CCCI: Code Completion with Contextual Information for Complex Data Transfer Tasks Using Large Language Models

Hangzhan Jin* Mohammad Hamdaqa†
PolyTechnique Montréal, Montréal, Canada

April 1, 2025

arXiv:2503.23231v1 [cs.SE] 29 Mar 2025

Abstract

Unlike code generation, which involves creating code from scratch, code completion focuses on integrating new lines or blocks of code into an existing codebase. This process requires a deep understanding of the surrounding context, such as variable scope, object models, API calls, and database relations, to produce accurate results. These complex contextual dependencies make code completion a particularly challenging problem. Current models and approaches often fail to effectively incorporate such context, leading to inaccurate completions with low acceptance rates (around 30%). For tasks like data transfer, which rely heavily on specific relationships and data structures, acceptance rates drop even further. This study introduces CCCI, a novel method for generating context-aware code completions specifically designed to address data transfer tasks. By integrating contextual information, such as database table relationships, object models, and library details into Large Language Models (LLMs), CCCI improves the accuracy of code completions. We evaluate CCCI using 289 Java snippets, extracted from over 819 operational scripts in an industrial setting. The results demonstrate that CCCI achieved a 49.1% Build Pass rate and a 41.0% CodeBLEU score, comparable to state-of-the-art methods that often struggle with complex task completion.

Keywords: Code Completion, Large Language Models (LLMs), Data Transfer, Contextual Information

1 Introduction

Recent advancements in Large Language Models (LLMs) like GPT and Copilot [1] have demonstrated the potential in code completion tasks [2, 3], thereby enhancing developer productivity. Despite these advancements, the acceptance rates for these tools remain low [4, 5], primarily due to their generation of generic, context-agnostic code. Prior studies in automated code completion face several significant limitations. Many approaches rely solely on integrating existing libraries [6, 7], recommending APIs [8, 9] based on those libraries without considering data structure or relations. Others require extensive labelled data for fine-tuning [10] or pre-training [11, 12, 13] and demand developers to provide pseudo-code [14, 15], which is labor-intensive and not scalable for large systems. Additionally, some methods focus only on retrieving [16] or searching [17] similar code examples for code suggestions, and some methods pay more attention to predicting code completion invocation [10], neglecting the contextual information like code comments [18, 19] embedded in the applications of enterprises. These methods fail to leverage the real-time data structure and relations present in current software development, resulting in code completions that often lack relevance, especially in data transfer tasks. This gap highlights the need for a more sophisticated approach that can integrate detailed contextual information to enhance the relevance and applicability of code completion.

Given the aforementioned challenges, our study aims to refine the integration of contextual information into LLMs. We hypothesize that this enriched context can improve the utility and accuracy of automatic code completion in data transfer tasks. By addressing these research gaps, our work seeks to pave the way for more sophisticated, context-aware tools that can genuinely enhance developer efficiency. The significance of our research lies in addressing this gap by integrating contextual information—such as database table relations, existing object models, and specific API usage—directly into LLMs. This approach aims to produce higher accuracy code, thereby potentially increasing the practical utility of these models in real-world software development environments. The main contributions of our paper are summarized as follows:

- We propose a retrieval-augmented code completion approach that can retrieve code from the current project and its complex dependencies and integrate different LLMs to improve code completion performance.
- We perform a comprehensive evaluation of our approach, and the results illustrate that our approach improves the GPT-4o by up to 142.6% in terms of

*hangzhan.jin@polymtl.ca
†mhamdaqa@polymtl.ca

CodeBLEU, the Build Pass rate improved dramatically from 0% to 49.1%.

- We examine the performance of our approach in six popular open-source and closed-source LLMs, and the results show our approach can produce a noticeable improvement in those models compared to the GPT-4o model with an original prompt.

The structure of this paper is organized as follows: Section II introduces the challenges of code completion for data transfer tasks necessary for understanding the concepts discussed. Section III details our research methodology. Section IV presents the empirical evaluation of the CCCI method. Section V reviews related work to contextualize our study within the current state of research. Section VI identifies potential threats to the validity of our study, and Section VII concludes the paper with a summary of our contributions and future research directions.

2 Challenges of Code Completion for Data Transfer Tasks

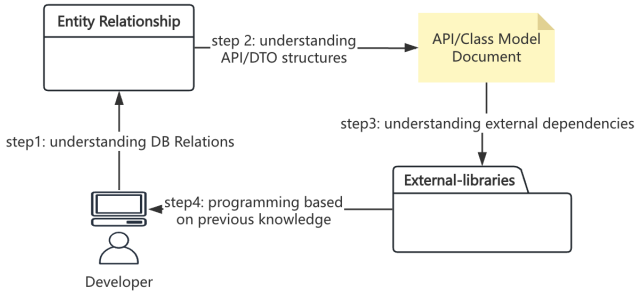

Figure 1: A common data transfer scenario

To illustrate the challenges of code completion for data transfer tasks, we will start by explaining a case study of a Warehouse Management System (WMS) and use this case study to highlight the challenges as well as showcase the steps in our approach. Data transfer tasks are repetitive and complex tasks that developers frequently encounter in real software development, transforming one or more input objects from different data sources like APIs or databases into an output object by mapping fields to the required format [20, 21]. This involves converting front-end request parameters into the format required by third-party services and modifying service outputs to meet the website's needs. As shown in Figure 1, a developer needs to understand the database Relations based on Entity Relationship Graph, API information with data structures in both document and source code, and external libraries to guide programming for the tasks. This process demands extensive time to understand the contextual information [22, 23] for programming. While capable of generating high-quality code, it often requires developers to invest substantial effort in preparing detailed prompts and contextual background, which can be as time-consuming as manual coding, and the quality of generated code depends on developers' skill to write prompts [24, 25, 26], making it difficult for them to adopt code completion to finish data transfer tasks manually.

2.1 Case Study

We explore the application of our CCCI methodology within a real enterprise's WMS. A WMS typically integrates with numerous external systems and is designed to optimize warehouse operations by streamlining tasks such as inventory management, order fulfillment, and logistics. The WMS under study in our research includes 111 tables, 2204 columns, and nine external libraries. In such a system, using LLMs for data transfer tasks needs to analyze the hierarchical data structures and flatten them into text by recursively retrieving data file information within current projects and the nine third-party dependencies, then combine the text with DB relationships and task definitions to construct the intricate prompt to guide LLMs implementing the code completion. The study's goal is to automate this process and use the WMS system to verify our approach.

