



# Zero-shot Code Language Models

CSE 561A: Large Language Models

Shangyu Chen



1

# Introduction

- Zero-shot:
  - No training data for some tasks
- Zero-shot code filling:
  - Facing completely new code requirements

# Introduction

➔ GitHub Copilot

➔ AI Coding

```
42     sub r11, 1
43     jmp loop2
44     endloop2:
45
46     mov rsp, rbp
47     ret
48     */
49
50     #include <stdio.h>
51     #include <stdlib.h>
52     #include <string.h>
53
54     void print_array(char *array, int size) {
        for (int i = 0; i < size; i++) {
            printf("%d ", array[i]);
        }
        printf("\n");
    }
```

# Introduction

- How does a machine generate code?
- Program synthesis: Left → Right
- Program editing: Performed in the middle

5

# InCoder: A Generative Model for Code Infilling and Synthesis

CSE 561A: Large Language Models

Shangyu Chen



# Masking Model

Sentinel Token

Books and friends [MASK:0] be few but good. [MASK:0]would[EOM]

## Casual Masking

- Casual models: Automatically regress
- Masking method: Limited content

## Combine together

**Inference API**

Fill-Mask Examples

Mask token: [MASK]

Books and friends [MASK] be few but good.

Compute

Computation time on cpu: 0.243 s

would	0.477
will	0.112
might	0.104
can	0.093
could	0.092

</> JSON Output Maximize

- Autoregressive: Generated from left to right, ignoring scenarios where code is filled in the middle.
- Masking method: Great for filling in the middle.
- We combine instead of using the mask rather than autoregressive.

# Token Segmentation

- Masked TOKEN in natural language inference:
  - A single word / phrase
- Masked TOKEN in coding:
  - Code fragments
- How to create token? BPE tokenizer
  - Common code idioms are counted as a single token
  - Reduce the number of tokens

- Mask the fragment, not randomly masking substring
- Providing better training data, better suitable for inference.

# Masking Model -- Training

- ➔ How to train the model?
- ➔ Random masking

## Training Creating spans

Original Document

```
def count_words(filename: str) -> Dict[str, int]:  
    """Count the number of occurrences of each word in the file."""  
    with open(filename, 'r') as f:  
        word_counts = {}  
        for line in f:  
            for word in line.split():  
                if word in word_counts:  
                    word_counts[word] += 1  
                else:  
                    word_counts[word] = 1  
    return word_counts
```

Masked Document

```
def count_words(filename: str) -> Dict[str, int]:  
    """Count the number of occurrences of each word in the file."""  
    with open(filename, 'r') as f:  
        <MASK:0> in word_counts:  
            word_counts[word] += 1  
        else:  
            word_counts[word] = 1  
    return word_counts  
<MASK:0> word_counts = {}  
    for line in f:  
        for word in line.split():  
            if word <EOM>
```

Count of “span”s: Manually construct the long-tailed distribution  
Poisson distribution, but truncated to 256.



# Masking Model -- Training

- Some notations **Randomly select fragments to mask**
  - Text:  $D \rightarrow$  multiple splitted token:  $D_0, D_1, D_2, \dots$
  - Part of tokens:  $D_{i:j}$
  - Mask part of fragments ( $i:j$ ):
    - Left context:  $D_{0:i}$
    - Right context:  $D_{j:N}$
- When training, we maximize the probability of the existing data:
  - Loss: Cross entropy error (except mask tokens)  
$$\log P([\text{Left}; \langle \text{Mask}:\emptyset \rangle; \text{Right}; \langle \text{Mask}:\emptyset \rangle; \text{Span}; \langle \text{EOM} \rangle])$$

# Masking Model -- Inference

```
def func(i: int, j: int) |> int:  
    return i * 2 + j * 4
```

D: The full document,  
the returned type part is masked.

