Computer Communications 212 (2023) 313–323

Contents lists available at ScienceDirect

Computer Communications

journal homepage: www.elsevier.com/locate/comcom

Enhancing intelligent IoT services development by integrated multi-token code completion

Yu Xia[a], Tian Liang[a], WeiHuan Min[a], Li Kuang[a,*], Honghao Gao[b,*]

[a] School of Computer Science and Engineering, Central South University, Changsha, 410083, PR China
[b] School of Computer Engineering and Science, Shanghai University, Shanghai, 200444, PR China

ARTICLE INFO

Dataset link: The artifacts of the S2RCC (Original data)

Keywords:
Intelligent device-free sensing
Code completion
Language model
Graph neural network

ABSTRACT

The Internet of Things (IoT) is a revolutionary network of interconnected devices embedded with sensors and software that enables seamless communication, data sharing, and intelligent decision-making in the form of IoT services. To facilitate the efficient development of IoT services, code completion technique provides a promising solution by providing suggestions for missing code snippets. The development trend of IoT services is to support more mobile device terminals. Mobile devices are portable and easy to use, allowing IoT device operation and management anytime and anywhere. However, the current multi-token completion methods struggle to guarantee code generation quality under the constraints of low resources and low latency, making it difficult to fully support IoT service development. We propose a multi-token code completion framework, S2RCC, which completes code from skeleton to refinement with dual encoder and dual decoder. The framework consists of two phases: first, the code skeleton, which is the simplification of code containing structure-sensitive tokens, is predicted based on the semantics of the code context; second, the broken context is repaired with the predicted skeleton, and then parsed into the code structure so that the specific tokens can be generated combining the semantics and structure of context. Furthermore, we then provide an implementation of the framework, representing the repaired code as an improved Heterogeneous code graph and fusing the semantics and structure of code context by the three-layer stacked attention. We conducted experiments on multi-token completion datasets, showing that our model has achieved the state-of-the-art with the smallest possible scale and the fastest generation speed.

1. Introduction

Internet of Things (IoT) is a new buzzword in information technology where real-world physical objects are made smart by integrating them with internet-enabled technologies. The things can sense information around them, communicate the sensed information over some protocol and employ the information to solve real-life problems [1]. IoT digitizes the real world, reshapes and revolutionizes every sphere from business to life, such as transportation and logistics, commercial manufacture, and smart environment.

A critical requirement of an IoT is that the things in the network must be inter-connected, therefore, IoT system architecture must guarantee the operations of IoT, which bridges the gap between the physical and the virtual worlds [2]. Service-oriented architecture (SOA) is a mainstream architecture for IoT, since it ensures the interoperability among heterogeneous devices in multiple ways. Fig. 1 presents a typical service-oriented architecture for IoT, which consist of four layers: (1) Sensing layer integrates available hardware objects to sense the statuses of things; (2) Network layer supports data transmission over wireless or wired connections among things; (3) Service layer creates and manages services required by users or applications; (4) Interfaces layer is the interaction methods among users and applications.

All of the service-oriented activities, such as information exchange and storage, management of data, ontologies database, search engines and communication, are performed at the service layer. It is crucial to develop code efficiently for IoT services. In this context, code completion emerges as a valuable tool to improve development efficiency by providing suggestions for missing parts of code snippets. By integrating code completion techniques into IoT service layer, developers can accelerate IoT services development and enhance the capabilities of intelligent sensing system.

Most specialized code completion methods [3–7] predict the next token or the next node of an Abstract Syntax Tree (AST). Recently, some

* Corresponding authors.
E-mail addresses: xiayu1999@csu.edu.cn (Y. Xia), leungtien@csu.edu.cn (T. Liang), csu_semwh@csu.edu.cn (W. Min), kuangli@csu.edu.cn (L. Kuang), gaohonghao@shu.edu.cn (H. Gao).

https://doi.org/10.1016/j.comcom.2023.10.014
Received 6 June 2023; Received in revised form 21 September 2023; Accepted 20 October 2023
Available online 24 October 2023

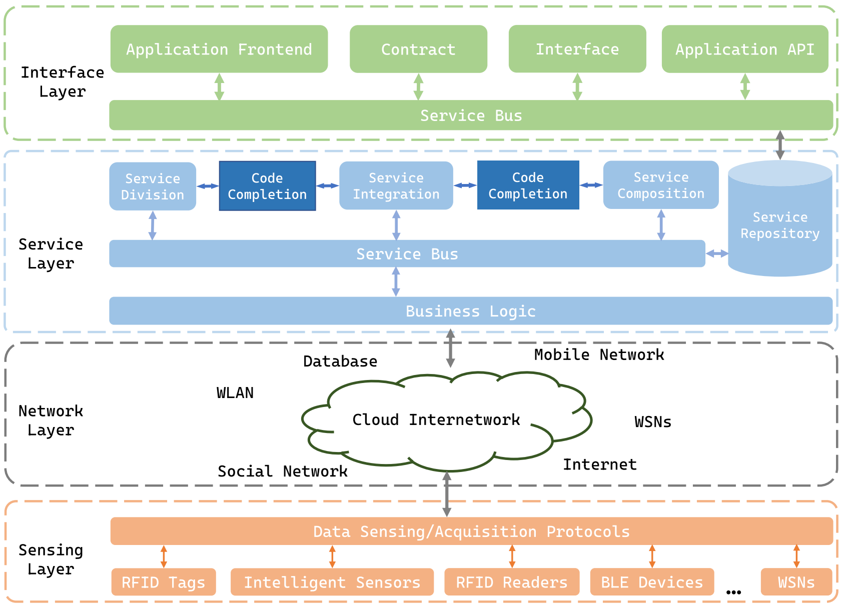
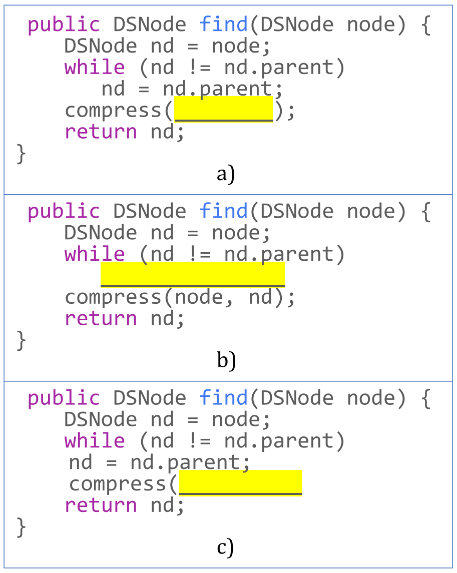


Fig. 1. An IoT architecture diagram with code completion.



```
 public DSNode find(DSNode node) {
    DSNode nd = node;
    while (nd != nd.parent)
       nd = nd.parent;
    compress(         );
    return nd;
}
```
a)

```
 public DSNode find(DSNode node) {
    DSNode nd = node;
    while (nd != nd.parent)

    compress(node, nd);
    return nd;
}
```
b)

```
 public DSNode find(DSNode node) {
    DSNode nd = node;
    while (nd != nd.parent)
    nd = nd.parent;
    compress(
    return nd;
}
```
c)

Fig. 2. Three situations for multi-token code completion (a) Construct-level (b) Block-level (c) Token-level.

general-purpose large-scale language models (LMs), such as Codex [8] and ChatGPT [9], have shown excellent capabilities in code generation. They focus on auto-regressive code generation, which cannot handle all scenarios in Integrated Development Environments (IDE), as programming is not always from the top down. Multi-token code completion can provide suggestions for an entire span when developers review or modify the code. Fig. 2 illustrates the three most typical situations of multi-token completion: construct-level (completing the code structure inside parentheses, e.g., parameters of a method call, Fig. 2a), block-level (completing an entire block inside curly braces, e.g., the body of a while statement, Fig. 2b), and token-level (completing the last few tokens within a line of code, Fig. 2c). We expect to support multi-token code completion, including these typical situations.

Existing research of multi-token code completion [5,10–14] are based only on LMs. These methods and large-scale pre-training LMs are strenuous to perform satisfactorily under low resources. LMs demand a large number of parameters to ensure their effectiveness. Although their smaller versions are released, the performances are often vastly inferior to their full one [15–17]. Some LLMs [9,18,19] support code completion, but they have many parameters, meaning higher computational resources and inference time. Moreover, despite this, their performance still falls short compared to some expert models. In contrast, IoT services are typically oriented towards lightweight, low-resource scenarios, and require immediate response, making the deployment of large-scale LMs challenging. It is indicated that the LM-only methods do not adequately support the applications of multi-token completion on IoT.

Under the limitation of low resources, introducing Graph Neural Networks (GNN) as auxiliaries can significantly improve small LMs [20, 21]. Separating the semantics and the structure of the code and leaving them to the LM and the GNN, respectively, can make the functions relatively definite and relieve the pressure on the LM. Some studies of completion on AST [6,7,22] have shown that models incorporating GNN outperform only-LM models. However, it is challenging to consider structure in multi-token code completion, since the broken code context may contain an illegal code span causing the wrong syntax, e.g., Fig. 2 (c): missing one of a pair of separators, the static analyzers are hard to parse the broken code context as a structure.

