

# Efficient Cross-Modality Alignment for Enhanced Code Structure Understanding of Large Language Model

Ruijun Feng

Program Analysis Group

University of New South Wales

25/09/2024

# What is code language model

- Code language model (CLM) is an artificial neural network trained on **code data** to understand code and perform various **code-related tasks**, such as bug detection, code translation, etc.

# Code language model pre-training

- Previous pre-training methods have tried various **code graph information** to enhance code language model's understanding to code.

# Code language model pre-training

- Previous pre-training methods have tried various **graph structures** to enhance code language model's understanding to code.
- Data Flow Graph—GraphCodeBERT (ICLR 2021)

# Code language model pre-training

- Previous pre-training methods have tried various **graph structures** to enhance code language model's understanding to code.
- Data Flow Graph—GraphCodeBERT (ICLR 2021)
- Abstract Syntax Tree—AST-T5 (ICML 2024)

# Code language model pre-training

- Previous pre-training methods have tried various **graph structures** to enhance code language model's understanding to code.
- Data Flow Graph—GraphCodeBERT (ICLR 2021)
- Abstract Syntax Tree—AST-T5 (ICML 2024)
- Semantic Flow Graph—SemanticCodeBERT (FSE 2023)

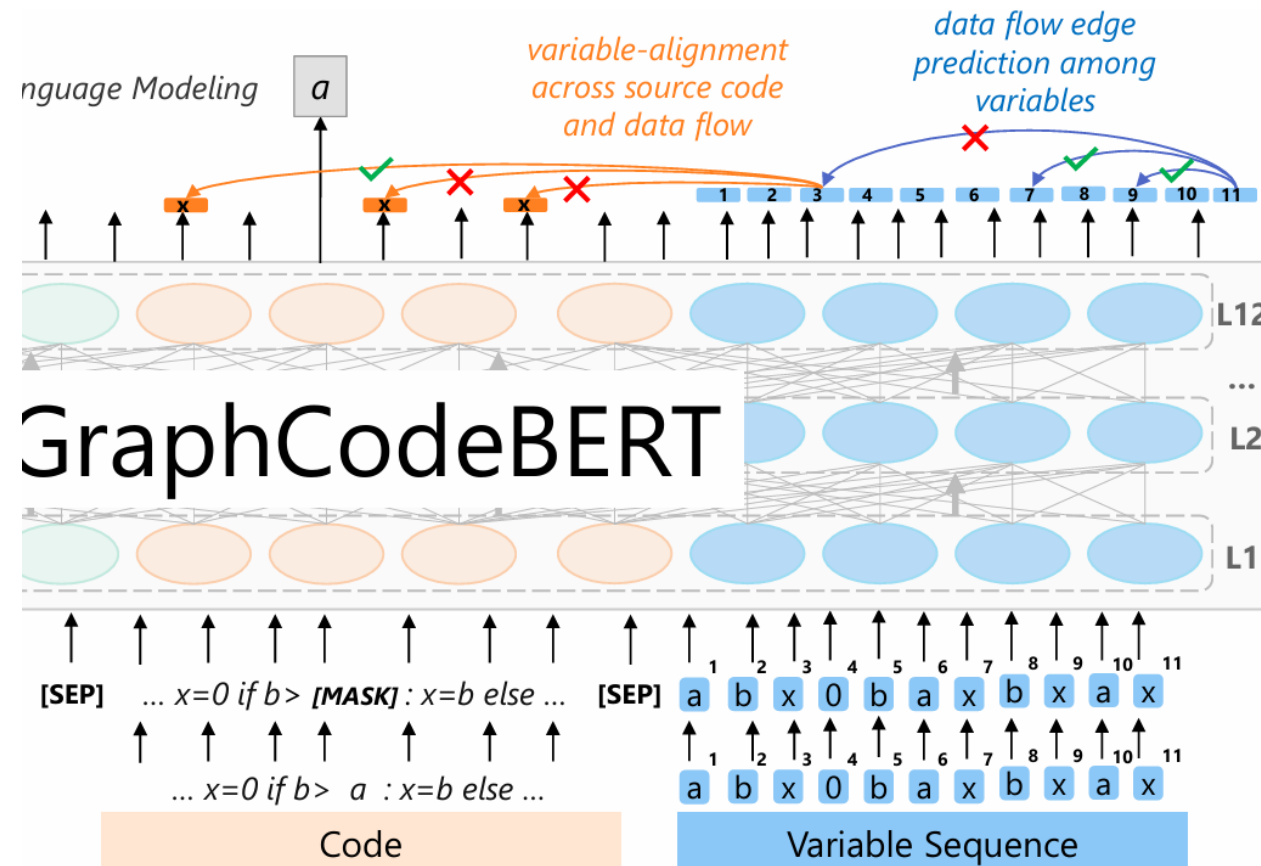
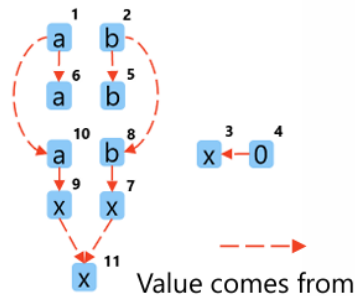
# How is code graph information used?