We use a data transfer task that maps the classes and field information from four input models: InventoryInfoDTO, SKUInfoDTO, UserDTO, and WarehouseDTO (Listing 1) to the output model InventoryResponseDTO (Listing 2) as a showcase to illustrate the challenges of this task due to complex data structures.

Listing 1: An example of input models

```
class InventoryInfoDTO {
    String inventoryName;
    int availableQuantity; }
class SKUInfoDTO {
    int inventoryId;
    String skuName;
    int ownerUserId;
    String ownName; }
class UserDTO {
    String name;
    String contactInfo; }
class WarehouseDTO {
    int inventoryId;
    String warehouseLocation;
    String managerName; }
```

Listing 2: An example of an output model

```
class InventoryResponseDTO {
    String name;
```

```
    int availableQuantity;
    SKUInfo sku;
    WarehouseInfo warehouse; }
```

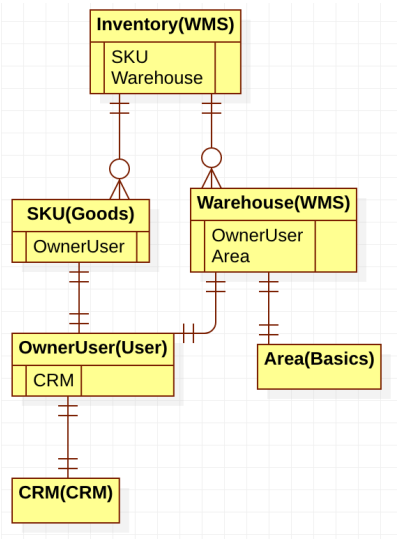

Figure 2: An example of complex data structures

2.2 Challenges

As shown in Figure 2, the Warehouse Management System (WMS) involves a complex relationship among several interconnected objects. The Inventory object includes both SKU (Stock Keeping Unit) and Warehouse objects. The SKU, categorized under the Goods module, and the Warehouse, located within the WMS service, both incorporate an OwnerUser as a sub-object. Additionally, the Warehouse object has an Area object, categorized under the Basics module, as a sub-object. Moreover, the OwnerUser object contains CRM (Customer Relationship Management) details housed in the CRM module. To transfer many input objects from different systems to such an intricate output object, there are four difficulties that need to be overcome:

- **Complicated external dependencies.** Many projects involve external libraries or third-party dependencies distributed in many systems like Customer Relationship Management(CRM), and current tools often fail to accurately account for these, resulting in incorrect or incomplete code [27].
- **Complex data structure.** AI models struggle to accurately generate code [28, 29], especially for projects with complex, hierarchical data structures and interdependent objects(e.g., the Inventory Data Object Model (DTO) includes multi-layers of DTOs), these models are scattered across different projects; leading to errors or inaccurate code.
- **Lack of Database Relations.** Code generation tools may overlook or misinterpret database relationships, which are crucial for tasks like data mapping and object-relational mapping (ORM).
- **Complexity of Prompt Formulation.** Prompt Formulation is a complex and ad-hoc task because prompt formulation requires manually providing project-specific details that the LLM lacks, making each prompt more complex. Crafting detailed prompts demands time, domain knowledge, and iterative refinement to ensure precision. Effective prompts guide the LLM away from common incorrect assumptions, requiring deep expertise in both prompt formulation and model behavior. Prompt designers must adjust and test multiple times to achieve accurate, complete results. Creating prompts that adapt to changing project structures or live data integrations is technically challenging.

The study's goal is to automate this process and use the WMS system to verify our approach.

3 Methodology

As we discussed in the previous section, the complexities of data structure, external dependencies, prompt formulation and the lack of DB relations are obstacles to the LLMs undertaking code completion for data transfer tasks.

In this section, we describe our research methodology called CCCI (Code Completion with Contextual Information), which is designed to enhance code completion by integrating contextual information into Large Language Models (LLMs).

CCCI addresses these issues by automating the inclusion of contextual information, thereby enhancing the relevance and accuracy of the generated code. CCCI takes as input the *project link*, which provides access to the project's source files, dependencies, and database tables, as well as the *Task Definition*, which is a high-level description of what needs to be achieved, as shown in Listing 3.

Listing 3: The task definition

```
Task Overview:
Given the following project <project link>
Generate Java code to transform Input DTOs into Output
  ↪ DTO.

Input/Output Description:
- Input: Multiple DTO objects or lists (e.g., Listing 1).
- Output: Transformed DTO object (e.g., Listing 2).

Additional Context (e.g., "Use BeanUtils.copyProperties
  ↪ for identical fields").
```

Figure 3 illustrates the CCCI approach. CCCI consists of five components that streamline context-aware code completion. It begins with the (i) **File Classifier**, which identifies and categorizes data sources into project files or external libraries. The (ii) **Information Retriever** extracts hierarchical data structures, including class details,