Developers' programming behavior may provide a solution to convert the illegal code context into a structure. Take Fig. 2(c) as an example. To complete the highlighted fields, developers would first fill the right bracket according to the separators, then analyze the number of parameters of *compress*, getting *?,?*, as the skeleton(abstract version of the code) of the target span(detailed version of the code). Finally, the structure of a whole code would be carefully analyzed to refine the actual identifiers of the parameters. This reveals a two-stage action, which matches the idea of dealing with code syntax and structure separately and can complete all code context, even though it loses any number of tokens in any location.

Inspired by this action, we propose the From **S**keleton **to Re**finement **C**ode **C**ompletion (S2RCC) Framework. The overview of S2RCC framework is shown in Fig. 3, which consists of dual encoder and dual decoder. It decomposes multi-token code completion into two phases: Skeleton Prediction and Span Refinement. In the Skeleton Prediction, the code context is input to the Context Encoder to obtain the semantic representation of code. The Skeleton Decoder then generates the span skeleton, a sequence containing tokens closely related to

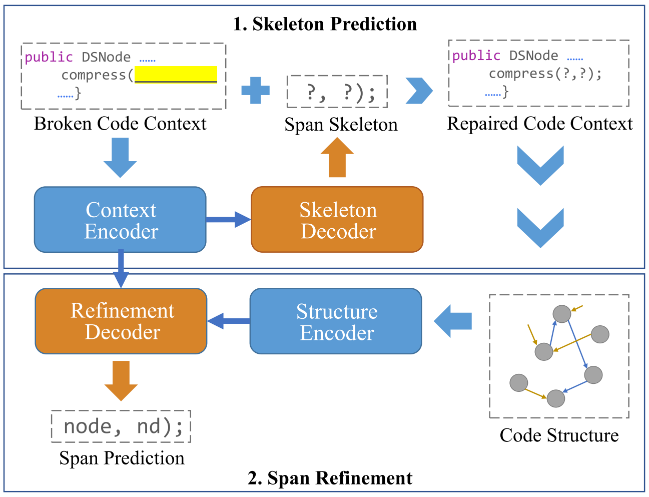


Fig. 3. Overview of S2RCC framework.

the structure (e.g., separators) but no specific tokens (e.g., identifiers). In the Span Refinement, the skeleton is applied to repair the broken context so that it can be parsed into the code structure, which is the input of the Structure Encoder. Finally, the semantics and structure are fused in the Refinement Decoder to generate the Span Prediction.

We explore a standard implementation of S2RCC framework — S2RCC model. It takes the modified Transformer [15,23] as the Context Encoder and Skeleton Decoder, GGNN [24] as the Structure Encoder. In addition, for Structure Encoder, because the homogeneous graph proposed by existing code completion studies [7,25] is not reasonable for multi-token completion, we propose the heterogeneous code graph for S2RCC, which can represent code structures properly. As for Refinement Decoder, we propose the stacked three-layer attention to fuse semantics and structure. Note: we have not restricted the S2RCC framework to a specific implementation, and S2RCC model is one of the feasible solutions after our comprehensive consideration.

The codes and predictions of S2RCC are publicly available and anonymized during the review.[1]

Our main contributions are:

- We propose the S2RCC framework. It completes codes in two phases: Skeleton Prediction and Span Refinement. S2RCC is the first solution for capturing code structure rebuilt by the predictive skeleton in multi-token code completion.
- We propose an implementation of S2RCC framework: S2RCC model, which employs the heterogeneous graph to represent code structure and three-layer stacked attention to fuse the code semantics and structure.
- Experiments show that: S2RCC model trained from scratch outperforms the multi-token completion methods, including LLMs, in the construct-level and block-level on the premise of similar results in the token-level; both after pre-trained, S2RCC performs better than all models in all levels; wrapping small pre-training LMs with S2RCC framework can significantly improve their performances, which proves that our framework can be widely migrated.

2. Related works

2.1. Code generation for the internet of things

In recent years, with the rapid development of the Internet of Things (IoT), an increasing amount of research has been focused on leveraging intelligent code generation techniques to assist in the development of smart devices and services, thereby accelerating the iterative evolution of the IoT. Cpp-tiny-client [26] focuses on generating platform-specific API client code for IoT devices. It introduces cpp-tiny-client as a plugin for the OpenAPI Generator project, enabling developers to tailor the generated code to the specified IoT platform. CAPSml [27] allows designers and architects using the CAPS environment to transform CAPS software models into ThingML models. ThingML provides code generation capabilities for multi-platform targets, simplifying the integration of heterogeneous devices and networks in IoT systems. CAPSml streamlines the development process for IoT designers and architects. These studies aim to introduce intelligent software methods into the IoT to address its rapid growth and ever-changing demands. By applying them, researchers are able to automatically generate high-quality and customizable code, thereby improving development efficiency, reducing error rates, and expediting the launch of new products and services.

Other common code completion methods can be divided into token-level completion and AST-level completion. Some pre-trained models can support code completion, likewise.

2.2. Token-level completion

Token-level completion treats code as a token sequence or type sequence. Following the design philosophy, we elaborate on three token-level code completion methods: expert models, LLMs, and sketch-style approaches. The expert models are specifically designed and trained for code completion. Svyatkovskiy et al. [3] propose a general framework that encodes the code diversely, while Liu et al. [4] propose a pre-training model for code completion based on multi-task learning. Izadi et al. [5] take the type sequences as supplementary of tokens. Ciniselli et al. [11] and Ciniselli et al. [10] explore the potential of the Roberta and T5 on multi-token code completion, respectively. Liu et al. [13] design a non-autoregressive model to complete a whole line. Lu et al. [28] introduce code retrieval into code completion. Recently, some LLMs have also focused on the fill-in style code completion tasks. Incoder [18] highlighted the importance of completion during editing, supporting code filling. SantaCoder [19] achieved better fill-in results with a smaller footprint. Additionally, the Codex's [8] variant code-davinci-002 can also support code filling. Several token-level completion models have adopted the concept of code sketching, which involves creating a rough version of the code to various extents. However, their motivations differ. Guo et al. [14] advocates sketching the code and leaving uncertain tokens for developers to handle, positing that the generated code should retain flexibility. On the other hand, Li et al. [29] introduces SKCODER, which surpasses other methods by effectively replicating the developer's tendency to reuse code and optimizing similar code snippets. **In summary, token-level completion mostly has not utilized code structure because of the challenge of parsing broken codes.**

2.3. AST-level completion

AST-level completion relies on the AST, given the completed structure, continuously predicting the next node. Liu et al. [6] define the path to the root as an auxiliary objective. Kim et al. [22] compare entering merely token sequences, sequences with the root paths, and sequences with the DFS paths. Wang et al. [12] convert AST into generating actions. Wang and Li [7] construct a graph including node–node and parent–child edges and then input it into GAT. Liu et al. [30] propose a framework that can handle AST-level and Token-level completion. **Methods for AST-level completion cannot be adapted to arbitrary multi-token completion.** The AST will be fatally broken if a structure-sensitive token is dropped, such as a closing parenthesis.

[1] https://figshare.com/s/a6c3f08c6d47ad517aa0.

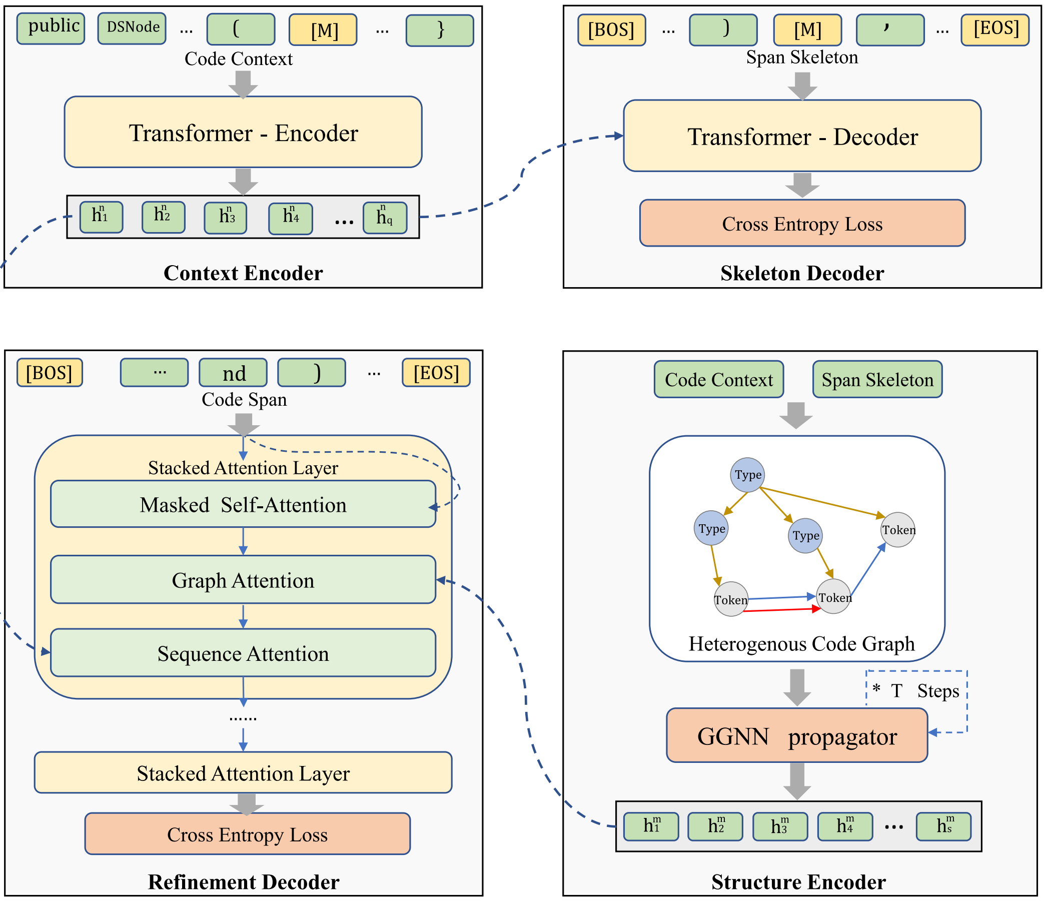


Fig. 4. Overview of S2RCC model (training).