- Example: GraphCodeBERT

## Source code

```
def max(a,1b2):
3x=40
5if b6>a:
7  x=8b
else:
9  x=10a
return x11
```

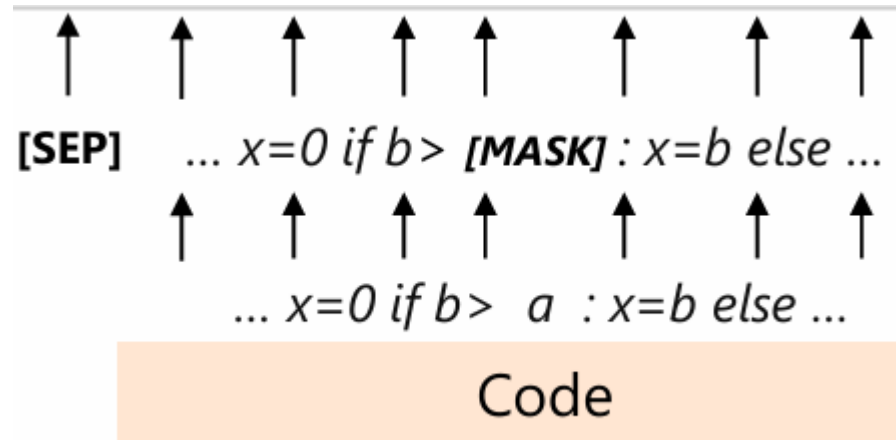
## Data Flow



# How is code graph information used?

## Source code

```
def max(a,1b2):  
  x3=04  
  if b5>a6:  
    x7=b8  
  else:  
    x9=a10  
  return x11
```





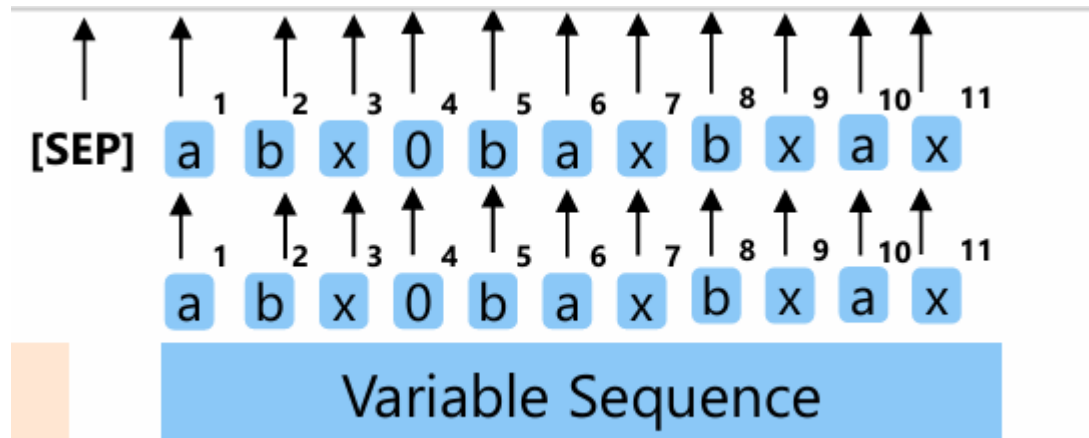
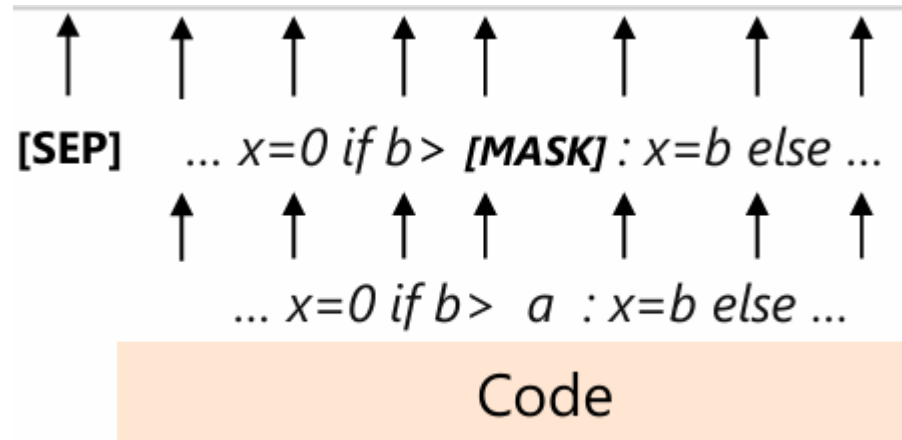
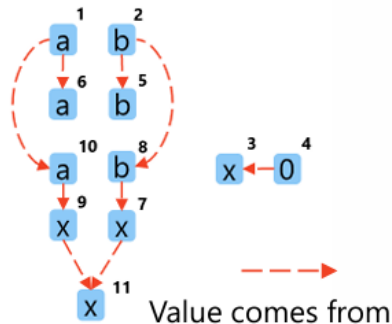
# How is code graph information used?

## Source code

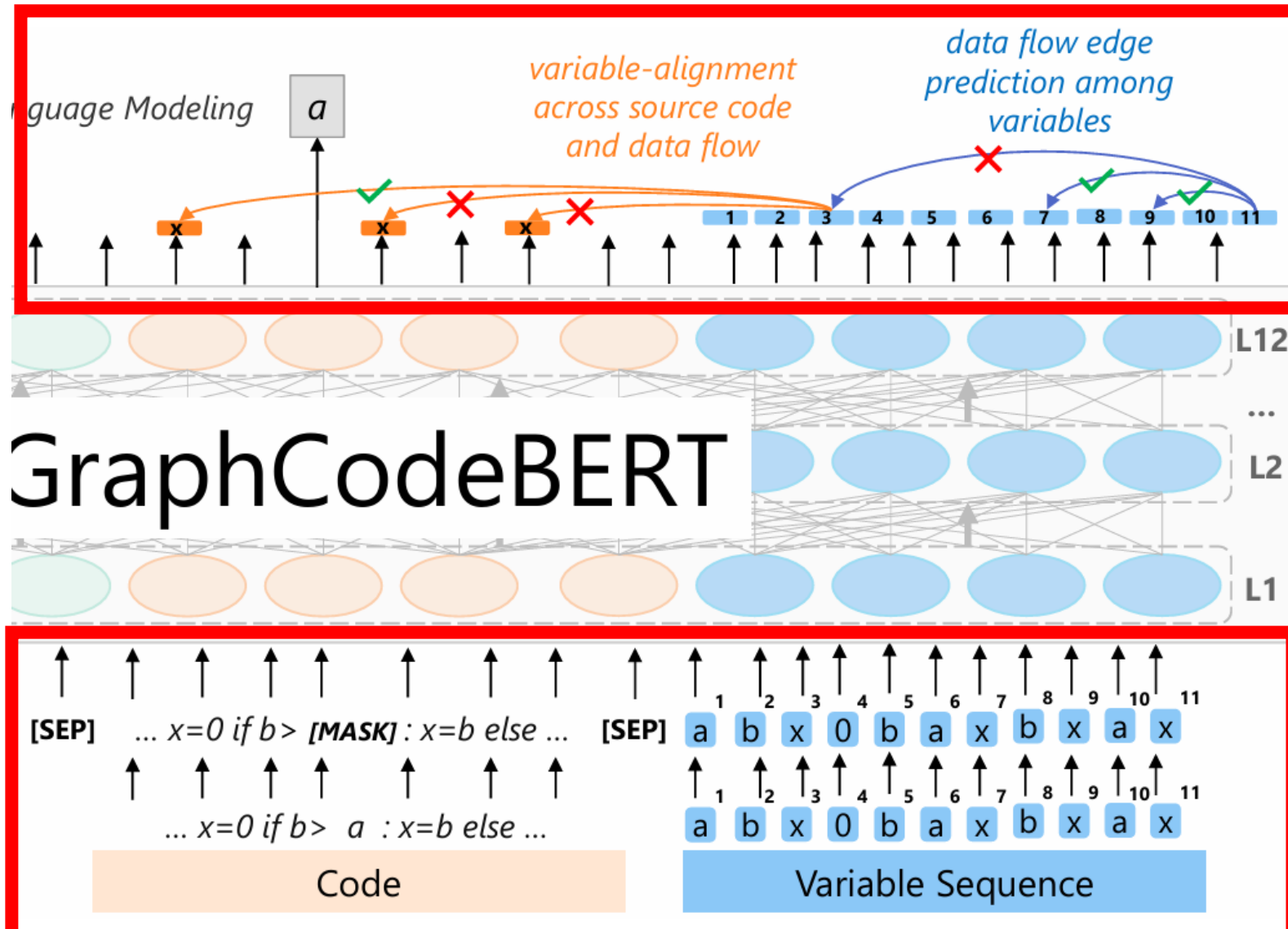
```
def max(a,1b2):
3x=04
5if b5>a6:
7    x7=b8
else:
9    x9=a10
return x11
```