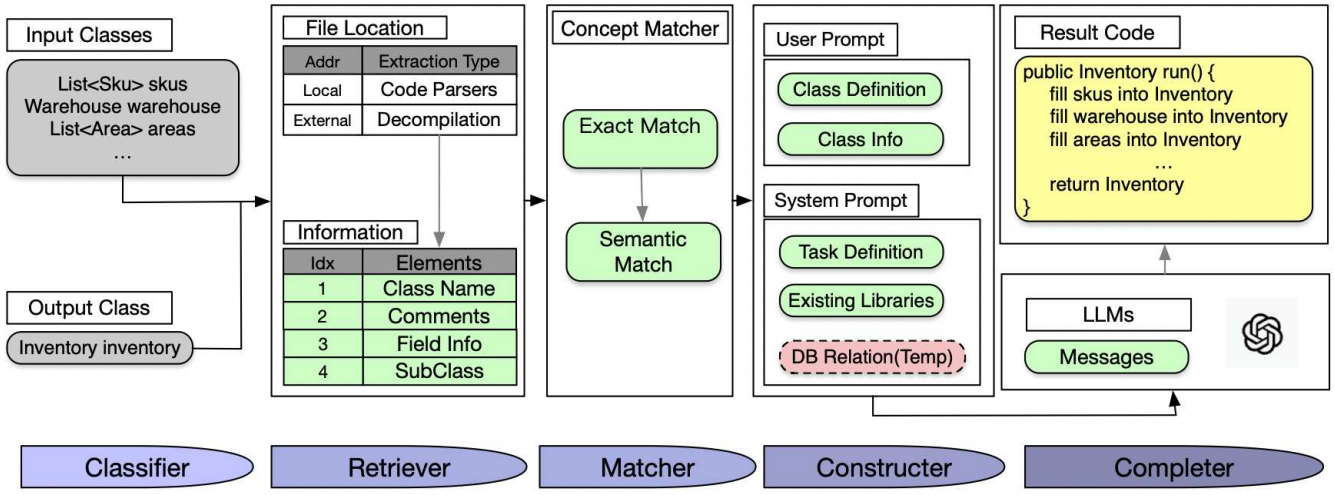

Figure 3: Overview of the CCCI process

field types, and relationships, from the categorized data sources. For project files this can be through parsing, and from external libraries by first decompilation then extraction. Next, the (iii) **Concept Matcher** aligns input and output DTOs fields through exact and semantic matching to establish detailed mappings. These mappings are transformed into structured, text-based prompts by the (iv) **Prompt Constructor**, combining hierarchical relationships and predefined rules to guide the LLM. Finally, the (v) **Code Completer** generates contextually relevant code using the enriched prompt, addressing challenges in complex data transfer tasks. The rest of this section explains each of these components in detail.

3.1 Classifier

The first component in CCCI is Classifier, which is to overcome the challenge of complicated and distributed dependencies across different projects by classifying and locating the DTO source files and then appropriate techniques to be applied to extract hierarchical data structures from both current projects and third-party libraries later. Classification determines whether the inputs and output DTO Source Files belong to the current project or external dependencies. The process begins by recursively searching the DTO files in the current project. If a file is found within the WMS system, it is tagged as a Local File, indicating its direct usage within the project's scope. Conversely, files not found are tagged as External Dependencies. Consider a Warehouse Management System (WMS) project with the following DTOs:

- `Input DTOs`: InventoryInfoDTO, SKUInfoDTO, UserDTO, WarehouseDTO.
- `Output DTO`: InventoryResponseDTO.

The Classifier begins by searching the project directory for each DTO. For example:

- InventoryInfoDTO and SKUInfoDTO are found in the local project directory and are classified as Local.
- UserDTO and WarehouseDTO are found in external libraries user-api.jar and warehouse-api.jar respectively, and are classified as External.

The output of the Classifier is a detailed mapping:

```
InventoryInfoDTO: Local
InventoryResponseDTO: Local
SKUInfoDTO: External (goods-api.jar)
UserDTO: External (user-api.jar)
WarehouseDTO: External (warehouse-api.jar)
```

This classification ensures subsequent Retriever can extract the class information from the appropriate sources.

3.2 Retriever

The goal of the second phase is to retrieve complicated data structures from the current project and its dependencies; we employ the tagged files preprocessed by Classifier to Retrieve data structures from the root node to children nodes and their children nodes recursively. These nodes include class names, field types, field names, and comments in each layer, which enrich the contextual information of the input and output DTOs (as demonstrated in Listing 4) utilized for the next field matching. We use different strategies to handle local and external files. For local files, we employ JavaParser as a code parser to extract detailed information from these files, facilitating a thorough understanding of the project's internal structure. However, files distributed within third-party libraries pose a unique challenge as they are complicated,

compiled and lack accessible source text. To address these problems, we employ decompilation techniques to analyze these compiled files. In this context, the Java Reflection Mechanism is implemented to decompile and extract hierarchical data structures from these files, such as annotations, fields, and their types. This dual approach ensures a comprehensive data structure extraction from both in-project (the WMS system) source files and external compiled libraries (third-party dependencies).

Code parsers are pivotal in our methodology for analyzing source code from current projects. They textually parse code to extract structural and semantic details such as class names, comments, field names, and field types. This process provides a rich context that enhances our model's ability to generate accurate and contextually relevant prompts. Serving as a fundamental component for data preparation, code parsers accommodate a variety of programming languages and environments.

Decompilation Techniques are employed to handle compiled DTO files, especially those from external libraries that typically lack source comments. These techniques enable us to decompile the compiled files to access their underlying structure, extracting metadata such as class names, field names, field types, and annotations. This approach helps recover structural information provided by annotations, compensating for the absence of direct commentary and ensuring a thorough extraction of class information critical for our processing needs.

Example: for the local DTOs identified in the Classifier, we have local DTO - InventoryInfoDTO:

```
//The inventory information
class InventoryInfoDTO {
    // Name of the inventory
    String inventoryName;
    // Stock available
    int availableQuantity;
}
```

The Retriever extracts:

```
Class: InventoryInfoDTO
Fields:
  - inventoryName: String
  - availableQuantity: int
Comments:
  - inventoryName: "Name of the inventory"
  - availableQuantity: "Stock available"
```

External DTO - UserDTO (from user-api.jar): after decompiling by Java Reflection, the Retriever extracts:

```
Class: UserDTO
Fields:
  - name: String
  - contactInfo: String
```

This information is stored in a structured format like Listing 4, ready for the next component.

3.3 Matcher

The Matcher facilitates accurate data transfer between input and output Data Transfer DTOs; our approach uses a two-step field-matching process: **Step 1: Exact Field Matching** This initial step automatically transfers fields with identical names between the input and output DTOs, ensuring straightforward and accurate data alignment when field names match exactly. **Step 2: Semantic Field Matching** In cases, as defined in Listing 3 for the second rule, where field names differ but represent the same concept, a semantic matching approach leverages large language models (LLMs) to predict and align fields based on contextual clues, such as class information or field annotations. This step enhances flexibility by enabling matches even when fields have different names but similar meanings, reducing the need for manual mapping adjustments. The Semantic Field Matching is designed to identify initial correspondences between source and target Data Transfer Objects, providing a foundation for creating compatible structures. This matching process involves four key steps: **1. Selecting representative concepts by filtering redundant fields:** This step focuses on selecting only the relevant class information directly related to the input and output parameters while eliminating redundancy. For example, if multiple DTOs reference the same class(e.g., OwnerUser in Figure 2), the class is selected only once. Similarly, if a superclass is already included, its subclasses are excluded unless they add distinct semantic meaning. This ensures the representation is both concise and meaningful. **2.Generating definitions that include annotations, field names, and comments:** Class information such as fields, comments, and hierarchical context is retrieved after identifying the representative concepts. This information, provided by the previous component in the pipeline, forms the basis for understanding the semantics of each concept. **3.Computing embeddings and cosine similarity scores for semantic alignment:** The selected concepts and their associated definitions are combined into structured representations. These representations are then converted into dense vector embeddings, capturing their semantic meanings. A cosine similarity [30] measure is applied to compare the vectors, calculating how closely aligned the source and target representations are. **4.Ranking and selecting the most rel-**

evant matches: The correspondences with the highest similarity scores are identified and ranked. These top-ranked correspondences are then selected and incorporated into the context for further refinement. Focusing on the most relevant matches, this step ensures that subsequent processing can combine these correspondences into instructions, which guides LLMs to match the semantic information to generate code from different DTOs.