➤ Mask: code to be filled

➤ Generate context:

$P(\cdot \mid [\text{Left}; \langle \text{Mask:0} \rangle; \text{Right}; \langle \text{Mask:0} \rangle])$

➤ Stop Condition:

➤  $\langle \text{EOM} \rangle$  reached

➤ Maximum tokens, etc.

# Masking Model -- Inference

```
def func(i: int, j: int) |> int:
  return i * 2 + j * 4
```

D: The full document,  
the returned type part is masked.

➤ Mask: code to be filled

➤ Generate context:

$$P(\cdot \mid [\text{Left}; \langle \text{Mask:0} \rangle; \text{Right}; \langle \text{Mask:0} \rangle])$$

➤ Insert a  $\langle \text{Mask:1} \rangle$ ,  
indicating that there will  
still be something after the  
context. [1]

$$P(\cdot \mid [\text{Left}; \langle \text{Mask:0} \rangle; \text{Right}; \langle \text{Mask:1} \rangle; \langle \text{Mask:0} \rangle])$$

# Experiments

- **Primary Model:** InCoder-6.7B (base: 6.7B)
- **Dataset:** GitHub, GitLab, StackOverflow
- **Compare w/ 2 inference methods:**
  - Casual Infilling  $P(\cdot \mid [\text{Left}; \langle \text{Mask:0} \rangle; \text{Right}; \langle \text{Mask:0} \rangle])$
  - Left-to-right Single  $P(\cdot \mid \text{Left})$
  - Left-to-right Reranking  $\log P([\text{Left}; \text{Span}_k; \text{Right}])$

# Experiments

- Categories of Tasks:
  - Infilling lines of code
  - Docstring generation
  - Return type prediction
  - Variable name prediction
- **Test datasets:** HumanEval & CodeXGlue

# Experiment 1: Infilling lines of code

- Generating code
- Categories:
  - Single-line Infilling
  - Multi-line Infilling
- Evaluation:
  - Passing rate: same in - out
  - Exact matching: masks = original code

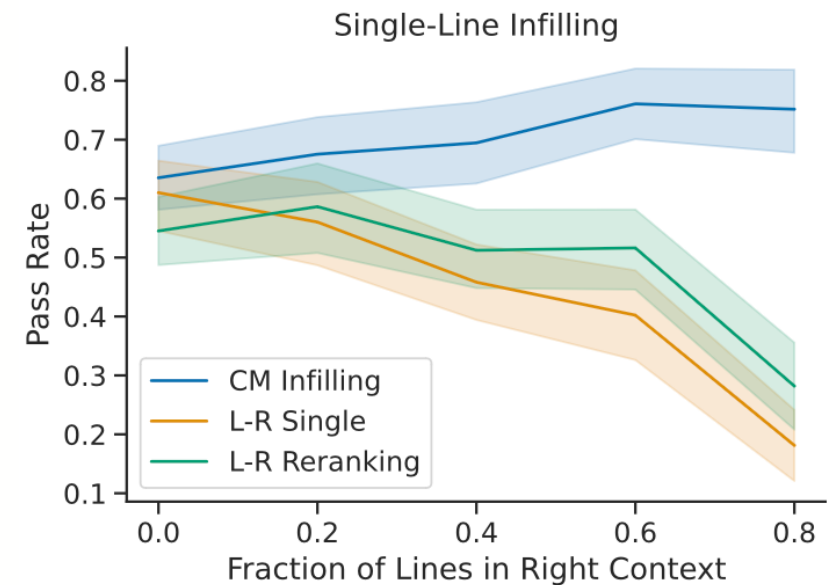
```
1  func test(a: int, b: int) -> int {  
2      // Multiply a and b  
3      return a * b  
4  }
```



# Experiment 1: Infilling lines of code

- Results: Single line
- Overall Performance
  - CM is better
- Performance by the position of the mask
  - CM shows less influence

Method	Pass Rate	Exact Match
L-R single	48.2	38.7
L-R reranking	54.9	44.1
CM infilling	69.0	56.3



