value-flow-based precise code embedding. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 233:1–233:27.
- [85] Weisong Sun, Chunrong Fang, Yuchen Chen, Guanhong Tao, Tingxu Han, and Quanjun Zhang. 2022. Code Search based on Context-aware Code Translation. In *Proceedings of the 44th International Conference on Software Engineering*. ACM, Pittsburgh, PA, USA, 388–400.
- [86] Weisong Sun, Chunrong Fang, Yuchen Chen, Quanjun Zhang, Guanhong Tao, Yudu You, Tingxu Han, Yifei Ge, Yuling Hu, Bin Luo, and Zhenyu Chen. 2024. An Extractive-and-Abstractive Framework for Source Code Summarization. *ACM Transactions on Software Engineering and Methodology* 33, 3 (2024), 75:1–75:39.
- [87] Weisong Sun, Chunrong Fang, Yifei Ge, Yuling Hu, Yuchen Chen, Quanjun Zhang, Xiuting Ge, Yang Liu, and Zhenyu Chen. 2024. A Survey of Source Code Search: A 3-Dimensional Perspective. *ACM Transactions on Software Engineering*

- and *Methodology* 33, 6 (2024), 166.
- [88] Weisong Sun, Chunrong Fang, Yudu You, Yuchen Chen, Yi Liu, Chong Wang, Jian Zhang, Quanjun Zhang, Hanwei Qian, Wei Zhao, Yang Liu, and Zhenyu Chen. 2023. A Prompt Learning Framework for Source Code Summarization. *CoRR abs/2312.16066*, 1 (2023), 1–23.
 - [89] Weisong Sun, Chunrong Fang, Yudu You, Yun Miao, Yi Liu, Yuekang Li, Gelei Deng, Shenghan Huang, Yuchen Chen, Quanjun Zhang, Hanwei Qian, Yang Liu, and Zhenyu Chen. 2023. Automatic Code Summarization via ChatGPT: How Far Are We? *CoRR abs/2305.12865* (2023), 1–13.
 - [90] Weisong Sun, Yun Miao, Yuekang Li, Hongyu Zhang, Chunrong Fang, Yi Liu, Gelei Deng, Yang Liu, and Zhenyu Chen. 2025. Source Code Summarization in the Era of Large Language Models. In *Proceedings of the 47th IEEE/ACM International Conference on Software Engineering*. IEEE Computer Society, Ottawa, Ontario, Canada, 419–431.
 - [91] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. 2014. Sequence to Sequence Learning with Neural Networks. In *Proceedings of the 27th International Conference on Neural Information Processing Systems*. MIT Press, Montreal, Quebec, Canada, 3104–3112.
 - [92] Takayuki Suzuki, Kazunori Sakamoto, Fuyuki Ishikawa, and Shinichi Honiden. 2014. An approach for evaluating and suggesting method names using n-gram models. In *22nd International Conference on Program Comprehension*. ICPC, Hyderabad, India, 271–274.
 - [93] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurélien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. LLaMA: Open and Efficient Foundation Language Models. *CoRR abs/2302.13971*, 1 (2023), 1–16.
 - [94] Ilya Utkin, Egor Spirin, Egor Bogomolov, and Timofey Bryksin. 2022. Evaluating the Impact of Source Code Parsers on ML4SE Models. *CoRR abs/2206.08713*, 1 (2022), 1–12.
 - [95] Dheeraj Vagavolu, Karthik Chandra Swarna, and Sridhar Chimalakonda. 2021. A Mocktail of Source Code Representations. In *Proceedings of the 36th International Conference on Automated Software Engineering*. IEEE, Melbourne, Australia, 1296–1300.
 - [96] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot-a Java Bytecode Optimization Framework. In *Proceedings of the 9th Conference of The Centre for Advanced Studies on Collaborative Research*. IBM, Mississauga, Ontario, Canada, 214–224.
 - [97] Zane Varner, Çerağ Oğuztüzün, and Feng Long. 2022. Neural Model for Generating Method Names from Combined Contexts. In *Proceedings of the 29th Annual Software Technology Conference*. IEEE, Gaithersburg, MD, USA, 1–6.
 - [98] Antonio Vitale, Antonio Mastropaolo, Rocco Oliveto, Massimiliano Di Penta, and Simone Scalabrino. 2025. Optimizing Datasets for Code Summarization: Is Code-Comment Coherence Enough? *arXiv preprint arXiv:2502.07611* 1, 1 (2025), 1–13.