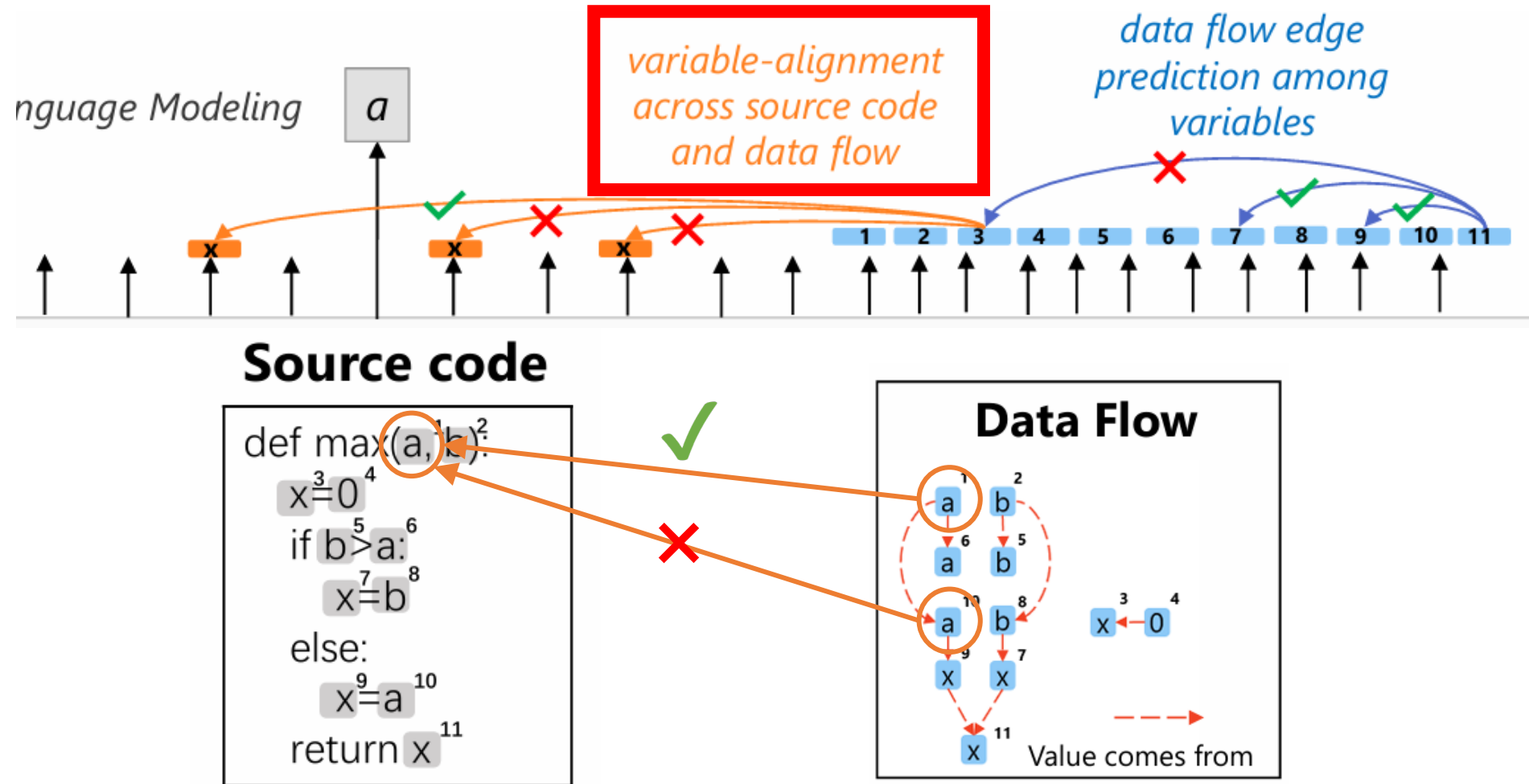
## Data Flow



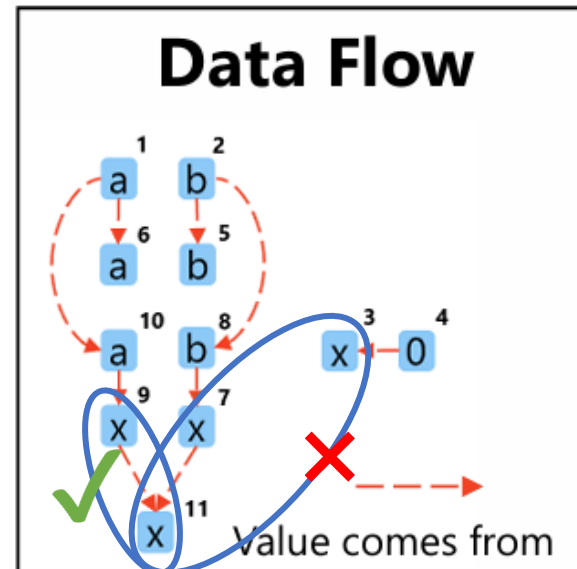
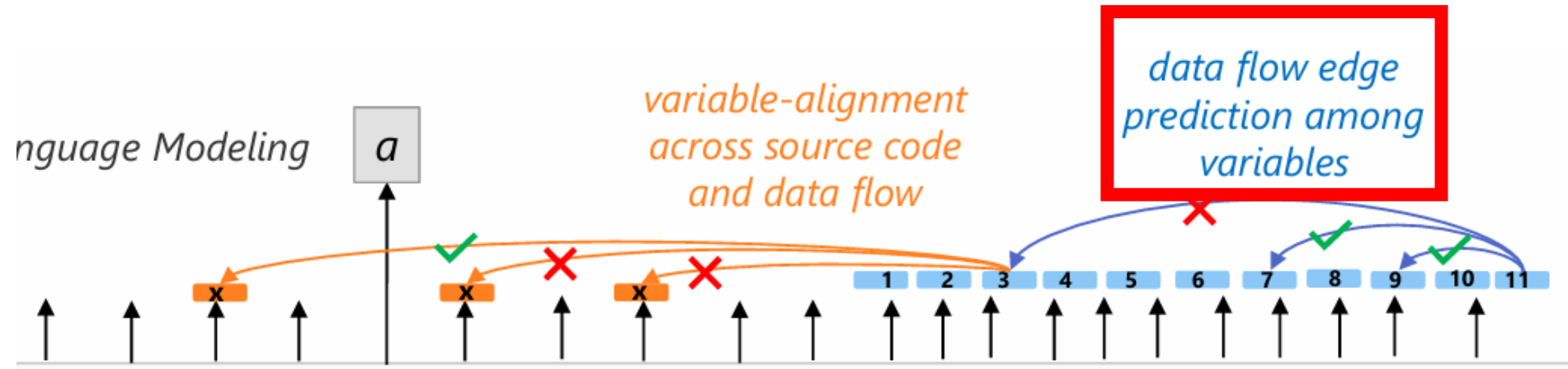
# How is code graph information used?