Example: For the input DTOs InventoryInfoDTO and SKUInfoDTO and the output DTO InventoryResponseDTO: Exact matching matches the fields with the exact same name:

```
InventoryInfoDTO.warehouseName →
InventoryResponseDTO.warehouseName
InventoryInfoDTO.availableQuantity →
InventoryResponseDTO.availableQuantity
```

Semantic matching matches the fields with similar meanings even if their names are different:

```
InventoryInfoDTO.inventoryName →
InventoryResponseDTO.name
SKUInfoDTO.skuName →
InventoryResponseDTO.sku.skuName
SKUInfoDTO.user.name →
InventoryResponseDTO.sku.ownName
```

The resulting mapping table merges all the fields matched, including both the exact matching and semantic matching:

```
Input Field                    →Output Field
InventoryInfoDTO.warehouseName →
  ↪ InventoryResponseDTO.warehouseName
InventoryInfoDTO.inventoryName →
  ↪ InventoryResponseDTO.name
InventoryInfoDTO.availableQuantity →
  ↪ InventoryResponseDTO.availableQuantity
SKUInfoDTO.skuName             →
  ↪ InventoryResponseDTO.sku.skuName
SKUInfoDTO.user.name           →
  ↪ InventoryResponseDTO.sku.ownName
```

3.4 Constructor

The Constructor formulates a structured prompt that combines the task definition and rules into a system prompt, as well as mappings and contextual information generated by Matcher into the user prompt. This prompt is used as detailed instruction to guide LLMs in generating contextually relevant code. The Constructor organizes the mappings into a hierarchical data model, which is then converted into a text-based prompt. This prompt includes task-specific instructions. For example, to alleviate the issue of token limitation, we use BeanUtils.copyProperties for exact matches. It also incorporates rules to avoid redundant operations or unnecessary class declarations. Example: Using the mapping table from the Matcher, the Constructor generates the following prompt, the input DTOs are the input for the matching task; the output DTO is the output that the matching task is supposed to return; the rules are to guide LLMs completing the both exact matching and semantic matching.

```
Task: Map input DTOs to the output DTO.
Input DTOs:
  - InventoryInfoDTO: [warehouseName, inventoryName,
    ↪ availableQuantity]
  - SKUInfoDTO: [skuName]
  - UserDTO: [name]
Output DTO:
  - InventoryResponseDTO:
      - warehouseName →InventoryInfoDTO.warehouseName
      - name →InventoryInfoDTO.inventoryName
      - availableQuantity →
        ↪ InventoryInfoDTO.availableQuantity
      - sku.skuName →SKUInfoDTO.skuName
      - sku.ownName →SKUInfoDTO.user.name
Rules:
  1. Use BeanUtils.copyProperties for fields with
    ↪ identical names.
  2. Manually map fields with different names but similar
    ↪ semantics.
```

3.5 Completer

The Completer employs the structured prompt formulated by Constructor to produce the final code. This step leverages the prompt with detailed mappings and rules to guide LLMs in generating syntactically and contextually correct code. The prompt is fed into LLMs, which interpret the instructions and generate code that adheres to the rules and mappings. Example: Given the prompt, the LLM generates:

```
InventoryResponseDTO response = new
  ↪ InventoryResponseDTO();
response.setName(inventoryInfoDTO.inventoryName);
// Copy identical fields
BeanUtils.copyProperties(inventoryInfoDTO, response);

// Map nested objects
SKUInfo skuInfo = new SKUInfo();
skuInfo.setSkuName(skuInfoDTO.getSkuName());
skuInfo.setOwnName(user.getName());
response.setSku(skuInfo);
```

This final code ensures that all mappings are implemented correctly, including handling of nested objects.

Listing 4: The example of class information retrieved by CCCI

```
Input Parameters:
// Query inventory flow table
- private com.xx.ResponseList<com.xx.
InventorySkuItemFlow> inventorySkuItemFlow;
// Query warehouse areas
- private java.util.List<com.xx.WarehouseArea>
  warehouseAreaList;
// Query warehouse locations
- private java.util.List<com.xx.
WarehouseLocation> warehouseLocationList;
// Query warehouse
```

```
- private com.xx.Warehouse
  warehouse;
Output Information:
- private List<com.xx.api_2694416191343104
.resp.Content> contents;

Entity Details:
- Entity:Inventory Info DTO:com.xx.Inventory
  Fields:
  id:primary key:long,
  name:name of inventory:String,
  sku: sku info object: com.xx.SkuDTO
- Entity:SKU for goods:com.xx.SkuDTO
  Fields:
  inventoryName:name of inventory:String,
  ownName:owner name:String
- Entity: com.xx.UserInfoDTO
  Fields:
  name:username:String
- Entity:Area Info:com.xx.WarehouseArea
  Fields: ...
- Entity: com.xx.WarehouseLocation
- Entity: com.xx.Warehouse
- Entity: com.xx..resp.Content
- Entity: com.xx.SkuInfoVO
- Entity: com.xx.WarehouseAreaVO
```

Listing 4 illustrates the retrieved class information, including data structure of inputs and output DTO(s), the comments and package of DTO(Entity), fields of DTO, comments and type of fields.

Listing 5: An example of database table relations

```
Warehouse Domain:
- warehouse (Warehouse)
  |-> warehouse_area (Warehouse Area):
  1:N relationship
  |-> warehouse_dock (Dock):
  1:N relationship
- warehouse_area (Warehouse Area)
  |-> warehouse_location (Warehouse
  Location): 1:N relationship
```

4 Empirical Evaluation

In the empirical evaluation section, we adapt the benchmarking study [31] and utilize a dataset consisting of 819 scripts from the WMS systems, which have been developed and deployed. These scripts, with their respective input and output objects, serve as the reference for the generated code and standard structure for our CCCI method. Using this approach, we generate new scripts and then employ CodeBLEU, BLEU-4, Edit Similarity, and Build Pass to score the generated scripts. We use the same 289 scripts from the whole system to generate code. The results will illustrate the accuracy of the generated scripts compared to the original ones and the impact of different models.