2.4. Pre-training models

Some pre-training models widely support code generation. CodeBERT [31] and CodeT5 [32] gather Programming Language (PL) and Natural Language (NL) for pre-training and support various NL-PL downstream tasks. GraphCodeBERT [20] further considers data flow. Codex [8] is based on GPT-3 [33], pre-trained by extensive code corpus. ChatGPT [9] is the latest chat model derived from GPT-3, and users can enter prompts instructing it to complete a snippet of code. Unfortunately, none of these GPT-3 models is open, and no light versions have been released. Although there are accessible APIs that can serve, their vast scale makes supporting immediate response in an offline, low-resource development environment impossible. OPT's [34] performance is comparable to GPT-3, and all its versions are open. The evaluation of OPT can, to some extent, reflect the capability of the GPT-3 series. In general, **these pre-training models do not entirely compete with S2RCC.** They can be alternatives to components of the S2RCC framework (see Fig. 4).

3. Approach

Multi-token code completion can be formally defined as: given the code context $[x_1, \dots, x_m]$, and the position of target span *pos*, to solve the mapping f: from the context and position to the target span $[y_1, \dots, y_n]$.

3.1. S2RCC framework

We propose the S2RCC framework to achieve multi-token code completion. S2RCC is a dual encoder–dual decoder architecture, whose overview is shown in Fig. 3. S2RCC consists of four core components: Context Encoder, Skeleton Decoder, Structure Encoder and Refinement Decoder. In practice, the Context Encoder, Skeleton Decoder and Refinement Decoder could be any sequence modeling approach, while Structure Encoder could be any relation modeling approach. S2RCC divides the code completion into two phases: Skeleton Prediction and Span Refinement.

Code Skeleton. The code skeleton is a unique mapping of the code sequence. Code tokens can be converted into a vector with an intact vocabulary V_{tk}, while the skeleton vocabulary V_{sk} is a subset of V_{tk}. Thus, a token sequence corresponds to a unique skeleton sequence through the following transformation:

$$\begin{aligned} & V_{\mathrm{sk}} \subset V_{tk} \\ & f_{sk}\left(y_i\right) \rightarrow \begin{cases} y_i & , y_i \in V_{\mathrm{sk}} \\ C_{\mathrm{hole}} & , y_i \notin V_{\mathrm{sk}} \end{cases} \\ & \left[s_1, \dots, s_n\right] = \left[f_{sk}(y_1), \dots, f_{sk}\left(y_n\right)\right] \end{aligned} \tag{1}$$

f_{sk} is the mapping from the token sequence to the code skeleton, C_{hole} is a "hole" marker in the skeleton, and $[s_1, \dots, s_n]$ is the skeleton. Briefly, a code skeleton is a "simplification" of the code. As the output of the first phase, only tokens in the skeleton vocabulary are retained, and others will be replaced by "hole", waiting for refinement. V_{sk} can be customized. Most simplified, it only includes the structure-sensitive tokens, or more complex, contains all except intricate tokens that are burdensome to the language models.

We have not rigidly defined which tokens belong to the skeleton at the framework level, aiming for a more flexible interpretation. Hence, the code abstraction termed "templates" or "sketches" in studies like [14,29] might fit our skeleton definition. For example, the skeleton might resemble the "sketch" of Guo et al. [14] when only hard-to-predict tokens are excluded. We emphasize the purpose of our skeleton definition, distinct from existing templates, focusing on structural restoration and information integration.

Skeleton Prediction. In this phase, as a language model, the Context Encoder encodes the broken code context into its semantic representation. Next, the Skeleton Decoder predicts the span skeleton based on this semantic. Then the skeleton is filled back into the context, so the code structure is repaired. Predictions of skeletons serve as an auxiliary task to forward losses. In summary, the goal of Skeleton Prediction is not to predict the fine-grained code spans but rather their simplified skeletons.

Span Refinement. In this phase, the repaired code context can be parsed into a code structure that is as close to its original. The Structure Encoder encodes the code structure as a representation that considers local rational biases. Finally, the Refinement Decoder fuses semantics and structure, combining representations from the two encoders to get a final span prediction. Two keys of this phase are: choosing the appropriate code structure and relational model and integrating features from code semantics and structure.

S2RCC Framework is inspired by developers' filling of code spans, following the perception from skeleton to refinement. With this framework, the code structure can be rebuilt in multi-token code completions despite any number of tokens lost at any position. We did not limit implementations for the S2RCC framework, which means that the components can be freely specified and combined. We hope the S2RCC represents a two-phase multi-token code completion style that is universal and evolvable.

3.2. S2RCC model

We propose a standard implementation of the S2RCC framework: the S2RCC model, to evaluate its performance. The S2RCC model applies an improved transformer-encoder as the Context Encoder and an improved transformer-decoder as Skeleton Decoder, which predict structure-sensitive tokens. As for Structure Encoder, we propose the heterogeneous code graph to represent the structure and take GGNN [24] as a propagator. For the Refinement Decoder, We employ the three-layer stacked attention to fuse semantics and structure.

Skeleton Definition. In the S2RCC model, we define code skeleton as its simplest form: only structure-sensitive tokens kept, whose modification may cause the AST's reconstruction. Specifically, we retain Operators, Modifiers, Separators, Annotations, and most of the Keywords. Other tokens will be replaced with a unique token *[M]*. Taking the code span: "*node,nd;*" as an example, its skeleton would be: "*[M],[M];*". Defined this, the skeleton is enough to repair the broken context so the context can be parsed on the one hand. On the other hand, we can minimize the vocabulary and ease the burden of the Skeleton Decoder to achieve an immediate response.

Context Encoder & Skeleton Decoder. The Context Encoder and Skeleton Decoder are based on the encoder and decoder of transformer [15], respectively, but with some minor adjustments. We modify the position embedding to relative position embedding because the relationship between tokens depends more on the relative position. In addition, following So et al. [23], we alter the activation of the position-wise feed-forward to the square of Relu for efficiency.[2]

$$y = max(Relu(X), 0)^2 \tag{2}$$

Structure Encoder. For Structure Encoder, we propose the heterogeneous code graph to represent the code structure, an example of which is shown in Fig. 5. Existing code structures proposed for code completion are not reasonable for multi-token. They model ASTs as homogeneous representations, i.e., each leaf is defined as TYPE and VALUE, and those non-leaves as TYPE | EMPTY [7,25]. Firstly, there are many non-leaves, but the EMPTY has limited significance. Secondly, as multi-token completion does not predict types, it makes sense for models to focus on VALUE rather than TYPE, but mixing them is confusing.

We take the VALUE of leaves of AST as Token nodes and the TYPE of non-leaf nodes as Type nodes to form nodes of the heterogeneous code graph. Following Ling et al. [35], the heterogeneous code graph comprises edges of Child, NextToken, and NextUse. The Child represents the parent–child relationship of the AST. The NextToken connects the Token nodes one by one, and the NextUse points a Token to its next appearance. Moreover, we creatively consider the "roles" of nodes, which differs from existing approaches. We define the "parent type – child role" join as the child's role to separate parent–child relationships essentially. Taking Fig. 5 as an example, the Type *Assignment* has three children: *MemberReference(nd)*, *MemberReference(nd.parent)*, =, which serve as *expression*, *value* and *type*, respectively. We extract their roles (e.g., *MemberReference_type* as the role of =) and embed them separately.

To represent nodes in the graph, we define: Token Embedding E_{token}, Type Embedding E_{type}, and Role Embedding E_{role}. Wherein, Role Embedding serves as position embedding. The node representation is expressed formally as Eq. (3).

$$\mathrm{n}_t(x, r) = \begin{cases} \frac{\sum_{i=1}^{N} E_{token}(x_i)}{N} + E_{role}(r) & , x \in V_{token} \\ E_{type} + E_{role}(r) & , x \in V_{type} \end{cases} \tag{3}$$

where x and r are determined by node values and roles, respectively. n_t is an R-d vector. V_{token} and V_{type} are vocabularies of tokens and types. x is decomposed into N subtokens: $x_1 \ldots x_n$.

We then employ GGNN [24] to encode the code structure, in which all nodes are propagated forward and reverse along the edges separately, after projected onto the same vector space.