# How is code graph information used?

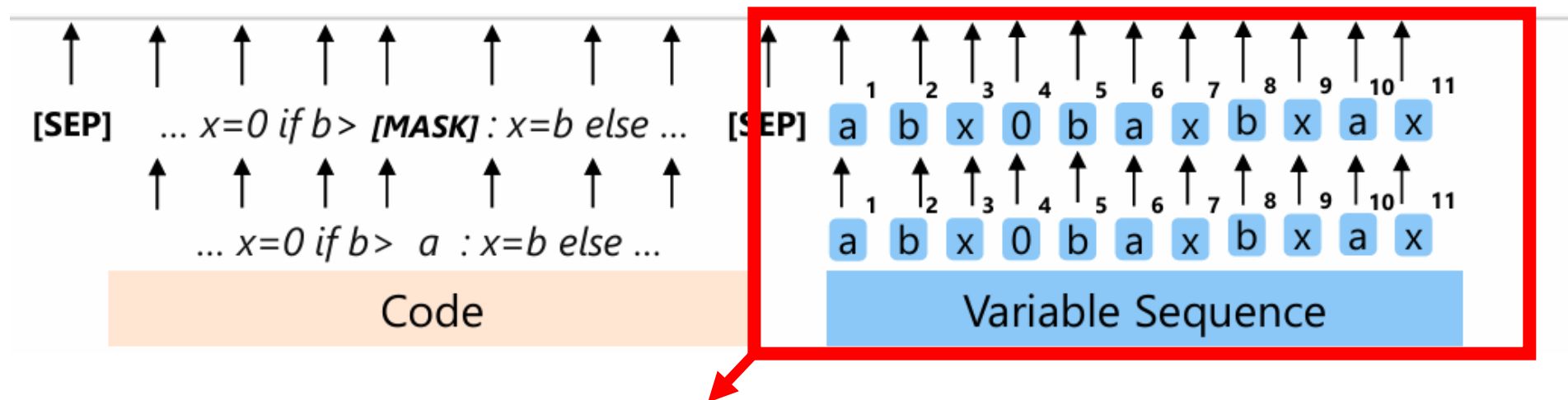


# How is code graph information used?



# Limitations of traditional CLM training

- Discrepancy in testing and inference: the code graph information is explicitly used during training, but in testing and inference, **only the source code** is provided.



No data flow is explicitly provided.

# Limitations of traditional CLM training

- Mismatch with the development of large language model (LLM) for code: mainstream LLMs for code, such as CodeLlama and StarCoder, are pre-trained on **source code only**.

# Limitations of traditional CLM training

- Mismatch with the development of large language model (LLM) for code: mainstream LLMs for code like CodeLlama and StarCoder are pre-trained on **source code only**.
- It is infeasible for most researchers to use code graph information to pre-train a LLM for code from scratch. For example, StarCoder was trained using 512 Nvidia **A100 GPUs** over a period of **24 days** on **1 trillion tokens**.

# Code graph information in LLM for code

- Two ways to incorporate code graph information into LLMs for code without pre-training:
  - In-context learning (ICL)—GRACE (JSS2024)
  - Parameter-efficient fine-tuning—GALLa (arXiv)

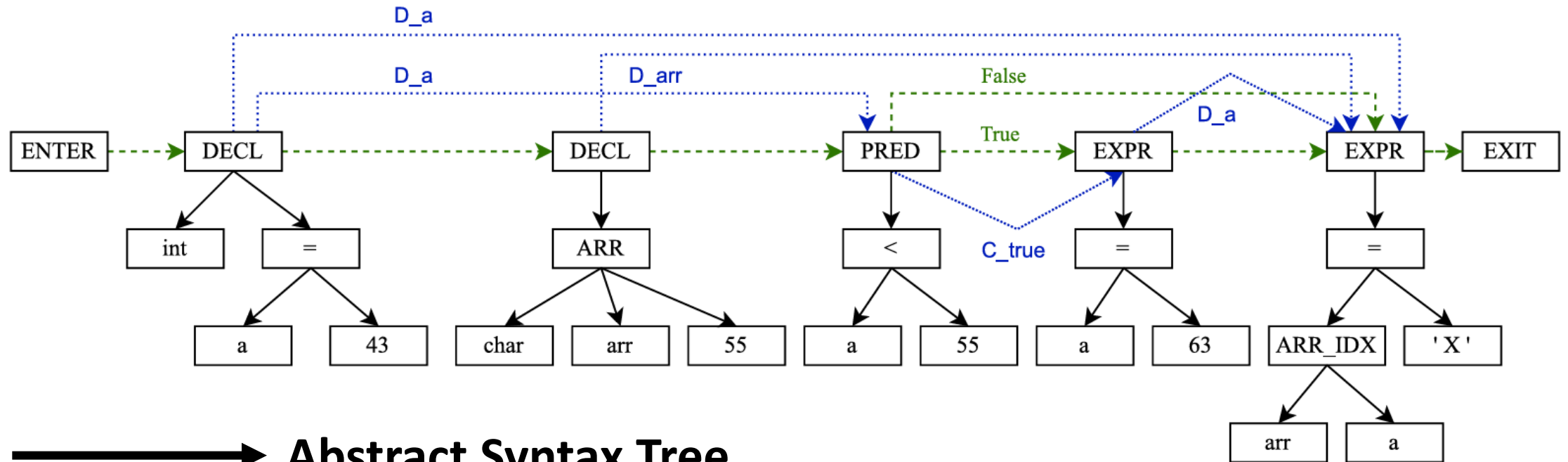


# In-context learning-based incorporation

- Example: GRACE.

```
01. void foo()  
02. {  
03.     int a = 43;  
04.     char arr[55];  
05.     if (a<55)  
06.     {  
07.         a=63;  
08.     }  
09.     arr[a] = 'X'  
10. }
```