Table 1: The dataset for WMS system

System	Snippets	Tables	Dependencies
WMS	819	111	9

As the table shows, the WMS system we use for the experiment includes 819 reference code snippets and 111 tables, and it depends on nine third-party libraries (dependencies); such an intricate system emphasizes the challenge of retrieving contextual information from both the current project and its third-party dependencies.

Listing 6: An example of snippets

```
xxx.RequestModel param = new xxx.RequestModel();

Integer skuForm = Optional.ofNullable(asnOrder)
        .map(AsnOrder::getAsnOrderItems)
        .filter(CollectionUtils::isNotEmpty)
        .map(it -> it.get(0))
        .map(AsnOrderItem::getSkuForm)
        .orElse(SkuFormEnum.GOOD.getCode());

ShelfOrder shelfOrder = new ShelfOrder();
shelfOrder.setRelatedOrderId(result.getId());
shelfOrder.setRelatedOrderType(RelatedOrderTypeEnum
.INBOUND_ORDER.getCode());
shelfOrder.setShelfItems(requestModel.getSkus()
        .stream()
        .filter(Sku::getIsCheck)
        .map(it -> {
            ShelfItem shelfItem = new ShelfItem();
            shelfItem.setQuantity(it.getQuantity());
            shelfItem.setSku(it.getSku());
            shelfItem.setSkuForm(skuForm);
            return shelfItem;
        })
        .collect(Collectors.toList()));
param.getShelfOrderList().add(shelfOrder);
param.setWarehouseId(asnOrder.getWarehouseId());
param.setType(ShelfOrderTypeEnum.SHELF.getCode());
return param;
```

4.1 Large Language Models

To avoid our approach overfitting a certain Large Language Model, we utilize six common LLMs (as shown in Table 2, three open-source and three closed-source models respectively) for code completion. These models operate with default settings designed to enhance result reproducibility: (a) a maximum output token limit of 4096, ensuring extensive output capacity, (b) a temperature setting of zero, which promotes deterministic output by eliminating randomness in response selection, and (c) a `top_p` setting of 0.2 configured to focus the model's predictions on the most likely outcomes, thereby improving the precision of the generated code.

Table 2: LLMs used to experiment

Model	Source	Provider
GPT-4o	Closed	OpenAI
Gemini-pro-1.5	Closed	Google
Claude-3.5-haiku	Closed	Anthropic
Llama-3.1-405b	Open	Meta
Qwen-2.5-coder-32b-instruct	Open	Alibaba
Deepseek-3	Open	Deepseek

4.2 Evaluation Metrics

4.2.1 CodeBLEU score

CodeBLEU [32] score is an extension of the BLEU score, designed specifically for code completion tasks. In addition to measuring textual similarity, it incorporates Abstract Syntax Tree (AST) and data-flow structures to evaluate both the grammatical correctness and the logical coherence of the generated code, providing a more comprehensive assessment than BLEU alone.

4.2.2 BLEU score

BLEU [33] score is a metric originally developed for evaluating machine translation but has been widely adopted for assessing code completion. In this context, it measures the similarity of n-grams between the generated code and the ground truth. For our evaluation, we use BLEU-4, which compares sequences of four tokens, following the methodology used in previous research.

4.2.3 Edit Similarity

Edit Similarity(ES) measures the similarity between two code snippets based on the editing operations.

4.2.4 Build Pass

Build Pass evaluates the correctness of code generated by LLMs due to the limitations of metrics that only compare the reference code and target code snippet in terms of similarity. We employ the compilation mechanism to check if the generated code snippets are runnable at the first stage. If a code script can be compiled, we further use the testing case to invoke the function in each script at the second stage to simply check the output of these functions. A code script will be regarded as successful in Build Pass if the two stages are passed.

4.3 RQ1 How effective is the CCCI method when using enhanced prompts tailored with contextual information?

We hypothesized that the CCCI method, by utilizing enhanced prompts enriched with contextual information, would significantly improve the effectiveness of code completion as measured by BLEU-4, CodeBLEU, Edit Similarity and Build Pass scores. This enhancement is expected to lead to a better alignment with the real software development tasks, thus producing more accurate and functional code than the original prompt without class information retrieved from the current project and its dependencies. To test this hypothesis, we selected 289 production code scripts and used the CCCI method to regenerate corresponding scripts. The effectiveness of the regenerated scripts was quantified by calculating the average BLEU-4, CodeBLEU, Edit Similarity and Build Pass scores across these samples.

Table 3: Overall BLEU-4(B4), CodeBLEU(CB), Build Pass(BP) and Edit Similarity(ES) for CCCI and the original prompt

Method	B4(%)	CB(%)	ES(%)	BP(%)
CCCI	**20.3**	**41.0**	**36.7**	**49.1**
original	10.6	16.9	5.5	0.0

4.3.1 BLEU-4 Score Result

The average BLEU-4 score for CCCI-generated scripts is 20.3, which is substantially higher than the original prompt's average of 10.6. This 91.5% increase in score highlights CCCI's enhanced capability in accurate code completion through n-gram comparison.

4.3.2 CodeBLEU Result

Similarly, the average CodeBLEU score for CCCI is 41.0, compared to 16.9 for the original prompt, reflecting an improvement of 142.6%. This improvement indicates more grammatical correctness and logic correctness within the generated scripts, demonstrating CCCI's superior contextual integration.

4.3.3 Edit Similarity Result

The average Edit Similarity has improved significantly from 5.5 to 36.7. The noticeable improvement indicates that developers can use the code scripts with less effort for code modification based on the generated code scripts to satisfy their requirements.

4.3.4 Build Pass Result

As Table 3 shows, the code scripts generated through the original prompt cannot be either compiled or tested, which demonstrates the challenges for code completion without the contextual information in real industrial settings.

The higher average scores for a series of metrics above validate CCCI's effectiveness in integrating contextual information retrieved from the current project into code

completion. The variability in CodeBLEU scores is predominantly attributed to a main factor observed in the original scripts:

Custom Code Exception Handling: Many scripts utilized custom exceptions for error handling (e.g., Listing 7), which were not adequately captured by CCCI. This discrepancy arose because the contextual information provided did not include sufficient details on the custom exception-handling classes and methods. Enhancing the contextual richness to include these specifics could potentially improve performance in scenarios involving custom error handling.