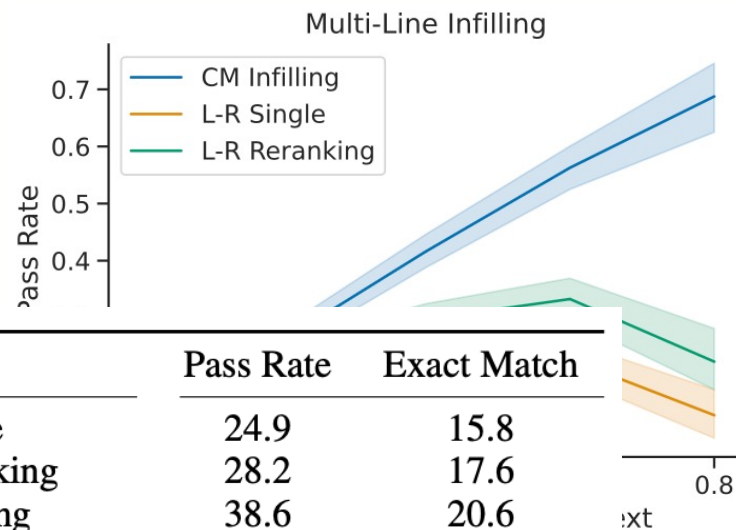
# Experiment 1: Infilling lines of code

- Results: Multiple line
- Overall: CM still better

Method	Pass Rate	Exact Match
L-R single	48.2	38.7
L-R reranking	54.9	44.1
CM infilling	69.0	56.3
PLBART	41.6	—
code-cushman-001	53.1	42.0
code-davinci-001	63.0	56.0

(a) Single-line infilling.

Method	Pass Rate	Exact Match
L-R single	24.9	15.8
L-R reranking	28.2	17.6
CM infilling	38.6	20.6



Method	Pass Rate	Exact Match
L-R single	24.9	15.8
L-R reranking	28.2	17.6
CM infilling	38.6	20.6
PLBART	13.1	—
code-cushman-001	30.8	17.4
code-davinci-001	37.8	19.8

(b) Multi-line infilling.

# Experiment 2: Docstring Generation

- ➔ Summarizing code snippet
- ➔ **Evaluation:** BLEU scores
- ➔ **Result:**
  - ➔ CM better than LR
  - ➔ Existing finetuned model have better performance

```
1 // test is a function that takes two integers, a and
  // b, and returns the result of raising a to the power of
  // b.
2 func test(a: int, b: int) -> int {
3     //
4     return a ** b
5 }
```

Method	BLEU
Ours: L-R single	16.05
Ours: L-R reranking	17.14
Ours: Causal-masked infilling	18.27
RoBERTa (Finetuned)	18.14
CodeBERT (Finetuned)	19.06
PLBART (Finetuned)	19.30
CodeT5 (Finetuned)	20.36

## Experiment 3: Return type prediction

- Predict the possible type returned in the function or some variable prediction
- **Additional Dataset:** Typewriter OSS
- **Evaluation:** If the return type is correct
  - Using AST (syntax tree algorithm)

```
def func(i: int, j: int) |> int:  
    return i * 2 + j * 4
```

# Experiments – Return type prediction

► **Performance:** CM infilling still better.

Method	Accuracy
Left-to-right single	12.0
Left-to-right reranking	12.4
Causal-masked infilling	<b>58.1</b>
↑CodeXGlue	Typewriter→

Method	Precision	Recall	F1
Ours: Left-to-right single	30.8	30.8	30.8
Ours: Left-to-right reranking	33.3	33.3	33.3
Ours: Causal-masked infilling	<b>59.2</b>	<b>59.2</b>	<b>59.2</b>
TypeWriter (Supervised)	54.9	43.2	48.3

# Experiments – Variable name prediction

- ➔ Predict what the variable represents according to the context
- ➔ Method
  - ➔ Mask all the variables in this task using AST
  - ➔ Predict according to the code / fragment