# In-context learning-based incorporation



—————> **Abstract Syntax Tree**

—————> **Control Flow**

—————> **Data Flow**

# In-context learning-based incorporation

## NODES

Node	Type	Code
3	FunctionDef	foo()
5	IdentifierDeclStmt	int a=43
6	IdentifierDecl	a=43
7	IdentifierDeclType	int
...	...	...
17	IfStatement	if(a<55)
...	...	...
39	Symbol	a

## EDGES

Node1	Node2	EdgeType
6	7	IS_AST_PARENT
5	6	IS_AST_PARENT
...	...	...
36	5	FLOWS TO
5	18	...
18	39	USE
...	...	...
5	39	DEF

# In-context learning-based incorporation

#You are now an excellent programmer. You are conducting a function vulnerability detection task for C/C++ language.

#[code snippet]

#In the above code snippet, check for potential security vulnerabilities and output either 'Vulnerable' or 'Non-vulnerable'.

#The node information of the code snippet is as follows:


#[node information]


#The edge information of the code snippet is as follows:

#[edge information]

#The following is an example, and it is [lable].

#[demonstration]





Vulnerable

# Advantages of ICL-based incorporation

- The code graph information can be provided to the LLM when performing different code-related downstream tasks, ensuring **no discrepancy in testing and inference.**

# Limitations of ICL-based incorporation

- No explicit cross-modality alignment exists between the graph and the text. It heavily relies on the **general capabilities** of LLMs to **implicitly** align the code and the code graph by itself.

NODES

```
01. void foo()  
02. {  
03.     int a = 43;  
04.     char arr[55];  
05.     if (a<55)  
06.     {  
07.         a=63;  
08.     }  
09.     arr[a] = 'X'  
10. }
```

Node	Type	Code
3	FunctionDef	foo()
5	IdentifierDeclStmt	int a=43
6	IdentifierDecl	a=43
7	IdentifierDeclType	int
...	...	...
17	IfStatement	if(a<55)
...	...	...
39	Symbol	a

# Limitations of ICL-based incorporation

- Not scalable for large inputs.
  - LLM will **cache all the previous context** into a key-value cache for generation.

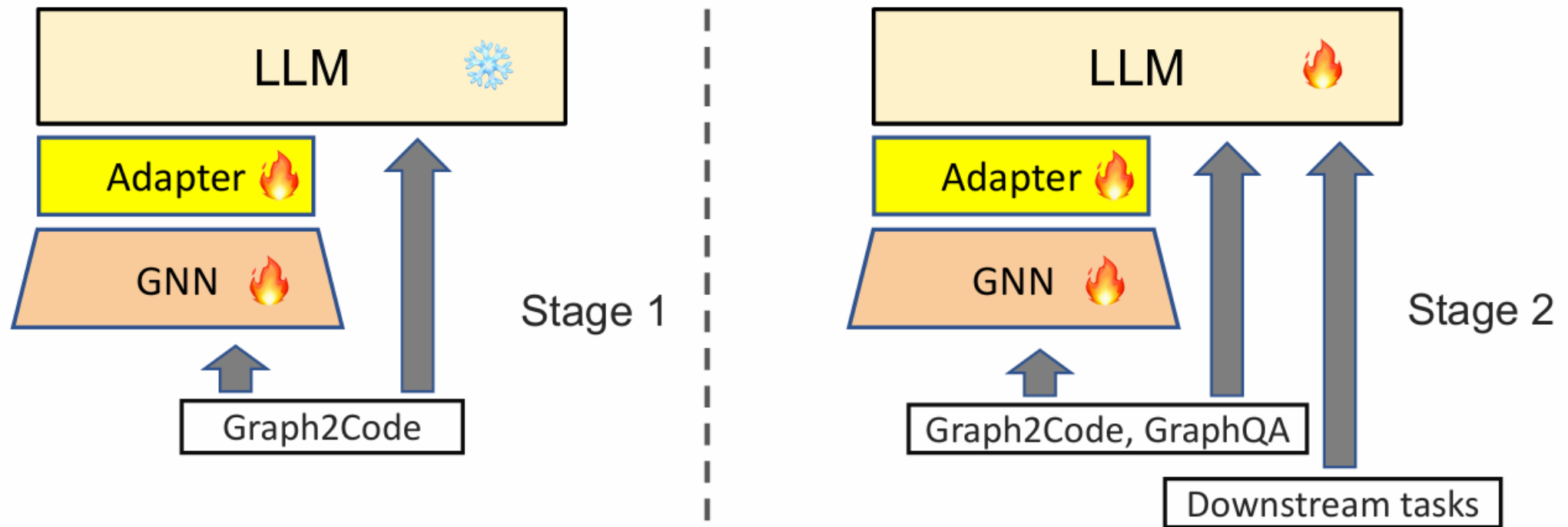
# Limitations of ICL-based incorporation

- Not scalable for large inputs.
  - LLM will **cache all the previous context** into a key-value cache for generation.
  - As the code graph grows, the length of the context used to describe it becomes proportional to the number of nodes ( $V$ ) and edges ( $E$ ). This will require more memory,  $O(V+E)$ , and may distract the model during generation.