Listing 7: An example of a script with custom code exception handling

```
if (CollectionUtils.isEmpty(inboundOrderList)) {
    throw INVENTORY_NOT_FOUND.instanceException();
}
InboundOrder inboundOrder = inboundOrderList.get(0);
if (!InboundOrderStatusEnum.DEFECTIVE_WAIT_CONFIRMED
.getCode().equals(inboundOrder.getStatus())) {
    throw INVENTORY_STATUS_ERROR.instanceException();
}
```

Table 4: BLEU-4(%) for CCCI with Different LLMs

Model	Original	CCCI	↑ (%)
GPT-4o	10.6	20.3	91.5
Gemini-pro-1.5	6.2	22.1	256.5
Claude-3.5-haiku	7.4	21.1	185.1
Llama-3.1-405b	14.2	25.0	76.1
Qwen-2.5-coder-32b	14.0	22.7	62.1
Deepseek-3	11.4	23.6	107.0

Table 5: CodeBLEU(%) for CCCI with different LLMs

Model	Original	CCCI	↑ (%)
GPT-4o	16.9	41.0	142.6
Gemini-pro-1.5	28.8	38.0	31.9
Claude-3.5-haiku	19.1	34.5	80.6
Llama-3.1-405b	18.9	41.0	116.9
Qwen-2.5-coder-32b	21.3	40.4	89.7
Deepseek-3	24.2	41.7	72.3

Table 6: Edit Similarity(%) for CCCI with different LLMs

Model	Original	CCCI	↑ (%)
GPT-4o	5.5	36.7	567.3
Gemini-pro-1.5	4.5	36.9	720.0
Claude-3.5-haiku	4.6	24.0	421.7
Llama-3.1-405b	5.2	42.7	721.2
Qwen-2.5-coder-32b	5.4	38.4	611.1
Deepseek-3	5.5	38.2	594.5

Table 7: Build Pass Rate(%) for CCCI with different LLMs, there is no single script produced by the original prompt passing the Build Pass test

Model	Total	Build Pass	Pass Rate (%)
GPT-4o	289	142	49.1
Gemini-pro-1.5	289	185	64.0
Claude-3.5-haiku	289	75	26.0
Llama-3.1-405b	289	150	51.9
Qwen-2.5-coder-32b	289	99	34.3
Deepseek-3	289	83	28.7

4.4 4.4 RQ2 What is the impact of the CCCI method across different large-scale language models?

The motivation behind RQ2 is to demonstrate the adaptability of the CCCI method across different large-scale language models (LLMs [34]), as validated on six popular open-source or closed-source models: ChatGPT4 [35, 36], Gemini-pro [37], Claude-3 [38], Deepseek-3 [39], Llama-3.1 [40], and Qwen-2.5 [41]. By evaluating the CodeBLEU, BLEU-4, Edit Similarity, and Build Pass scores, we aim to prove that CCCI can consistently generate accurate and relevant code across a variety of LLMs, highlighting its robustness in different model environments. As shown in the tables above, most models achieve similar scores, with CodeBLEU, BLEU-4, Edit Similarity, and Build Pass results relatively close across models such as GPT-4o, Gemini-pro, Llama-3.1, Deepseek-3, Qwen-2.5. However, Claude-3.5-haiku significantly underperforms, with lower scores in these metrics. This discrepancy is attributed to Claude-3.5-haiku's weaker instruction-following capabilities, while the other models demonstrate comparable performance, reinforcing the CCCI method's effectiveness across various LLMs.

5 Related Work

The field of code completion using large language models (LLMs) has seen significant advancements in recent years. This section reviews relevant literature that contributes to understanding the current state of research in this domain. Various approaches have been explored for code completion. Sequence-based methods generate code token by token based on input descriptions, while tree-based methods construct parse trees from natural language descriptions and convert them into code. Recent models like CodeT5 and CodeGPT leverage the transformer architecture to enhance the quality of generated code [42, 43]. Pre-trained language models such as CodeBERT, CodeT5, InCoder, and CodeGPT have been pivotal in advancing code generation tasks. These models are typically fine-tuned for specific tasks, such as generating code from natural language descriptions, completing code snippets,

and generating unit tests. For instance, CodeBERT and CodeT5 have shown substantial improvements in generating accurate code by leveraging large-scale pre-training on code and natural language data. Retrieval-augmented language models [44, 45, 16, 46] on for their ability to improve the performance of code completion tasks by incorporating relevant external information. For example, the REDCODER framework retrieves relevant code snippets or summaries to enhance the performance of code completion models. This approach has been shown to improve the accuracy and relevance of generated code by providing additional context to the model.

Recent research has also focused on domain-specific code completion, such as generating unit tests and library-oriented code. For instance, the study on using large language models for automated unit test generation demonstrates the potential of these models to significantly reduce the effort required for test creation by generating high-quality unit tests automatically. Similarly, the CodeGen4Libs approach presents a two-stage method for generating code that interacts with third-party libraries, addressing the challenges of library-specific code completion.

6 Threats to Validity

Quality of the Dataset. The dataset used in this study comprises code that is currently deployed in production within a real industrial sector; we extract these code scripts from the WMS project. While this ensures that our findings are grounded in practical, real-world applications, it also poses a limitation: the results obtained might not be generalizable to other datasets or environments. The specific characteristics and challenges of the industrial dataset may influence the performance metrics, and thus, transferring our approach to a different context might yield different outcomes.

Parameter Specification in Models. In our experiments, the top_p and temperature parameters were set to 0.2 and zero based on preliminary tests designed to stabilize results and mitigate the risk of variability. However, this parameter setting may not be optimal for other datasets or different types of tasks. While the selection of 289 scripts was intended to provide a robust average by minimizing the effects of outliers, different configurations or datasets might require adjustments to this parameter to achieve the best results. In addition, we use all the scripts with lengths between 300 and 700 because of the limitation of the LLMs(the shorter length will be just lines of code, which makes the metrics unreliably high; the longer length will cause the LLMs generating code with fewer details), and the invocation speed also poses a challenge to experiment with all the scripts. For example, we invoke LLMs 12 times for every script; compiling and testing the script is also time-consuming. In general, the whole experiment takes more than 8 hours for these 289 scripts.

7 Conclusion

This research introduced the CCCI method, a novel approach aimed at enhancing the quality of automated code completion. By retrieving 289 scripts from over 819 operational scripts currently deployed across internet enterprise applications, we demonstrated a notable improvement over many large language models in CodeBLEU, BLEU-4, Edit Similarity, and Build Pass scores. These results emphasize the practical applicability and effectiveness of our approach in real-world settings.