## Variable Name Prediction

```
def count_words(filename: str) -> Dict[str, int]:  
    """Count the number of occurrences of each word in the file."""  
    with open(filename, 'r') as f:  
        word_count = {}  
        for line in f:  
            for word in line.split():  
                if word in word_count:  
                    word_count[word] += 1  
                else:  
                    word_count[word] = 1  
    return word_count
```



## Experiments – Variable name prediction

- **Performance:** CM made full use of the right context -- important in this task

Method	Accuracy
Left-to-right single	18.4
Left-to-right reranking	23.5
Causal-masked infilling	30.6

# Ablation Experiments

## ► Prove:

- It is casual masking (CM), rather than language modeling (LM), makes the model better!

## ► Comparison:

- 1.3B parameter model
- Using CM and LM, comparing the passing rate.

# Ablation Experiments

## ► Results:

► CM shows significantly better performance

- For certain languages, using a single language as the training set may be better
- For languages w/ more constraints (Java...), the training data required is generally less than that with fewer constraints (Python...).

#	Size (B)	Obj.	Training Data	Data Size	Train Tokens	Train Compute	HumanEval Pass@1	MBPP Pass@1
1)	6.7	CM	multi lang + SO	204 GB	52 B	3.0 Z	15	19.4
2)	1.3	CM	multi lang + SO	204 GB	52 B	0.6 Z	8	10.9
3)	1.3	LM	multi lang + SO	204 GB	52 B	0.6 Z	6	8.9
4)	1.3	LM	Python + SO	104 GB	25 B	0.3 Z	9	9.8
5)	1.3	LM	Python	49 GB	11 B	0.1 Z	5	6.1
6)	2.3	LM	multi lang + SO	204 GB	52 B	1.1 Z	9	12.7

# Conclusion

- New Method in InCoder: Casual Masking
- Better performance for right context identifying
- Established a basis for future research on supervised infilling and editing

Any Questions?

# Code Llama: Open Foundation Models for Code

CSE 561A: Large Language Models

Shangyu Chen

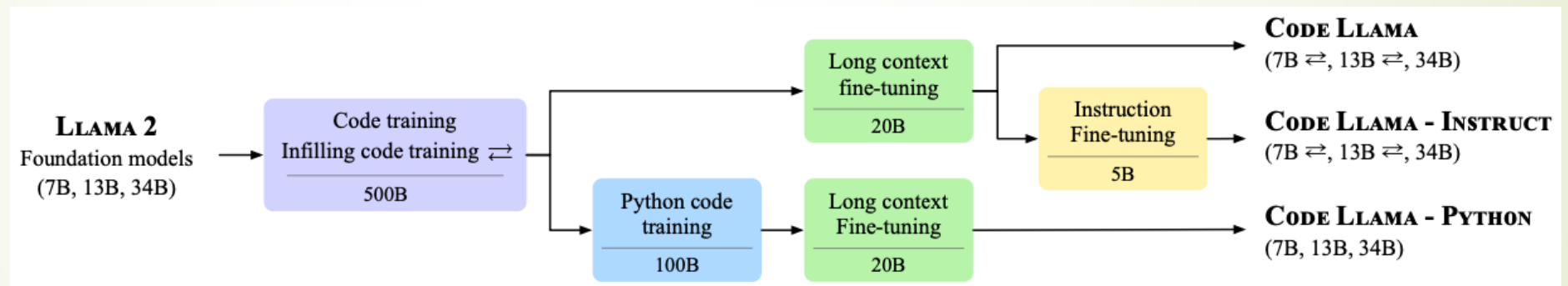


# Introduction

- Llama 2
  - a large language model developed by Meta
- Code Llama
  - program synthesis – just like the InCoder before
- Same concepts in program synthesis as InCoder
  - Infilling (masking)
  - Long input contexts
  - Fine-tuning