# Tuning-based incorporation

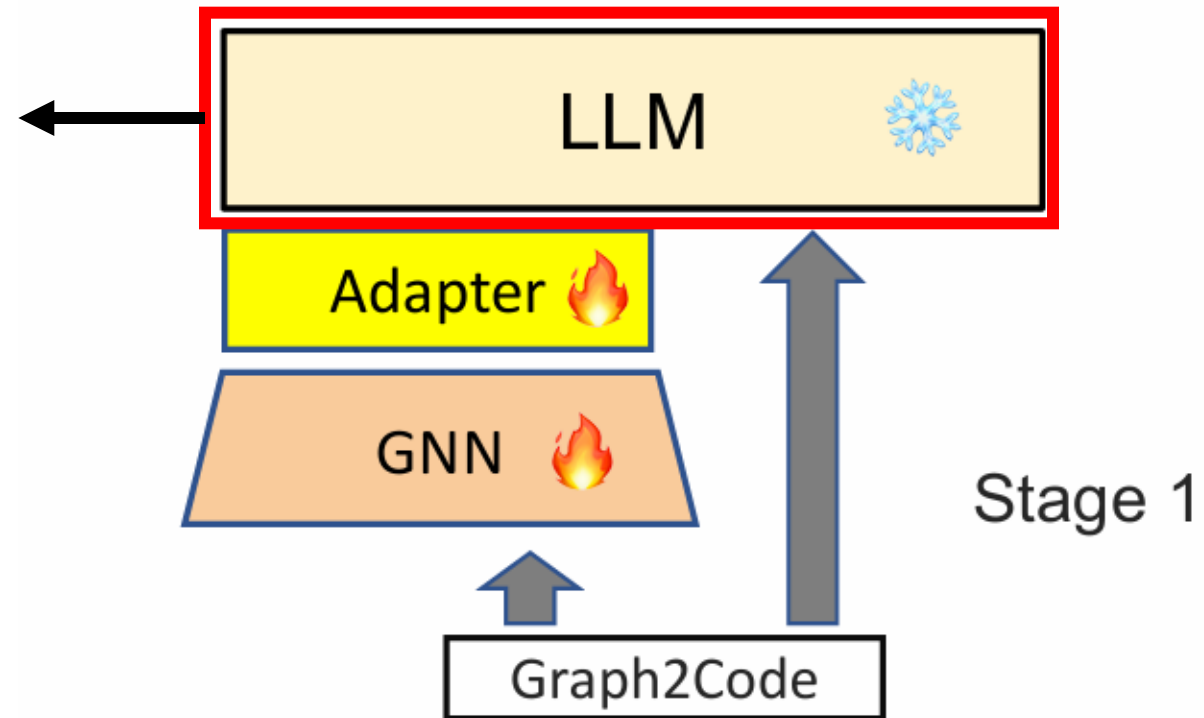
- Example: GALLa



# Tuning-based incorporation

- Stage 1: Pre-train a graph neural network

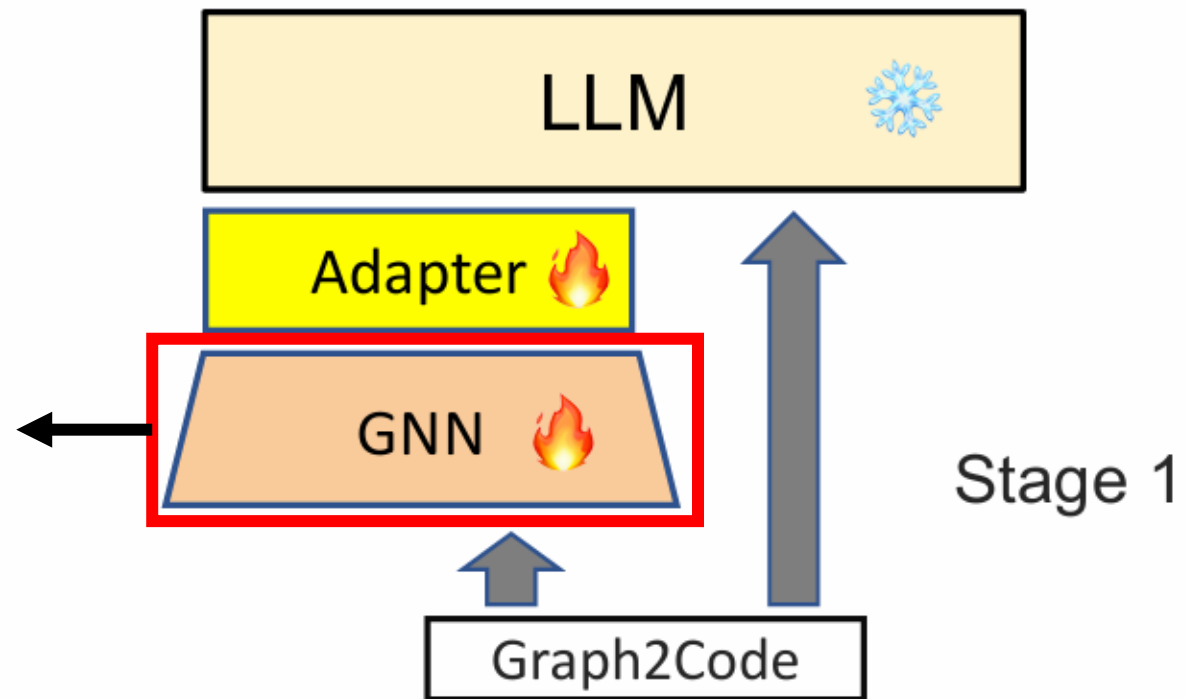
1. Froze the LLM



# Tuning-based incorporation

- Stage 1: Pre-train a graph neural network

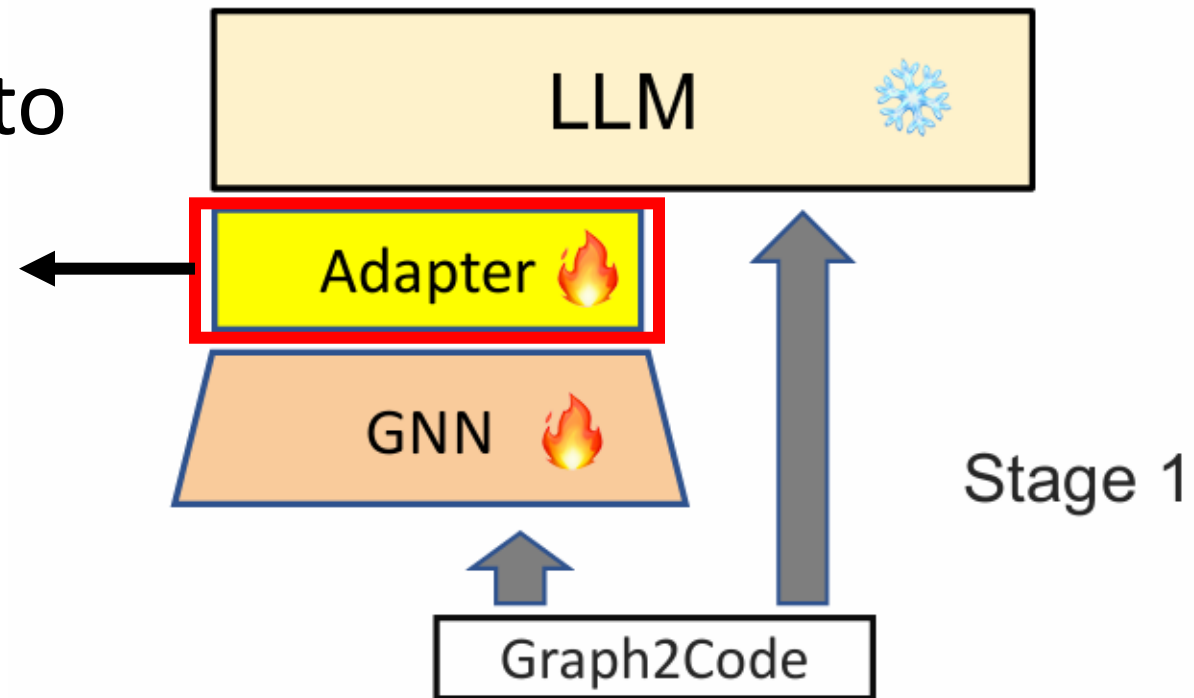
2. Encode graph into node embeddings



# Tuning-based incorporation

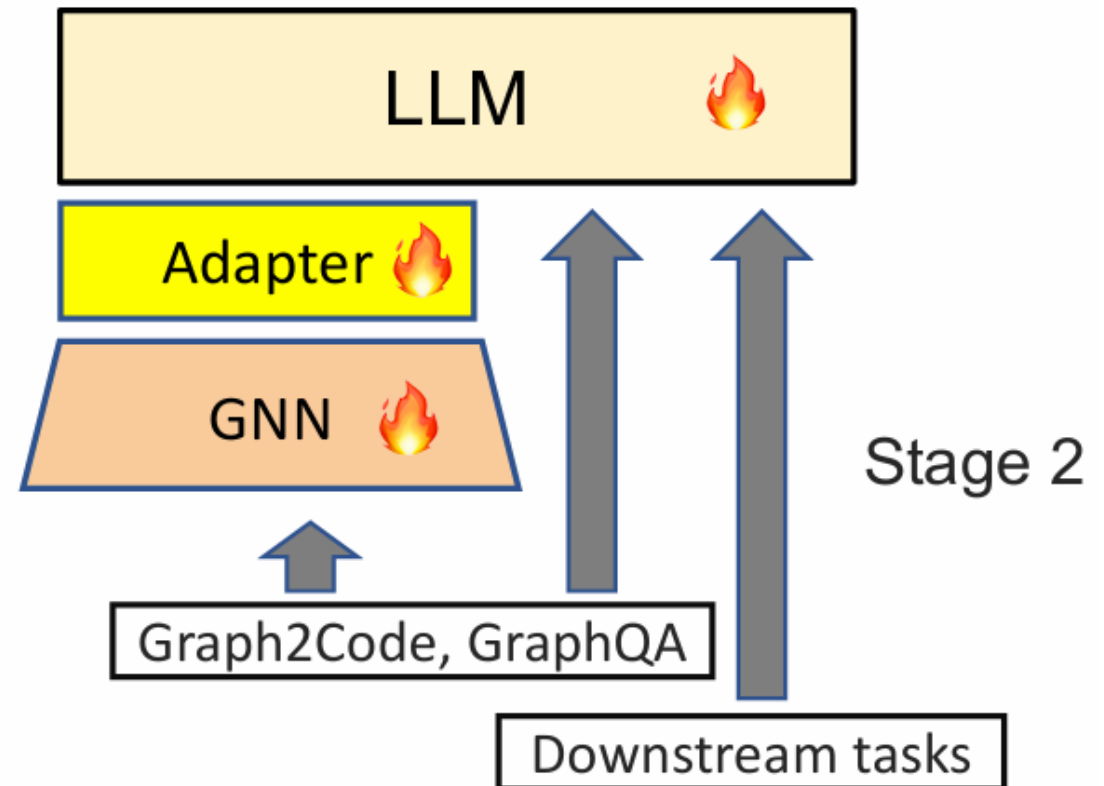
- Stage 1: Pre-train a graph neural network

3. Use learnable queries to interact with node embeddings and project into the latent space of LLM.



# Tuning-based incorporation

- Stage 2: Joint fine-tuning on code-related task



# Advantage of tuning-based incorporation

- More scalable to large input than ICL.

# Advantage of tuning-based incorporation

- More scalable to large input than ICL.
  - Graph neural network have encoded **edge information** into node embeddings, reducing the length of the context used to describe the code graph to  $O(V)$ .

# Limitation of tuning-based incorporation

- High computing resource requirements due to full parameter fine-tuning—**8 A100 GPUs**.



# Limitation of tuning-based incorporation

- High computing resource requirements due to full parameter fine-tuning—**8 A100 GPUs**.
- No **explicit cross-modality alignment** between graph and text. Their relationships are learnt via joint fine-tuning.

# Objectives

- **Explicitly** describes the cross-modality alignment between code graph and the textual information.
- Describe the code graph in a **memory efficient manner**.

# Overview

The proposed method consists of the following three key components:

- Graph attention network for extracting graph features.
- Cross-modality alignment module to explicitly align graph features with text features.
- Graph-enhanced soft prompt for fine-tuning.

# Preliminaries—text embeddings

Given a piece of code, it can be represented as text embedding vectors  $X \in R^{N \times d}$  through the dictionary and embedding layer of LLM.

- $N$  is the number of tokens.
- $d$  is the dimension of vectors.

# Preliminaries—code property graph

Code property graph is denoted as  $G = (V, E)$ , where  $V$  is the set of nodes, and  $E$  is the set of edges.

# Preliminaries—code property graph

Code property graph is denoted as  $G = (V, E)$ , where  $V$  is the set of nodes, and  $E$  is the set of edges.

- Each node contains multiple code statements.
- Each edge describes the relationships between src node and dst node (AST, CFG, or DFG).