Furthermore, our evaluation across multiple large-scale language models demonstrated the adaptability of the CCCI method, as it consistently generated accurate code with similar performance across most models, further validating its robustness and versatility in different LLM environments.

We do not compare our approach to the prior studies such as RLPG [47] because they focus on different dimensions of code completion. For example, RLPG completes line-level code in the current repository, and our method completes body-level code scripts that are more complex than line-level code completion. Comparing with popular retrieval-based approaches is not suitable because our dataset is not generic enough to compare. In addition, we conduct the evaluation on various metrics, and we employ different LLMs to evaluate our approach, which is commonly absent in prior approaches.

Despite being tailored primarily for data mapping tasks, the principles underlying the CCCI method apply to broader scenarios, such as Service Mashup. This versatility presents a potential path for future research, exploring the adaptation and application of CCCI in various other contexts where automated, context-aware code completion can play a pivotal role.

References

[1] *GitHub Copilot, your AI pair programmer*. `https://github.com/features/copilot`. Last accessed May 2023. 2021.

[2] Fang Liu et al. "A self-attentional neural architecture for code completion with multi-task learning". In: *Proceedings of the 28th International Conference on Program Comprehension*. 2020, pp. 37–47.

[3] Samuel Abedu, Ahmad Abdellatif, and Emad Shihab. "LLM-Based Chatbots for Mining Software Repositories: Challenges and Opportunities". en. In: *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*. Salerno Italy: ACM, June 2024, pp. 201–210. ISBN: 9798400717017. DOI: `10.1145/3661167.`

3661218. URL: https://dl.acm.org/doi/10.1145/3661167.3661218 (visited on 10/14/2024).

[4] Hussein Mozannar et al. *When to Show a Suggestion? Integrating Human Feedback in AI-Assisted Programming*. en. arXiv:2306.04930 [cs]. Apr. 2024. URL: http://arxiv.org/abs/2306.04930 (visited on 10/14/2024).

[5] Jenny T. Liang, Chenyang Yang, and Brad A. Myers. "A Large-Scale Survey on the Usability of AI Programming Assistants: Successes and Challenges". en. In: *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. Lisbon Portugal: ACM, Feb. 2024, pp. 1–13. ISBN: 9798400702174. DOI: 10.1145/3597503.3608128. URL: https://dl.acm.org/doi/10.1145/3597503.3608128 (visited on 10/14/2024).

[6] Ze Tang et al. *Domain Adaptive Code Completion via Language Models and Decoupled Domain Databases*. en. arXiv:2308.09313 [cs]. Sept. 2023. URL: http://arxiv.org/abs/2308.09313 (visited on 10/14/2024).

[7] Mingwei Liu et al. "CodeGen4Libs: A Two-Stage Approach for Library-Oriented Code Generation". en. In: *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Luxembourg, Luxembourg: IEEE, Sept. 2023, pp. 434–445. ISBN: 9798350329964. DOI: 10.1109/ASE56229.2023.00159. URL: https://ieeexplore.ieee.org/document/10298327/ (visited on 10/14/2024).

[8] Zexiong Ma et al. "Compositional API Recommendation for Library-Oriented Code Generation". In: *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension*. ICPC '24. New York, NY, USA: Association for Computing Machinery, 2024, pp. 87–98. ISBN: 9798400705861. DOI: 10.1145/3643916.3644403. URL: https://dl.acm.org/doi/10.1145/3643916.3644403 (visited on 10/17/2024).

[9] Nihal Jain et al. *On Mitigating Code LLM Hallucinations with API Documentation*. en. arXiv:2407.09726 [cs]. July 2024. URL: http://arxiv.org/abs/2407.09726 (visited on 10/14/2024).

[10] Tim Van Dam et al. "Investigating the Performance of Language Models for Completing Code in Functional Programming Languages: a Haskell Case Study". en. In: *Proceedings of the 2024 IEEE/ACM First International Conference on AI Foundation Models and Software Engineering*. Lisbon Portugal: ACM, Apr. 2024, pp. 91–102. ISBN: 9798400706097. DOI: 10.1145/3650105.3652289. URL: https://dl.acm.org/doi/10.1145/3650105.3652289 (visited on 10/14/2024).

[11] Xue Jiang et al. "TreeBERT: A tree-based pre-trained model for programming language". In: *Uncertainty in Artificial Intelligence*. PMLR, 2021, pp. 54–63.

[12] Yanlin Wang and Hui Li. "Code completion by modeling flattened abstract syntax trees as graphs". In: *Proceedings of the AAAI conference on artificial intelligence*. Vol. 35. 2021, pp. 14015–14023.

[13] Fang Liu et al. "Multi-task learning based pre-trained language model for code completion". In: *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 2020, pp. 473–485.

[14] Hao Yu et al. "CoderEval: A Benchmark of Pragmatic Code Generation with Generative Pre-trained Models". en. In: *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. arXiv:2302.00288 [cs]. Feb. 2024, pp. 1–13. DOI: 10.1145/3597503.3623322. URL: http://arxiv.org/abs/2302.00288 (visited on 10/14/2024).

[15] Victoria Jackson et al. *Creativity, Generative AI, and Software Development: A Research Agenda*. en. arXiv:2406.01966 [cs]. June 2024. URL: http://arxiv.org/abs/2406.01966 (visited on 10/14/2024).

[16] Shuai Lu et al. "ReACC: A Retrieval-Augmented Code Completion Framework". In: *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 2022, pp. 6227–6240.

[17] Gabriel Ryan et al. *Code-Aware Prompting: A study of Coverage Guided Test Generation in Regression Setting using LLM*. en. arXiv:2402.00097 [cs]. Apr. 2024. URL: http://arxiv.org/abs/2402.00097 (visited on 10/14/2024).

[18] David OBrien et al. "Are Prompt Engineering and TODO Comments Friends or Foes? An Evaluation on GitHub Copilot". en. In: *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. Lisbon Portugal: ACM, Apr. 2024, pp. 1–13. ISBN: 9798400702174. DOI: 10.1145/3597503.3639176. URL: https://dl.acm.org/doi/10.1145/3597503.3639176 (visited on 10/14/2024).

[19] Toufique Ahmed et al. *Automatic Semantic Augmentation of Language Model Prompts (for Code Summarization)*. en. arXiv:2304.06815 [cs]. Jan. 2024. URL: http://arxiv.org/abs/2304.06815 (visited on 10/14/2024).