Refinement Decoder. As for Refinement Decoder, because we expect the model to be self-repaired and not over-reliant on skeleton predictions, we use the transformer-decoder to re-predict an entire span. We propose the three-layer stacked attention: Masked Self-attention, Graph Attention, and Sequence Attention, allowing the decoder to boost from span itself, structure, and context semantics, jointly. To simplify, we define the multi-head attention as $Attn(Q, K, V)$, and the masked multi-head self-attention as $MaskAttn(Q, K, V)$. The layer l is calculated as shown in Eq. (4), where E_t^{l-1} is the output of layer $l-1$. H^n and H^s are the outputs of the Structure Encoder and Context Encoder, respectively.

$$\begin{aligned} O_t^l &= Norm(E_t^{l\text{-}1} + MaskAttn(E_t^{l\text{-}1}, E_t^{l\text{-}1}, E_t^{l\text{-}1})) \\ O_{t \varsigma g}^l &= Norm(O_t^l + \mathrm{Attn}(O_t^l, H^{\mathrm{n}}, H^{\mathrm{n}})) \\ O_{t \varsigma g \varsigma s}^l &= Norm(O_{t \varsigma g}^l + Attn(O_{t \varsigma g}^l, H^s, H^s)) \end{aligned} \tag{4}$$

Finally, the S2RCC model optimizes the loss from skeleton and refinement in tandem, which are summed by constant weights, as is shown in Eq. (5).

$$l = C_{aux} \times l_{aux} + C_{main} \times l_{main} \tag{5}$$

4. Experimental setup

4.1. Datasets

We directly obtain the multi-token code completion fine-tuning datasets[3] collected by Ciniselli et al. [10,11], including sources from Java and Android, subdivided into construction-level, block-level, and token-level, and follow their initial training, testing, and validation division. We follow their guide to create a pre-training dataset from

[2] It should be noted that although applying the pre-training LMs here achieve better results (Section 5), we cannot promise their pre-training dataset not intersect with the code completion test set. We employ the original transformer to have more control over the training dataset for fair comparisons with the baselines.

[3] https://github.com/mciniselli/T5_Replication_Package.

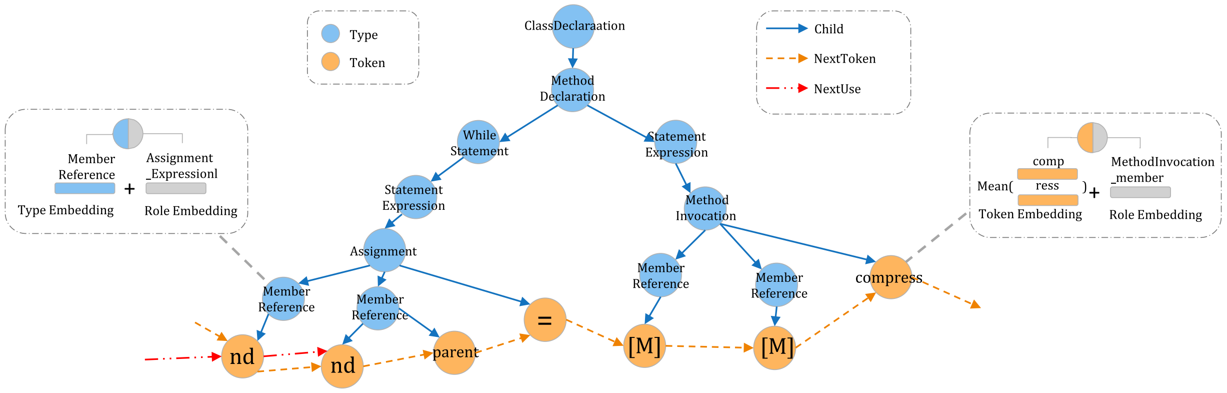


Fig. 5. An example of a heterogeneous Code Graph and its representation of Token and Type nodes.

CodeSearchNet[4] [36] as their pre-training dataset is not publicly available. We preprocess the datasets. First, extract their span skeletons. Then, to improve efficiency for the training set, we build the heterogeneous code graphs in advance, based on the AST parsed by Javalang.[5] All of the above artifacts are under the MIT License.

4.2. Baselines and metrics

We mainly consider the following models as baselines for S2RCC:

n-gram [37] is a statistical language model. We regard it as a representative of traditional methods.

Roberta-CC [11] is trained from the same dataset. They evaluate the effect of a BERT-style auto-encoding model on multi-token code completion.

T5-CC [10] achieves the best results on the same dataset, and it is the SOTA on multi-token completion.

Transformer [15] is based on the self-attention mechanism entirely. We select two versions: the transformer-86M as a baseline to reflect our improvements, and the transformer-172M to compare the performance of a similar size with S2RCC.

CCAG [7] is the latest model designed for next-token completion. We applied their homogeneous graph as a comparison to our proposed heterogeneous graph.

Graph-Sandwich & **GREAT** [38] are proposed for code representation. They have similar ideas with S2RCC: combining code semantics and structure. We migrated them to multi-token code completion to measure our stacked-attention fusion.

CodeT5 [32] is a unified encoder–decoder pre-training model for code understanding and generation. We bring it as a representative of the code-focused pre-training model and evaluate the performance of its small version.

OPT [34] is an open large LM whose full version rivals GPT-3 [33]. We take the OPT-125M as a substitute for these GPT-3-based auto-regressive models and evaluate its performance in limited parameters.

InCoder [18] is a unified code LLM that supports code generation and editing. During training, code regions are randomly masked and moved to the end of each file, allowing for bidirectional context in code infilling. We use InCoder-1B as one of the baselines for code LLMs.

SantaCoder [19] found that more aggressively filtering near-duplicate items can further enhance performance. Their best model outperforms previous models in left-to-right generation and infilling. We choose SantaCoder-1B as one of the baselines for code LLMs.

Code-davinci-002 is the most powerful variant of Codex [8]. Besides completing the code, it also supports inserting completions within the code. Regrettably, OpenAI has currently discontinued access support for code-davinci-002 and recommends using GPT3.5-Turbo instead. Therefore, we follow their advice and choose GPT-3.5-Turbo-0613 as one of the baselines for LLMs.

For metrics, We compute Perfect Predictions, BLEU1-4, CodeBLEU, and average inference time for the baselines and S2RCC.

Perfect Predictions are cases when the predicted code exactly matches its answers. The percentage of perfect predictions is called Perfect.

BLEU, Bilingual Evaluation Understudy, can reflect the quality of text generation. BLEU-n indicates the cumulative score from 1-gram to n-gram.

CodeBLEU is better suited for evaluating code generation quality. It absorbs the strengths of BLEU and further considers AST and data flow to evaluate code syntax and semantics.

Time in this paper refers to the average inference time. On a 3090 machine, we conducted one round of inference for either an LLM or S2RCC, summing the computation times and then averaging them.[6]

4.3. Model configuration

We make a similar configuration with T5-CC for the S2RCC model. Detailed hyperparameters are in Table 1. The S2RCC model with this configuration has **163M** parameters.

There are other configurations specific to the S2RCC model. During the test, an illegal skeleton leads to the incorrect syntax of the context in some rare cases, in which its graph will be replaced with an empty one. Accordingly, during training, each instance has a 1% probability to be occupied by an empty graph to inspire the model to handle illegal skeletons. Concerning pre-training, inspired by Replace Spans [39], we randomly select 5–20 tokens from the code as a sequence, extract its skeleton and build a heterogeneous code graph accordingly to generate an instance of multi-token completion.

There are also some unique configurations for other baselines. As for Graph-Sandwich and GREAT [38], they are designed for intact code structure. We can only evaluate them on construct-level and block-level datasets by replacing their neat missing spans with *[M]*. For CodeT5 [32] and OPT [34], we limit their parameter amount to simulate low-resource scenarios of IoT, so we select the smallest versions of them respectively (CodeT5-small and OPT-125M) and then fine-tune them. If available from their paper, we apply their optimal fine-tuning configuration for baselines. Otherwise, configure them the same as S2RCC.

We also evaluated the performance of several LLMs that support code filling under different levels of completion. For InCoder [18],

[4] https://github.com/github/CodeSearchNet.
[5] https://github.com/c2nes/javalang.

[6] The time recorded for GPT-3.5-Turbo might be longer, as we could not eliminate the error caused by network fluctuations.

Table 1
Important parameters of S2RCC.