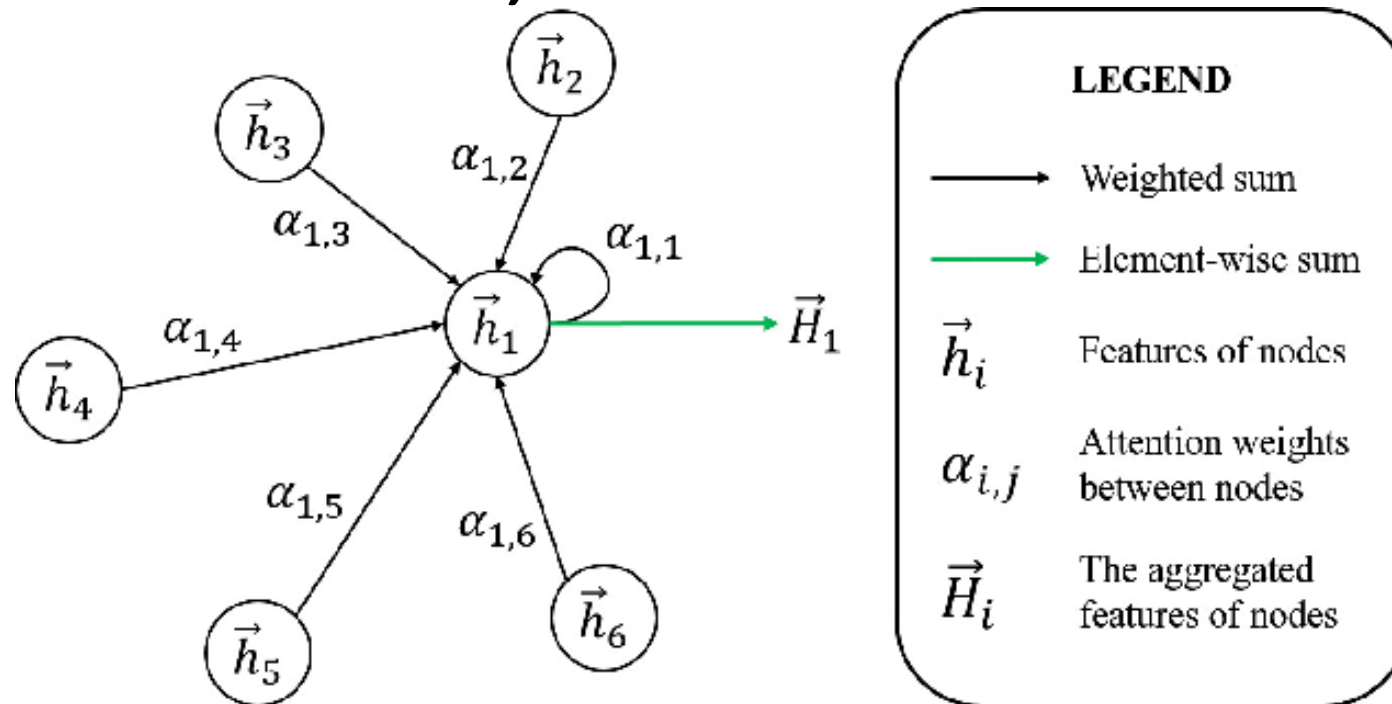
# Graph attention network

Node embeddings is initialized by aggregating the text embeddings of code statements, denoted as  $H \in R^{|\mathcal{V}| \times d}$ .

Edge is represented with an adjacency list  $R^{|\mathcal{E}| \times 2}$ , describes the src node and dst node of the edge.

# Graph attention network

Encode the edge information into node embeddings as **graph features** via graph attention network, denoted as  $Z \in R^{|V| \times d}$ .





# Cross-modality alignment

Given graph features  $Z \in R^{|V| \times d}$  and the input text embeddings of LLM  $X \in R^{N \times d}$ , compute the correlation scores  $A = ZX^T$ ,  $A \in R^{|V| \times N}$ .

	Token1	Token2	...	TokenN
Node1	0.812861	0.725886	...	0.056634
Node2	0.360199	0.040836	...	0.564214
...	...	...	...	...
Node  V	0.131826	0.578522	...	0.594849

# Cross-modality alignment

Perform softmax to correlation score matrix  $A$  to **quantitatively** describe how each node on the code graph is related to each token in the text.

	Token1	Token2	...	TokenN
Node1	0.419088	0.384178	...	0.196734
Node2	0.338647	0.246065	...	0.415288
...	...	...	...	...
Node $ V $	0.240847	0.376478	...	0.382675

# Cross-modality alignment

Use normalized correlation score matrix  $A$  to compute the weighted sum of relevant text embedding vectors of each node as the aligned graph feature matrix  $O=AX$ , where  $O \in \mathbb{R}^{|\mathcal{V}| \times d}$ .

# Cross-modality alignment

Each row of aligned graph feature matrix  $O \in \mathbb{R}^{|\mathcal{V}| \times d}$ , is a node feature vector that aligned to the text features.

	d1	d2	...	d
Node1	0.034109	0.81971	...	0.356225
Node2	0.66992	0.844896	...	0.290443
...	...	...	...	...
Node $ \mathcal{V} $	0.363469	0.747208	...	0.911386

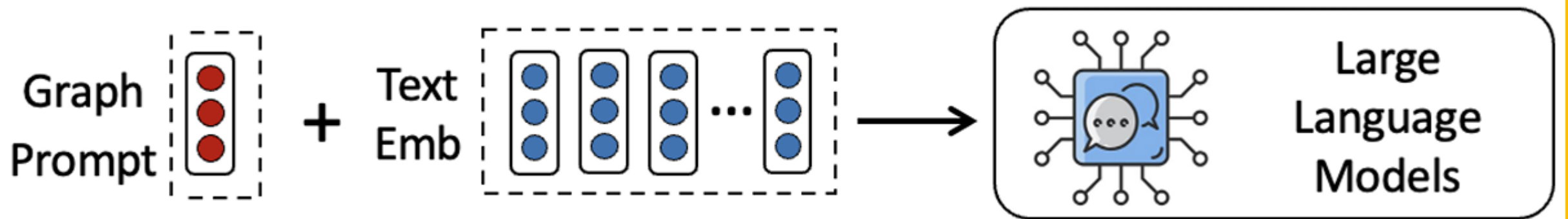
# Soft prompt tuning

Aggregate  $O \in R^{|\mathcal{V}| \times d}$  into a single soft prompt vector  $P \in R^{1 \times d}$  and concatenate it with original text embedding vectors  $X \in R^{N \times d}$ .

# Soft prompt tuning

Aggregate  $O \in R^{|\mathcal{V}| \times d}$  into a single soft prompt  $P \in R^{1 \times d}$  and concatenate it with original text embedding vectors  $X \in R^{N \times d}$ .

Feed  $P || X \in R^{(\mathbf{N}+1) \times d}$  to the LLM for soft prompt tuning on code-related tasks.



# Advantage of the proposed method

**Explicit cross-modality alignment** between code graph and text features by computing the correlation score matrix  $A \in R^{|\mathcal{V}| \times N}$ .

# Advantage of the proposed method

**Explicit cross-modality alignment** between code graph and text features by computing the correlation score matrix  $A \in R^{|\mathcal{V}| \times N}$ .

Scalable for large input. By aggregating  $O \in R^{|\mathcal{V}| \times d}$  into  $P \in R^{1 \times d}$ , the memory complexity of the graph context is always  **$O(1)$** .



# Experiment

**Dataset:** DiverseVul 33 million samples.

**Model:** CodeLlama (still tuning), Llama2, Gemma, CodeGemma (in progress).

# Research Question

**RQ1:** Is cross-modality alignment effective in improving the LLM for code ability on code-related task, like bug detection.

**RQ2:** Can cross-modality alignment enhance the code capabilities of a general-purpose LLM?  
(Llam2 and CodeLlama)

**RQ3:** Is cross-modality alignment less dependent on the LLM's comprehension ability? (7B, 13B)

# Thank you