 - [99] Chong Wang, Jian Zhang, Yebo Feng, Tianlin Li, Weisong Sun, Yang Liu, and Xin Peng. 2024. Teaching Code LLMs to Use Autocompletion Tools in Repository-Level Code Generation. *CoRR abs/2401.06391*, 1 (2024), 1–13.
 - [100] Daniel C. Wang, Andrew W. Appel, Jeffrey L. Korn, and Christopher S. Serra. 1997. The Zephyr Abstract Syntax Description Language. In *Proceedings of the 1st Conference on Domain-Specific Languages*. USENIX, Santa Barbara, California, USA, 213–228.
 - [101] Ke Wang and Zhendong Su. 2020. Blended, Precise Semantic Program Embeddings. In *Proceedings of the 41st International Conference on Programming Language Design and Implementation*. ACM, London, UK, 121–134.
 - [102] Simin Wang, Liguang Huang, Amiao Gao, Jidong Ge, Tengfei Zhang, Haitao Feng, Ishna Satyarth, Ming Li, He Zhang, and Vincent Ng. 2023. Machine/Deep Learning for Software Engineering: A Systematic Literature Review. *IEEE Trans. Software Eng.* 49, 3 (2023), 1188–1231.
 - [103] Shangwen Wang, Ming Wen, Bo Lin, Yepang Liu, Tegawendé F. Bissyandé, and Xiaoguang Mao. 2023. Pre-implementation Method Name Prediction for Object-oriented Programming. *ACM Transactions on Software Engineering and Methodology* 32, 6 (2023), 157:1–157:35.
 - [104] Shangwen Wang, Ming Wen, Bo Lin, and Xiaoguang Mao. 2021. Lightweight Global and Local Contexts Guided Method Name Recommendation with Prior Knowledge. In *Proceedings of the 29th Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, Athens, Greece, 741–753.
 - [105] Yu Wang, Ke Wang, Fengjuan Gao, and Linzhang Wang. 2020. Learning Semantic Program Embeddings with Graph Interval Neural Network. *Proceedings of the ACM on Programming Languages* 4, POPL (2020), 137:1–137:27.
 - [106] Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Proceedings of the 26th Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Virtual Event / Punta Cana, Dominican Republic, 8696–8708.
 - [107] Zejun Wang, Jia Li, Ge Li, and Zhi Jin. 2023. ChatCoder: Chat-based Refine Requirement Improves LLMs’ Code Generation. *CoRR abs/2311.00272*, 1 (2023), 1–11.

- [108] Cody Watson, Nathan Cooper, David Nader-Palacio, Kevin Moran, and Denys Poshyvanyk. 2022. A Systematic Literature Review on the Use of Deep Learning in Software Engineering Research. *ACM Transactions on Software Engineering and Methodology* 31, 2 (2022), 32:1–32:58.
- [109] Da Xiao, Dengji Hang, Lu Ai, Shengping Li, and Hongliang Liang. 2022. Path context augmented statement and network for learning programs. *Empir. Softw. Eng.* 27, 2 (2022), 37.
- [110] Rui Xie, Wei Ye, Jinan Sun, and Shikun Zhang. 2021. Exploiting Method Names to Improve Code Summarization: A Deliberation Multi-Task Learning Approach. In *29th International Conference on Program Comprehension*. ICPC, Madrid, Spain, 138–148.
- [111] Yutao Xie, Jiayi Lin, Hande Dong, Lei Zhang, and Zhonghai Wu. 2024. Survey of Code Search Based on Deep Learning. *ACM Transactions on Software Engineering and Methodology* 33, 2 (2024), 54:1–54:42.
- [112] Frank F. Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. 2022. A Systematic Evaluation of Large Language Models of Code. In *Proceedings of the 6th International Symposium on Machine Programming*. ACM, San Diego, CA, USA, 1–10.
- [113] Sihan Xu, Sen Zhang, Weijing Wang, Xinya Cao, Chenkai Guo, and Jing Xu. 2019. Method Name Suggestion with Hierarchical Attention Networks. In *Proceedings of the 2019 Workshop on Partial Evaluation and Program Manipulation*. ACM, Cascais, Portugal, 10–21.