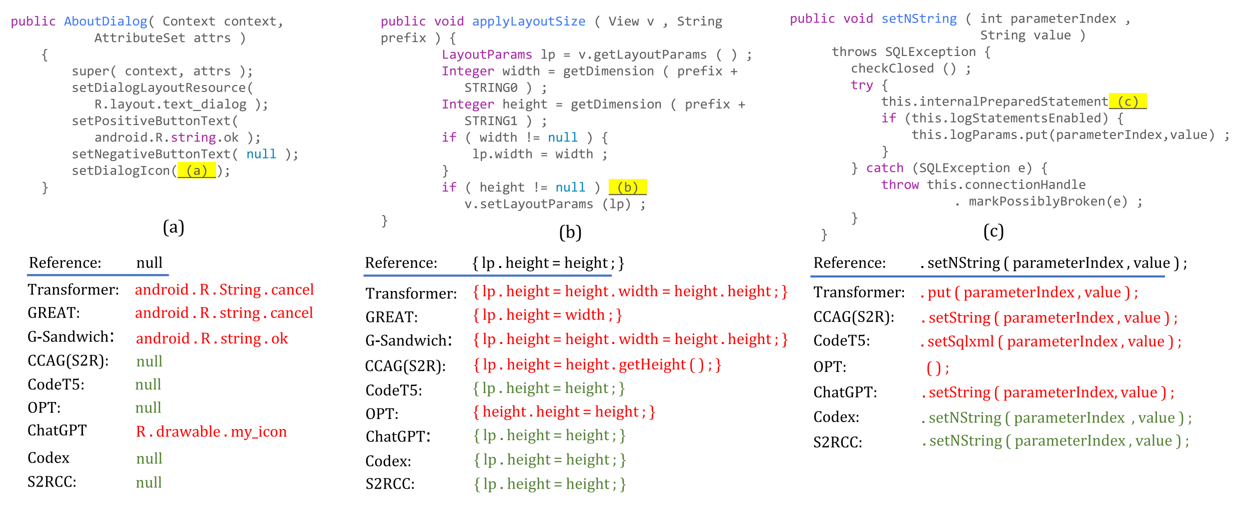
Parameter	Value	Parameter	Value	Parameter	Value	Parameter	Value
max epochs	200	max drop token	20	TF.num_layers	6	token v_size	80 000
early stop	5	min drop token	5	TF.num_heads	8	role v_size	165
optimizer	Adam	src.max_len	420	TF.d_ff	2048	ske v_size	79
pre-train lr	1e−5	tgt.max_len	80	TF.d_k	64	type v_size	60
pre-train steps	400 000	max_relative	64	TF.d_v	64	emb.size	512
lr	5e−5	C_{main}	1	TF.dropout	0.2	emb.dropout	0.2
stable epochs	10	C_{aux}	1	TF.width	512	max sub-node	7
decay	0.97	drop graph	0.01	GGNN.steps	6	valid metric	Perfect

Table 2
Perfect predictions(left) and CodeBLEU(right) of S2RCC and other baselines.

	Construct				Block				Token			
	JAVA		Android		JAVA		Android		JAVA		Android	
n-gram	20.5%	\	22.0%	\	8.5%	\	11.9%	\	43.9%	\	42.2%	\
Transformer	36.6%	92.13	31.7%	91.82	15.7%	83.18	14.7%	85.09	54.8%	94.26	56.2%	94.22
RoBERTa-CC	33.4%	90.66	37.4%	92.66	8.7%	76.19	9.4%	?%	38.9%	89.29	51.8%	92.64
CodeT5	39.7%	92.37	32.2%	91.31	16.2%	85.02	16.4%	84.56	54.9%	94.27	58.1%	94.50
OPT	32.0%	91.45	29.5%	91.45	15.3%	84.67	13.9%	84.81	51.5%	93.73	52.9%	93.98
T5-CC	48.4%	\	46.8%	\	22.9%	\	22.8%	\	**61.0%**	\	**63.8%**	\
S2RCC	**51.6%**	**94.81**	**48.0%**	**94.62**	**24.5%**	**88.79**	**23.3%**	**88.83**	60.2%	**95.98**	62.3%	**96.26**

Table 3
Metrics of LLMs and S2RCC.

DataSet	LLM	Perfect	Bleu1	Bleu2	Bleu3	Bleu4	CodeBLEU	Time
Construct	InCoder	37.60	61.19	57.81	**55.77**	**53.87**	89.91	192.83
	SantaCode	35.60	50.80	49.23	47.80	46.25	83.82	189.10
	GPT3.5-trubo	33.83	61.86	**57.90**	55.36	53.17	91.81	942.19
	S2RCC	**51.60**	**68.25**	56.19	47.62	42.41	**94.81**	**106.27**
Block	InCoder	1.58	24.60	21.80	20.17	19.06	74.02	197.08
	SantaCode	16.25	52.44	51.69	49.87	**48.54**	80.92	201.15
	GPT3.5-trubo	12.84	53.61	49.32	45.32	44.32	82.20	1218.76
	S2RCC	**24.50**	**67.32**	**59.15**	**51.22**	42.84	**88.79**	**124.62**
Token	InCoder	5.69	16.67	15.40	14.72	12.78	78.76	193.77
	SantaCode	40.71	56.92	51.67	50.59	**46.83**	86.98	181.24
	GPT3.5-trubo	15.39	43.83	38.53	35.76	32.84	88.04	914.02
	S2RCC	**60.20**	**82.19**	**71.83**	**59.17**	45.90	**95.98**	**85.12**



```
public AboutDialog( Context context,
        AttributeSet attrs )
    {
        super( context, attrs );
        setDialogLayoutResource(
           R.layout.text_dialog );
        setPositiveButtonText(
           android.R.string.ok );
        setNegativeButtonText( null );
        setDialogIcon( (a) );
    }
```

(a)

Reference:	null
Transformer:	android . R . String . cancel
GREAT:	android . R . string . cancel
G-Sandwich:	android . R . string . ok
CCAG(S2R):	null
CodeT5:	null
OPT:	null
ChatGPT	R . drawable . my_icon
Codex	null
S2RCC:	null

```
public void applyLayoutSize ( View v , String
prefix ) {
        LayoutParams lp = v.getLayoutParams ( ) ;
        Integer width = getDimension ( prefix +
           STRING0 ) ;
        Integer height = getDimension ( prefix +
           STRING1 ) ;
        if ( width != null ) {
            lp.width = width ;
        }
        if ( height != null )  (b) 
           v.setLayoutParams (lp) ;
}
```

(b)

Reference:	{ lp . height = height ; }
Transformer:	{ lp . height = height . width = height . height ; }
GREAT:	{ lp . height = width ; }
G-Sandwich:	{ lp . height = height . width = height . height ; }
CCAG(S2R):	{ lp . height = height . getHeight ( ) ; }
CodeT5:	{ lp . height = height ; }
OPT:	{ height . height = height ; }
ChatGPT:	{ lp . height = height ; }
Codex:	{ lp . height = height ; }
S2RCC:	{ lp . height = height ; }

```
public void setNString ( int parameterIndex ,
                         String value )
    throws SQLException {
      checkClosed () ;
      try {
          this.internalPreparedStatement (c) 
          if (this.logStatementsEnabled) {
              this.logParams.put(parameterIndex,value) ;
          }
      } catch (SQLException e) {
          throw this.connectionHandle
                   . markPossiblyBroken(e) ;
      }
  }
```

(c)

Reference:	. setNString (parameterIndex , value) ;
Transformer:	. put (parameterIndex , value) ;
CCAG(S2R):	. setString (parameterIndex , value) ;
CodeT5:	. setSqlxml (parameterIndex , value) ;
OPT:	( ) ;
ChatGPT:	. setString (parameterIndex, value) ;
Codex:	. setNString (parameterIndex , value) ;
S2RCC:	. setNString (parameterIndex , value) ;

Fig. 6. Qualitative examples of S2RCC and other baselines.

and SantaCoder [19], we processed the prefixes and suffixes of the code, converting them into their supported filling formats. For GPT-3.5-Turbo-0613 [9] (a substitute for code-DaVinci-002), we obtained completion results by accessing the API provided by OpenAI. We added the instruction for each instance: "Fill in the missing span at <MASK> in the following Java code snippet. You only need to indicate the missing part, but not to explain or repeat the code". We then replaced the missing portions in the code with <MASK>.

5. Research questions and results

In this section, we conduct experiments to investigate the following research questions:

RQ1: How does the S2RCC model perform compare to other methods in multi-token code completion?

We divide baselines into four categories and evaluate their performances: traditional methods including n-gram [37] and Transformer

Table 4
Performances of S2RCC, pre-trained T5-CC and pre-trained S2RCC. Pre: pre-trained.

		Method	Perfect	BLEU1	BLEU2	BLEU3	BLEU4
Construct	Java	S2RCC	51.6%	0.68	0.56	0.48	0.42
		Pre T5-CC	51.2%	0.70	**0.58**	**0.51**	0.40
		Pre S2RCC	**54.6%**	**0.71**	**0.58**	0.50	**0.44**
	Android	S2RCC	48.0%	0.67	0.57	0.49	0.45
		Pre T5-CC	49.3%	0.70	**0.60**	0.51	0.45
		Pre S2RCC	**51.5%**	**0.70**	0.59	**0.51**	**0.47**
Block	Java	S2RCC	24.5%	0.67	0.59	0.51	0.43
		Pre T5-CC	27.2%	0.68	0.61	0.54	0.46
		Pre S2RCC	**28.4%**	**0.69**	**0.62**	**0.55**	**0.47**
	Android	S2RCC	23.3%	0.65	0.56	0.49	0.40
		Pre T5-CC	27.5%	0.66	0.59	0.52	0.44
		Pre S2RCC	**28.2%**	**0.67**	**0.60**	**0.54**	**0.45**
Token	Java	S2RCC	60.2%	0.82	0.72	0.59	0.46
		Pre T5-CC	62.9%	0.84	0.75	0.63	0.51
		Pre S2RCC	**65.3%**	**0.85**	**0.77**	**0.64**	**0.52**
	Android	S2RCC	62.8%	0.84	0.75	0.62	0.49
		Pre T5-CC	64.8%	0.85	0.77	0.66	0.53
		Pre S2RCC	**67.6%**	**0.87**	**0.79**	**0.68**	**0.56**

Table 5
Statistics of S2RCC's experimental results. Pre N/Y: without or with pre-training.