# Model Mechanics

## ► Training pipeline example



## ► Differences? Let's start with the model

# Model Mechanics

## ➤ Dataset: Llama 2 Sample

➤ With code and natural language related to code

Dataset	Sampling prop.	Epochs	Disk size
<b>Code Llama (500B tokens)</b>			
Code	85%	2.03	859 GB
Natural language related to code	8%	1.39	78 GB
Natural language	7%	0.01	3.5 TB
<b>Code Llama - Python (additional 100B tokens)</b>			
Python	75%	3.69	79 GB
Code	10%	0.05	859 GB
Natural language related to code	10%	0.35	78 GB
Natural language	5%	0.00	3.5 TB

# Model Mechanics

- Training infilling models
  - Causal masking, but slightly different
- Split:
  - Prefix, middle and suffix
  - The split positions of the tokens are uniformly distributed
- Tokenizer: PSM and SPM
  - In PSM, token is likely to be splitted into subtokens

# Model Mechanics

## ► Fine-tuning

### ► Long context fine-tuning: support more tokens

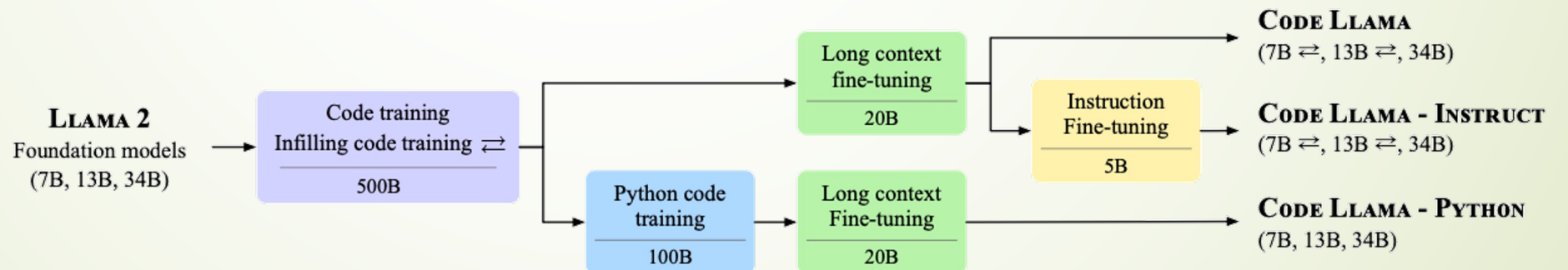
► Principle: Rotary Position Encoding

► position interpolation -> linear transformation

### ► Instruction fine-tuning

► Add realistic natural language problems

► Using Llama 2 and Code Llama to solve them



# Performance Results

- **Test Datasets:** HumanEval, MBPP, etc.
- **Multiple Languages**
- **Results focusing**
  - Impact of Llama 2 and Llama 2 -> Code Llama (and its specialization models)
  - Impact of fine-tuning