- [114] Yanming Yang, Xin Xia, David Lo, and John C. Grundy. 2022. A Survey on Deep Learning for Software Engineering. *ACM Comput. Surv.* 54, 10s (2022), 206:1–206:73.
- [115] Yanping Yang, Ling Xu, Meng Yan, Zhou Xu, and Zhongyang Deng. 2022. A Naming Pattern Based Approach for Method Name Recommendation. In *Proceedings of the 33rd International Symposium on Software Reliability Engineering*. IEEE, Charlotte, NC, USA, 344–354.
- [116] Hiroshi Yonai, Yasuhiro Hayase, and Hiroyuki Kitagawa. 2019. Mercem: Method Name Recommendation Based on Call Graph Embedding. In *Proceedings of the 26th Asia-Pacific Software Engineering Conference*. IEEE, Putrajaya, Malaysia, 134–141.
- [117] Yijun Yu. 2019. fAST: Flattening Abstract Syntax Trees for Efficiency. In *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings*. IEEE / ACM, Montreal, QC, Canada, 278–279.
- [118] Oleksandr Zaitsev, Stéphane Ducasse, Alexandre Bergel, and Mathieu Eveillard. 2020. Suggesting Descriptive Method Names: An Exploratory Study of Two Machine Learning Approaches. In *Proceedings of the 13th International Conference on the Quality of Information and Communications Technology*. QUATIC, Faro, Portugal, 93–106.
- [119] Fengyi Zhang, Bihuan Chen, Rongfan Li, and Xin Peng. 2021. A Hybrid Code Representation Learning Approach for Predicting Method Names. *Journal of Systems and Software* 180, 1 (2021), 111011.
- [120] Jingxuan Zhang, Junpeng Luo, Jiahui Liang, Lina Gong, and Zhiqiu Huang. 2023. An Accurate Identifier Renaming Prediction and Suggestion Approach. *ACM Transactions on Software Engineering and Methodology* 32, 6 (2023), 1–51.
- [121] Kechi Zhang, Wenhan Wang, Huangzhao Zhang, Ge Li, and Zhi Jin. 2022. Learning to Represent Programs with Heterogeneous Graphs. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*. ACM, Virtual Event, 378–389.
- [122] Quanjun Zhang, Chunrong Fang, Yuxiang Ma, Weisong Sun, and Zhenyu Chen. 2024. A Survey of Learning-based Automated Program Repair. *ACM Transactions on Software Engineering and Methodology* 33, 2 (2024), 55:1–55:69.
- [123] Quanjun Zhang, Chunrong Fang, Yang Xie, Yaxin Zhang, Yun Yang, Weisong Sun, Shengcheng Yu, and Zhenyu Chen. 2023. A Survey on Large Language Models for Software Engineering. *CoRR abs/2312.15223*, 1 (2023), 1–57.
- [124] Quanjun Zhang, Tongke Zhang, Juan Zhai, Chunrong Fang, Bowen Yu, Weisong Sun, and Zhenyu Chen. 2023. A Critical Review of Large Language Model on Software Engineering: An Example from ChatGPT and Automated Program Repair. *CoRR abs/2310.08879*, 1 (2023), 1–12.
- [125] Xufan Zhang, Yilin Yang, Yang Feng, and Zhenyu Chen. 2019. Software Engineering Practice in the Development of Deep Learning Applications. *CoRR abs/1910.03156*, 1 (2019), 1–11.
- [126] Li Zhong, Chengcheng Xiang, Haochen Huang, Bingyu Shen, Eric Mugnier, and Yuanyuan Zhou. 2024. Effective Bug Detection with Unused Definitions. In *Proceedings of the Nineteenth European Conference on Computer Systems, EuroSys 2024, Athens, Greece, April 22–25, 2024*. ACM, Athens, Greece, 720–735.
- [127] Jie Zhu, Lingwei Li, Li Yang, Xiaoxiao Ma, and Chun Zuo. 2023. Automating Method Naming with Context-Aware Prompt-Tuning. In *Proceedings of the 31st International Conference on Program Comprehension*. IEEE, Melbourne, Australia, 203–214.
- [128] Daniel Zügner, Tobias Kirschstein, Michele Catasta, Jure Leskovec, and Stephan Günnemann. 2021. Language-Agnostic Representation Learning of Source Code from Structure and Context. In *Proceedings of the 9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, 2021*. OpenReview.net, Virtual Event, Austria.