Statistics	Pre	Construct	Block	Token
Perfect prediction	N	49.86%	24.00%	61.48%
	Y	**53.10%**	**28.33%**	**66.41%**
Perfect skeleton	N	71.20%	39.14%	77.43%
	Y	**72.46%**	**41.40%**	**79.90%**
Illegal skeleton	N	0.14‰	**0.48‰**	2.41‰
	Y	**0.31‰**	0.33‰	**11.34‰**
Self-repaired	N	**0.39‰**	0.69‰	1.00‰
	Y	0.37‰	**0.92‰**	**4.10‰**
Chi-square	N	82 656	32 291	191 401
	Y	**88 471**	**36 704**	**195 084**

Table 6
Perfect predictions of S2RCC and that after components replaced.

	Construct		Block		Token	
	Java	Android	Java	Android	Java	Android
S2R-CCAG	45.1%	44.9%	17.4%	16.9%	55.7%	60.4%
S2R-GCN	21.4%	19.7%	10.8%	10.2%	57.9%	36.6%
S2R-HGT	22.4%	20.5%	10.6%	10.9%	45.2%	46.8%
S2R-GAT	22.2%	20.3%	10.7%	10.1%	47.9%	47.8%
G-Sandwich	36.4%	26.3%	15.0%	13.2%	–	–
GREAT	33.6%	34.5%	11.4%	12.8%	–	–
S2RCC(GGNN)	**51.6%**	**48.0%**	**24.5%**	**23.3%**	**60.2%**	**62.3%**

[15], pre-training models including CodeT5 [32] and OPT [34], state-of-the-art methods on the same task and dataset including RoBERTa-CC [11] and T5-CC [10], LLMs including InCoder [18], SantaCoder [19] and GPT3.5-Turbo [9]. For the former two, we train or fine-tune them on the same dataset; for the third one, we directly report their results from their papers; for the LLMs, because of the substantial computational resources or costs required by LLMs, we only evaluated their performance on Java under Zero-Shot. Table 2 shows the perfect predictions of S2RCC[7] and other baselines for multi-token completion when both are trained or fine-tuned from scratch.

Analyzing Table 2, whether from the Perfect or the CodeBLEU metrics,[8] S2RCC is superior to all other baselines in construct-level and block-level completion. This proves that S2RCC is more advisable for dealing with long-span code completion. Further, it demonstrates that introducing code structure into multi-token completion is helpful compared to the mere language models. We also find that pre-training models, such as CodeT5 [32] and OPT [34], perform unsatisfactorily in their small versions. This also proves that language models demand sufficient parameters to exploit their advantages. From another perspective, OPT does not perform well, perhaps because it is unidirectional and autoregressive, with poor performance in the code filling.

Next, we assessed the performance of baseline LLMs on code filling across three levels, as shown in Table 3. Despite the vast parameter size of LLMs, their Perfect scores across the three levels of completion were significantly lower than S2RCC. The inference time also indicates that, compared to LLMs, S2RCC offers the advantage of immediate response in low-resource settings. For BLEU3 and BLEU4, LLMs score higher than S2RCC, which might be because LLMs tend to output longer completion texts. However, on CodeBLEU, which indicates code data flow and structure, S2RCC still surpasses LLMs. Compared to LLMs, S2RCC shows a significant advantage in both generation speed and quality.

We also notice that S2RCC without pre-training is second only to T5-CC at the token-level, possibly because it has not fully learned to restore the structure. In combination with Table 5, the illegal skeletons at the token-level are more than the other two, we infer that token-level datasets contain a large number of irregular instances, leading to inclined requirements on restoring structures through the syntax. However, the model cannot expertly handle the syntax via a limited dataset, so the advantages of fusion cannot be highlighted.

To further prove the above inference, we conduct pre-training experiments for S2RCC, the same as T5-CC [10]. The results are shown in Table 4.

Analyzing Table 4, first of all, pre-training can significantly improve S2RCC on all metrics in all levels. Pre-trained S2RCC is observed to be superior to T5-CC in perfect prediction, which is pre-trained likewise. It is worth noting that S2RCC also surpasses T5-CC in the token-level. It shows that it can learn enough syntax from the pre-training so that the role of structural information can be prominent. As for BLEU, despite slightly lower BLEU-2 or BLEU-3 in a few cases, pre-trained S2RCC exceeded T5-CC on the whole, especially in BLEU4. As longer matching sequences signify better results in code completion, it is reasonable that pre-trained S2RCC is superior to pre-trained T5-CC.

We also select qualitative examples to analyze the predictions of each model. We choose three qualitative examples sampled from construct, block, and token-level. Several baseline methods are picked as representatives, including two full models: ChatGPT [9] and Codex [8], which are only accessible through their online APIs. We ask S2RCC and baselines to complete the missing parts of these samples. ChatGPT is told the code segments and given the prompt: "fill in the code for the [M] section". The results are shown in Fig. 6.

Analyzing Fig. 6, other models with similar scales (smaller than 180M) cannot perfectly complete the three examples, while S2RCC can. The results of ChatGPT and Codex come from their full release (175B), but even then, ChatGPT still does not complete all examples right. Codex can complete them accurately based on its large number of parameters and vast code corpus. However, we do not believe it is a fair comparison to S2RCC: first, there is no evidence that its pre-training corpus does not incorporate these samples; second, it is difficult to guarantee the performances of such pre-training LM when with few parameters, as the effectiveness of small OPT – with a similar structure – is not significant. Overall, S2RCC is the only method in those with similar parameters to handle these three multi-token completion samples perfectly.

To summarize the experimental results of RQ1, S2RCC outperforms existing code completion methods and is more suitable for real-time response on low-resource IoT service scenarios. Compared to the expert model under the same constraints of low parameters, S2RCC has a

[7] For brevity, in this section, we refer to the S2RCC model simply as S2RCC, while the S2RCC framework is still referred to by its full name.

[8] Perfect of n-gram and T5-CC were obtained from other works, but they did not release CodeBLEU values or prediction results for the test set, so CodeBLEU is ignored.

higher CodeBLEU. Compared to larger models, the quality of S2RCC's results significantly surpasses the baseline, and it has a marked advantage in inference speed. Furthermore, S2RCC can benefit significantly from pre-training.

RQ2: What role does skeleton prediction play? Do all the components of S2RCC make sense?

We cannot simply remove one of the components of S2RCC to show their roles, as they are connected in order (e.g., the Structure Encoder cannot receive a valid structure after removing the Skeleton Decoder). But ablating both the Skeleton Decoder and the Structure Encoder has been proved inferior according to Table 2, where the model degenerates into Transformer.

A feasible way to further determine the role of the skeleton is to perform a statistical analysis. We calculated the following statistical properties of the S2RCC's predictions in it: Perfect Prediction and Perfect Skeleton (the proportion of instances whose whole span or whose skeleton are predicted perfectly, respectively), Illegal Skeleton (the ratio of skeleton predictions that lead to syntax errors), Self-repaired (the proportion of N_{S0-R1}, which is the number of instances with the wrong skeleton but perfect refinement) and Chi-square (the χ^2 of the independence test for perfect or not of skeleton and span, calculated by Eq. (6)). Table 5 shows these statistics of S2RCC's prediction.

$$\begin{aligned}&a = N_{S0-R0}, b = N_{S0-R1}\\&c = N_{S1-R0}, d = N_{S1-R1}\\&\chi^2 = \frac{\mathrm{N}(ad-bc)}{(a+b)(c+d)(a+c)(b+d)}\end{aligned} \tag{6}$$

From Table 5, the perfect skeleton positively correlates with perfect prediction. We then conduct Chi-square tests and get $p < .001$ at all levels, proving a significant relationship between perfect skeleton and perfect prediction. These examinations show that the skeleton is not independent but strongly associated with the result. On the other hand, the skeleton does not entirely determine the outcome. As we expect, a few instances repair themselves: the model finds an incongruous structure during the refinement and gives a correct prediction. Overall, the statistics prove that all the components indeed work as expected.

Substitution experiments can also reveal the significance of each component. It can be seen in Table 7 how CodeT5 and OPT perform as substitutions of Context Encoder and Skeleton Decoder.[9] For the Structure Encoder, we first evaluate the impact of the heterogeneous code graph and replace it with the construction from CCAG [7], called S2R-CCAG. Then we employ multiple GNNs: GGNN [24], GCN [40], GAT [41], and HGT [42] as the Structure Encoder. For the Refinement Decoder, which fuses the semantic and structural representation of the code, we propose a three-layer stacked attention for S2RCC. Graph-Sandwich and GREAT [38] embody a similar fusion idea, so we consider them as alternatives. We performed the above experiments and integrated the results into Table 6.

Analyzing Table 6, firstly, the homogeneous graph from CCAG is inferior to the heterogeneous code graph proposed by S2RCC. Among alternative GNNs, GGNN is the most suitable Structure Encoder for S2RCC. In the end, the ways of fusion from G-sandwich and GREAT are not as satisfactory as the three-layer stacked attention we designed. The results show that our S2RCC model is the most effective combination of the S2RCC framework.