[20] Anne H.H. Ngu et al. "Semantic-Based Mashup of Composite Applications". In: *IEEE Transactions on Services Computing* X.X (2010). Last accessed October 2023. DOI: 10.1109/TSC.2010.8. URL: https://www.researchgate.net/publication/220595094_Semantic-Based_Mashup_of_Composite_Applications.

[21] Giusy Di Lorenzo et al. "Data Integration in Mashups". In: *ACM SIGMOD Record* 38.1 (2009). Last accessed October 2023, pp. 59–66. DOI: 10.1145/1558334.1558343. URL: https://www.researchgate.net/publication/220415359_Data_Integration_in_Mashups.

[22] Egor Bogomolov et al. *Long Code Arena: a Set of Benchmarks for Long-Context Code Models*. en. arXiv:2406.11612 [cs]. June 2024. URL: http://arxiv.org/abs/2406.11612 (visited on 10/14/2024).

[23] Zhe Liu et al. "Fill in the blank: Context-aware automated text input generation for mobile gui testing". In: *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 2023, pp. 1355–1367.

[24] Marcus J. Min et al. *Beyond Accuracy: Evaluating Self-Consistency of Code Large Language Models with IdentityChain*. en. arXiv:2310.14053 [cs]. Feb. 2024. URL: http://arxiv.org/abs/2310.14053 (visited on 10/14/2024).

[25] Tristan Coignion, Clément Quinton, and Romain Rouvoy. "A Performance Study of LLM-Generated Code on Leetcode". en. In: *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*. Salerno Italy: ACM, June 2024, pp. 79–89. ISBN: 9798400717017. DOI: 10.1145/3661167.3661221. URL: https://dl.acm.org/doi/10.1145/3661167.3661221 (visited on 10/14/2024).

[26] Balreet Grewal et al. "Analyzing Developer Use of ChatGPT Generated Code in Open Source GitHub Projects". en. In: *Proceedings of the 21st International Conference on Mining Software Repositories*. Lisbon Portugal: ACM, Apr. 2024, pp. 157–161. ISBN: 9798400705878. DOI: 10.1145/3643991.3645072. URL: https://dl.acm.org/doi/10.1145/3643991.3645072 (visited on 10/14/2024).

[27] Yuhao Zhang et al. *CodeFort: Robust Training for Code Generation Models*. en. arXiv:2405.01567 [cs]. Apr. 2024. URL: http://arxiv.org/abs/2405.01567 (visited on 10/14/2024).

[28] Xiaoli Lian et al. *Uncovering Weaknesses in Neural Code Generation*. en. arXiv:2407.09793 [cs]. July 2024. URL: http://arxiv.org/abs/2407.09793 (visited on 10/14/2024).

[29] Yangruibo Ding et al. *SemCoder: Training Code Language Models with Comprehensive Semantics*. en. arXiv:2406.01006 [cs]. June 2024. URL: http://arxiv.org/abs/2406.01006 (visited on 10/14/2024).

[30] M. El Hamlaoui et al. "A Model-Driven Approach to Align Heterogeneous Models of a Complex System". In: *The Journal of Object Technology* 20.2 (2021). Last accessed November 2024, pp. 1–24. DOI: 10.5381/jot.2021.20.2.a1. URL: https://www.jot.fm/issues/issue_2021_02/article1/.

[31] *Benchmarking (of Software Systems)*. https://www2.sigsoft.org/EmpiricalStandards/docs/standards?standard=Benchmarking. Last accessed October 2023. 2015.

[32] Shuo Ren et al. "Codebleu: a method for automatic evaluation of code synthesis". In: *arXiv preprint arXiv:2009.10297* (2020).

[33] Kishore Papineni et al. "Bleu: a Method for Automatic Evaluation of Machine Translation". In: *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*. Philadelphia, Pennsylvania, USA: Association for Computational Linguistics, 2002, pp. 311–318. DOI: 10.3115/1073083.1073135.

[34] "Large Language Models and Their Applications". In: (2023). Last accessed October 2023.

[35] *ChatGPT, OpenAI*. https://chat.openai.com/chat. Last accessed May 2023. 2022.

[36] *Introduction - OpenAI API*. https://platform.openai.com/docs/guides/completion/introduction. Last accessed May 2023. 2023.

[37] *Learn about the Gemini models*. https://firebase.google.com/docs/vertex-ai/gemini-models. Last accessed October 2023. 2023.

[38] *Introducing computer use, a new Claude 3.5 Sonnet*. https://www.anthropic.com/news/3-5-models-and-computer-use. Last accessed October 2023. 2023.

[39] X Bi, D Chen, G Chen, et al. "Deepseek LLM: Scaling Open-Source Language Models with Longtermism". In: *arXiv preprint arXiv:2401.02954* (2024). URL: https://arxiv.org/pdf/2401.02954.

[40] Hugo Touvron et al. "Llama 3: Open Foundation and Fine-Tuned Models". In: *arXiv preprint arXiv:2407.21783* (2024). Last accessed October 2023. URL: https://arxiv.org/abs/2407.21783.

[41] An Yang et al. "Qwen2.5 Technical Report". In: *arXiv preprint arXiv:2412.15115* (2024).

[42] Alexey Svyatkovskiy et al. “Intellicode compose: Code generation using transformer”. In: *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2020, pp. 1433–1443.

[43] Zeyu Sun et al. “Treegen: A tree-based transformer architecture for code generation”. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 34. 2020, pp. 8984–8991.

[44] Junkai Chen et al. “Code Search is All You Need? Improving Code Suggestions with Code Search”. en. In: *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. Lisbon Portugal: ACM, Apr. 2024, pp. 1–13. ISBN: 9798400702174. DOI: 10.1145/3597503.3639085. URL: https://dl.acm.org/doi/10.1145/3597503.3639085 (visited on 10/14/2024).

[45] Shirley Anugrah Hayati et al. “Retrieval-Based Neural Code Generation”. In: *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*. 2018.

[46] Md Rizwan Parvez et al. “Retrieval Augmented Code Generation and Summarization”. In: *Findings of the Association for Computational Linguistics: EMNLP 2021*. 2021, pp. 2719–2734.

[47] Disha Shrivastava. *Repository-level prompt generation for large language models of code*. https://arxiv.org/abs/2206.12839. Journal reference: ICML, 2023; arXiv:2206.12839 [cs.LG]. 2023. DOI: 10.48550/arXiv.2206.12839.