To summarize the experimental results of RQ2, each component of S2RCC plays an indispensable role. Statistical analysis proves that the predicted skeleton is indeed strongly related to the final prediction. The substitution experiment proves that our proposed S2RCC model, with the Transformer, GGNN, heterogeneous code graph, and stacked attention, is the optimal solution among all candidate combinations.

[9] We do not consider them as alternatives of standard implementations of S2RCC for a fair comparison, as we are not sure that the training dataset of these pre-training LMs does not include instances in the code completion test set.

RQ3: Can the S2RCC framework be adapted to other pre-training models? How much potential it has?

To explore the adaptability of the S2RCC framework, we take it to wrap the CodeT5 [32] and OPT [34] to form S2R-CodeT5 and S2R-OPT, with fine-tuning to the code completion task. The encoder and decoder of CodeT5-small are the Context Encoder and Skeleton Decoder of S2R-CodeT5, respectively. OPT-125M simultaneously takes the role of Context Encoder and Skeleton Decoder of S2R-OPT, as it has relatively large parameters. In addition, we also present Transformer-172M as a comparison, served as an LM that has similar parameters with S2RCC (163M). These results are shown in Table 7.

Table 7 shows that after rebuilt by the S2RCC framework, the performances of S2R-CodeT5 (compared to CodeT5), S2R-OPT (compared to OPT), and S2RCC (compared to Transformer-86M) is improved. This proves the applicability of the S2RCC framework. It is worth noting that the performance of Transformer-172M is also inferior to S2RCC (163M), even though they have similar scales. This proves that taking the S2RCC framework to rebuild the LMs is more effective than just enlarging their scales. On the other hand, the S2R-CodeT5 even outperformed the standard S2RCC on some metrics, which means the S2RCC framework has more room for improvement by absorbing the latest pre-training LMs. Overall, the S2RCC framework has broad applicability and great potential to be extended further.

The potential of the S2RCC framework may also exist in its applications. External tools such as compilers or static analyzers can declare the code structure (e.g., brackets matching and the number of parameters in a function call). This suggests that S2RCC has the potential for further improvement by getting syntactic aid from the IDE. We evaluate it in the *micro change*, where the skeletons are directly informed. In this case, the perfect predictions reach 58.46%, 39.51%, and 68.79% in construct, block, and token-level, respectively. The results show that S2RCC has excellent potential for improvement in practice.

To summarize the experimental results of RQ3, S2RCC has enormous application potential. First, it can wrap any other language model to achieve improvements, thus benefiting from the development of language models. Secondly, under the condition of a known correct skeleton, S2RCC performs impressively, proving that it has the potential to interact with grammar analysis tools to gain benefits.

6. Discussion

There are some threats to the validity of S2RCC. The external threat is that we did not discuss model compression approaches. We mentioned that S2RCC can encapsulate small models to achieve effects comparable to large models. Some knowledge distillation and model pruning methods might also enable the transfer of effects from large to small models. However, we did not include them in the main text for two reasons: 1. Few studies discuss model compression strategies on code LLMs, and discussing general strategies is not in line with the focus of this paper. 2. To the best of our knowledge, model compression strategies can only roughly maintain the performance of LLMs but not achieve a reverse surpassing effect, whereas S2RCC can achieve this. The internal threat is that we have not tuned all the hyperparameters of all experiments. We implement a close configuration with T5-CC's for a fair comparison and cost saving. The threat in pre-training is that instances generated at each step are relatively random, causing inconsistent results.

There are also some practical advantages of S2RCC, which better support IDE applications. First, S2RCC supports *micro change* — in some cases when modifications are not followed by skeleton reconstruction. S2RCC can make full use of known skeletons, bringing improved results. Second, S2RCC can further adapt low-resource programming environment, keeping only Skeleton Prediction and leaving the restoration of the "hole" in the skeleton to the heuristic rules or developers' specifications. Finally, the problem of the unbearable time of inference

Table 7
Performances of LMs before and after wrapped in S2RCC framework.

		Method	Perfect	BLEU1	BLEU2	BLEU3	BLEU4		Method	Perfect	BLEU1	BLEU2	BLEU3	BLEU4
		CodeT5	39.7%	0.58	0.47	0.36	0.26		CodeT5	32.1%	0.54	0.45	0.31	0.27
		S2R-CodeT5	40.12%	0.59	0.45	0.37	0.28		S2R-CodeT5	32.3%	0.54	0.45	0.30	0.28
		OPT	32.0%	0.52	0.40	0.28	0.19		OPT	29.5%	0.53	0.44	0.31	0.28
Construct	Java	S2R-OPT	33.6%	0.52	0.41	0.27	0.21	Android	S2R-OPT	30.1%	0.52	0.44	0.32	0.28
		Trans-86M	36.6%	0.55	0.45	0.33	0.25		Trans-86M	27.6%	0.45	0.33	0.23	0.28
		Trans-172M	36.4%	0.55	0.43	0.31	0.25		Trans-172M	29.5%	0.52	0.43	0.30	0.27
		S2RCC	**51.6%**	**0.68**	**0.56**	**0.48**	**0.42**		S2RCC	**48.0%**	**0.67**	**0.57**	**0.49**	**0.45**
		CodeT5	16.2%	0.63	0.53	0.44	0.35		CodeT5	16.3%	0.61	0.51	0.41	0.32
		S2R-CodeT5	16.4%	0.63	0.52	0.44	0.35		S2R-CodeT5	16.4%	0.62	0.52	0.43	0.32
		OPT	15.3%	0.62	0.52	0.43	0.33		OPT	13.9%	0.59	0.48	0.39	0.28
Block	Java	S2R-OPT	21.4%	0.65	0.56	0.47	0.39	Android	S2R-OPT	19.3%	0.61	0.52	0.44	0.34
		Trans-86M	15.7%	0.56	0.45	0.33	0.25		Trans-86M	14.5%	0.59	0.48	0.39	0.29
		Trans-172M	15.6%	0.57	0.45	0.33	0.26		Trans-172M	14.5%	0.58	0.47	0.40	0.29
		S2RCC	**24.5%**	**0.67**	**0.59**	**0.51**	**0.43**		S2RCC	**23.3%**	**0.65**	**0.56**	**0.49**	**0.40**
		CodeT5	54.9%	0.80	0.68	0.51	0.36		CodeT5	58.1%	0.82	0.71	0.56	0.41
		S2R-CodeT5	**60.6%**	**0.83**	**0.73**	0.59	0.46		S2R-CodeT5	**63.9%**	**0.85**	**0.76**	**0.64**	**0.50**
		OPT	51.5%	0.77	0.64	0.46	0.31		OPT	52.9%	0.79	0.67	0.51	0.36
Token	Java	S2R-OPT	57.0%	0.81	0.69	0.55	0.41	Android	S2R-OPT	60.7%	0.83	0.73	0.59	0.45
		Trans-86M	55.9%	0.80	0.68	0.53	0.38		Trans-86M	55.8%	0.81	0.70	0.54	0.40
		Trans-172M	56.9%	0.81	0.69	0.52	0.38		Trans-172M	55.7%	0.82	0.70	0.53	0.40
		S2RCC	60.2%	0.82	0.72	**0.59**	**0.46**		S2RCC	62.8%	0.84	0.75	0.62	0.49

in auto-regressive language models [43] can be relieved. During inference, S2RCC can "buffer" completion for the developer: forwarding skeleton first, then code prediction. Due to the limitation of conditions, we did not make practical studies on these conjectures.

7. Conclusion

Integrating code completion techniques into Intelligent Device-free Sensing (IDFS) enables developers to efficiently develop algorithms for data processing and analysis, thereby enhancing the capabilities of intelligent sensing systems. We propose a two-stage, dual encoder–dual decoder, multi-token code completion framework: S2RCC. In the first phase, the model predicts only the skeleton, which contains only structure-sensitive tokens, and restores the code structure accordingly. The model fuses the structural and semantic representations in the second phase to give refined predictions. We propose an implementation and propose a heterogeneous code graph for Structure Encoder and use stacked attention to fuse structure and semantics for Refinement Decoder. The experimental results indicate our approach is better suited for low-resource, low-latency scenarios, such as IoT services. Constrained by low resources, our model outperforms the current best multi-token completion methods in both generation quality and speed. Our scalable framework can encapsulate smaller Language Models for enhanced performance.

CRediT authorship contribution statement

Yu Xia: Conceptualization, Methodology, Software, Writing – original draft. **Tian Liang:** Data Curation, Writing – review & editing. **WeiHuan Min:** Investigation, Visualization. **Li Kuang:** Writing – review & editing, Supervision, Project administration. **Honghao Gao:** Writing – review & editing.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Li Kuang reports financial support was provided by CCF-Tencent Open Fund.

Data availability

The data that has been used is confidential.

The artifacts of the S2RCC (Original data) (figshare)
.

Acknowledgments

This work was sponsored by CCF-Tencent Rhino-Bird Open Research Fund (No. RAGR20230109) and supported in part by the High Performance Computing Center of Central South University, PR China. We express our sincere gratitude for their financial support.

References

[1] S.S. Goel, A. Goel, M. Kumar, G. Moltó, A review of Internet of Things: qualifying technologies and boundless horizon, J. Reliab. Intell. Environ. (2021) http://dx.doi.org/10.1007/s40860-020-00127-w, URL: http://link.springer.com/10.1007/s40860-020-00127-w.

[2] S. Li, L.D. Xu, S. Zhao, The internet of things: a survey, Inf. Syst. Front. 17 (2) (2015) 243–259, http://dx.doi.org/10.1007/s10796-014-9492-7, URL: https://ideas.repec.org/a/spr/infosf/v17y2015i2d10.1007_s10796-014-9492-7.html.

[3] A. Svyatkovskiy, S. Lee, A. Hadjitofi, M. Riechert, J.V. Franco, M. Allamanis, Fast and memory-efficient neural code completion, in: 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR), 2021, pp. 329–340, http://dx.doi.org/10.1109/MSR52588.2021.00045.

[4] F. Liu, G. Li, Y. Zhao, Z. Jin, Multi-task learning based pre-trained language model for code completion, in: Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, ASE '20, Association for Computing Machinery, New York, NY, USA, 2021, pp. 473–485, http://dx.doi.org/10.1145/3324884.3416591.

[5] M. Izadi, R. Gismondi, G. Gousios, CodeFill: Multi-token code completion by jointly learning from structure and naming sequences, in: Proceedings of the 44th International Conference on Software Engineering, ICSE '22, Association for Computing Machinery, New York, NY, USA, 2022, pp. 401–412, http://dx.doi.org/10.1145/3510003.3510172.

[6] F. Liu, G. Li, B. Wei, X. Xia, Z. Fu, Z. Jin, 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.

[7] Y. Wang, H. Li, Code completion by modeling flattened abstract syntax trees as graphs, in: Proceedings of the AAAI Conference on Artificial Intelligence, 2021, pp. 14015–14023.

[8] M. Chen, J. Tworek, H. Jun, Q. Yuan, H.P.d.O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, et al., Evaluating large language models trained on code, 2021, arXiv preprint arXiv:2107.03374.

[9] J. Schulman, B. Zoph, C. Kim, ChatGPT, 2023, https://chat.openai.com/.

[10] M. Ciniselli, N. Cooper, L. Pascarella, A. Mastropaolo, E. Aghajani, D. Poshyvanyk, M. Di Penta, G. Bavota, An empirical study on the usage of transformer models for code completion, IEEE Trans. Softw. Eng. (2021).

[11] M. Ciniselli, N. Cooper, L. Pascarella, D. Poshyvanyk, M. Di Penta, G. Bavota, An empirical study on the usage of BERT models for code completion, in: 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR), IEEE, 2021, pp. 108–119.

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

[13] F. Liu, Z. Fu, G. Li, Z. Jin, H. Liu, Y. Hao, Non-autoregressive model for full-line code completion, 2022, arXiv preprint arXiv:2204.09877.

[14] D. Guo, A. Svyatkovskiy, J. Yin, N. Duan, M. Brockschmidt, M. Allamanis, Learning to complete code with sketches, in: International Conference on Learning Representations, 2021.

[15] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A.N. Gomez, Ł. Kaiser, I. Polosukhin, Attention is all you need, Adv. Neural Inf. Process. Syst. 30 (2017).

[16] J. Devlin, M.-W. Chang, K. Lee, K. Toutanova, BERT: Pre-training of deep bidirectional transformers for language understanding, 2018, http://dx.doi.org/10.48550/ARXIV.1810.04805, URL: https://arxiv.org/abs/1810.04805.

[17] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, V. Stoyanov, Roberta: A robustly optimized bert pretraining approach, 2019, arXiv preprint arXiv:1907.11692.

[18] D. Fried, A. Aghajanyan, J. Lin, S. Wang, E. Wallace, F. Shi, R. Zhong, W.-t. Yih, L. Zettlemoyer, M. Lewis, Incoder: A generative model for code infilling and synthesis, 2022, arXiv preprint arXiv:2204.05999.

[19] L.B. Allal, R. Li, D. Kocetkov, C. Mou, C. Akiki, C.M. Ferrandis, N. Muennighoff, M. Mishra, A. Gu, M. Dey, et al., SantaCoder: don't reach for the stars!, 2023, arXiv preprint arXiv:2301.03988.

[20] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, et al., Graphcodebert: Pre-training code representations with data flow, 2020, arXiv preprint arXiv:2009.08366.

[21] M. Allamanis, M. Brockschmidt, M. Khademi, Learning to represent programs with graphs, 2017, http://dx.doi.org/10.48550/ARXIV.1711.00740, URL: https://arxiv.org/abs/1711.00740.

[22] S. Kim, J. Zhao, Y. Tian, S. Chandra, Code prediction by feeding trees to transformers, in: Proceedings of the 43rd International Conference on Software Engineering, ICSE '21, IEEE Press, 2021, pp. 150–162, http://dx.doi.org/10.1109/ICSE43902.2021.00026.

[23] D. So, W. Mańke, H. Liu, Z. Dai, N. Shazeer, Q.V. Le, Searching for efficient transformers for language modeling, Adv. Neural Inf. Process. Syst. 34 (2021) 6010–6022.

[24] Y. Li, D. Tarlow, M. Brockschmidt, R. Zemel, Gated graph sequence neural networks, 2015, arXiv preprint arXiv:1511.05493.

[25] J. Li, Y. Wang, M.R. Lyu, I. King, Code completion with neural attention and pointer networks, 2017, arXiv preprint arXiv:1711.09573.

[26] A.A. Springborg, M.K. Andersen, K.H. Hattel, M. Albano, Cpp-tiny-client: A secure API client generator for IoT devices, in: Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing, SAC '22, Association for Computing Machinery, New York, NY, USA, 2022, pp. 202–205, http://dx.doi.org/10.1145/3477314.3508387.

[27] M. Sharaf, M. Abusair, R. Eleiwi, Y. Shana'a, I. Saleh, H. Muccini, Modeling and code generation framework for IoT, in: P. Fonseca i Casas, M.-R. Sancho, E. Sherratt (Eds.), System Analysis and Modeling. Languages, Methods, and Tools for Industry 4.0, Springer International Publishing, Cham, 2019, pp. 99–115.

[28] S. Lu, N. Duan, H. Han, D. Guo, S.-w. Hwang, A. Svyatkovskiy, ReACC: A retrieval-augmented code completion framework, 2022, arXiv preprint arXiv:2203.07722.

[29] J. Li, Y. Li, G. Li, Z. Jin, Y. Hao, X. Hu, Skcoder: A sketch-based approach for automatic code generation, 2023, arXiv preprint arXiv:2302.06144.

[30] F. Liu, G. Li, B. Wei, X. Xia, Z. Fu, Z. Jin, A unified multi-task learning model for AST-level and token-level code completion, Empir. Softw. Eng. 27 (4) (2022) 1–38.

[31] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, et al., Codebert: A pre-trained model for programming and natural languages, 2020, arXiv preprint arXiv:2002.08155.

[32] Y. Wang, W. Wang, S. Joty, S.C. Hoi, Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation, 2021, arXiv preprint arXiv:2109.00859.

[33] T. Brown, B. Mann, N. Ryder, M. Subbiah, J.D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, et al., Language models are few-shot learners, Adv. Neural Inf. Process. Syst. 33 (2020) 1877–1901.

[34] S. Zhang, S. Roller, N. Goyal, M. Artetxe, M. Chen, S. Chen, C. Dewan, M. Diab, X. Li, X.V. Lin, et al., Opt: Open pre-trained transformer language models, 2022, arXiv preprint arXiv:2205.01068.

[35] X. Ling, L. Wu, S. Wang, G. Pan, T. Ma, F. Xu, A.X. Liu, C. Wu, S. Ji, Deep graph matching and searching for semantic code retrieval, ACM Trans. Knowl. Discov. Data (TKDD) 15 (5) (2021) 1–21.

[36] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, M. Brockschmidt, CodeSearchNet challenge: Evaluating the state of semantic code search, 2019, arXiv preprint arXiv:1909.09436.

[37] V.J. Hellendoorn, P. Devanbu, Are deep neural networks the best choice for modeling source code? in: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, 2017, pp. 763–773.

[38] V.J. Hellendoorn, C. Sutton, R. Singh, P. Maniatis, D. Bieber, Global relational models of source code, in: International Conference on Learning Representations, 2019.

[39] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, P.J. Liu, et al., Exploring the limits of transfer learning with a unified text-to-text transformer, J. Mach. Learn. Res. 21 (140) (2020) 1–67.

[40] M. Schlichtkrull, T.N. Kipf, P. Bloem, R.v.d. Berg, I. Titov, M. Welling, Modeling relational data with graph convolutional networks, 2017, http://dx.doi.org/10.48550/ARXIV.1703.06103, URL: https://arxiv.org/abs/1703.06103.

[41] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, Y. Bengio, Graph attention networks, 2017, arXiv preprint arXiv:1710.10903.

[42] Z. Hu, Y. Dong, K. Wang, Y. Sun, Heterogeneous graph transformer, in: Proceedings of the Web Conference 2020, 2020, pp. 2704–2710.

[43] Z. Li, E. Wallace, S. Shen, K. Lin, K. Keutzer, D. Klein, J. Gonzalez, Train big, then compress: Rethinking model size for efficient training and inference of transformers, in: International Conference on Machine Learning, PMLR, 2020, pp. 5958–5968.