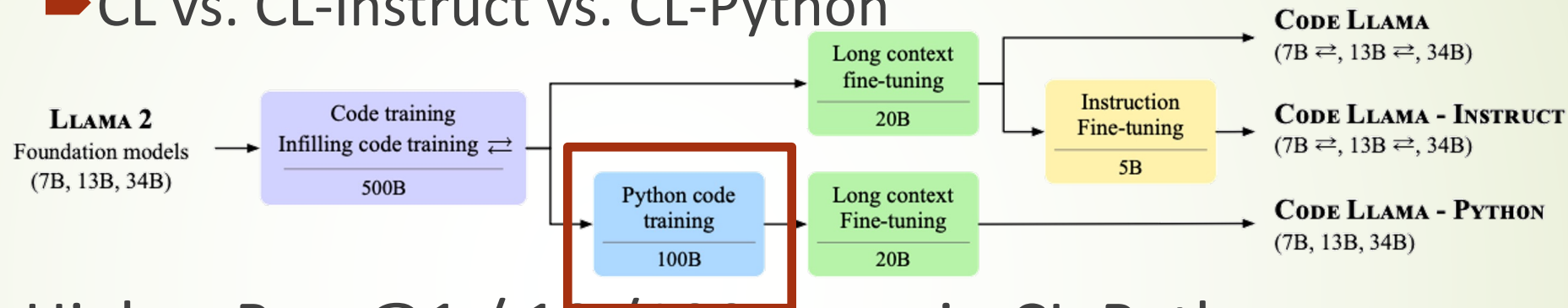
# Performance Results: Specialization

- ▶ Llama 2 vs. Code Llama
  - ▶ **Llama 2**: 2 trillion tokens
  - ▶ **Code Llama**: 500B extra tokens
  - ▶ Llama 2 70B  $\leftrightarrow$  Code Llama 7B

# Performance Results: Specialization

## General Model vs. specialized model

### CL vs. CL-Instruct vs. CL-Python



## Higher Pass@1 / 10 / 100 score in CL-Python

### Analog InCoder: w/ StackOverflow, multi languages

CODE LLAMA	7B	33.5%	59.6%	85.9%	41.4%	66.7%	82.5%
	13B	36.0%	69.4%	89.8%	47.0%	71.7%	87.1%
	34B	48.8%	76.8%	93.0%	55.0%	76.2%	86.6%
CODE LLAMA - INSTRUCT	7B	34.8%	64.3%	88.1%	44.4%	65.4%	76.8%
	13B	42.7%	71.6%	91.6%	49.4%	71.2%	84.1%
	34B	41.5%	77.2%	93.5%	57.0%	74.6%	85.4%
UNNATURAL CODE LLAMA	34B	62.2%	85.2%	95.4%	61.2%	76.6%	86.7%
CODE LLAMA - PYTHON	7B	38.4%	70.3%	90.6%	47.6%	70.3%	84.8%
	13B	43.3%	77.4%	94.1%	49.0%	74.0%	87.6%
	34B	53.7%	82.8%	94.7%	56.2%	76.4%	88.2%

# Performance Results: Infilling

## Code Llama without Long Context fine-tuning

Model	FIM	Size	HumanEval			MBPP			Test loss
			pass@1	pass@10	pass@100	pass@1	pass@10	pass@100	
CODE LLAMA (w/o LCFT)	$\times$	7B	33.2%	43.3%	49.9%	44.8%	52.5%	57.1%	0.408
		13B	36.8%	49.2%	57.9%	48.2%	57.4%	61.6%	0.372
CODE LLAMA (w/o LCFT)	$\checkmark$	7B	33.6%	44.0%	48.8%	44.2%	51.4%	55.5%	0.407
		13B	36.2%	48.3%	54.6%	48.0%	56.8%	60.8%	0.373
Absolute gap	$\times - \checkmark$	7B	-0.4%	-0.7%	1.1%	0.6%	1.1%	1.6%	0.001
		13B	0.7%	0.9%	3.3%	0.2%	0.6%	0.8%	-0.001

- Question response: Infilling training incurs no cost on autoregressive test set loss, only small cost on pass@k where k is large

# Performance Results: fine-tuning

- ▶ Long context fine-tuning (LCFT)
  - ▶ Obvious perplexity decrease in large source files
  - ▶ Better completion performance

# Safety Consideration

- Another objective of fine-tuning
- Fine-tune bad data
  - Fake code
  - Malicious code
  - Biased code



**You**

Do you know how to hack the `mv` command?



**ChatGPT**

The term "hack" can imply different contexts depending on usage. In a constructive context, "hacking" the `mv` command could mean finding creative or unconventional ways to use it for various tasks beyond its basic functionality of moving or renaming files in Unix/Linux systems. However, if you're referring to malicious hacking or unauthorized access, it's important to note that discussing or engaging in such activities is unethical and illegal.

# Safety Consideration

- Truthfulness
  - TruthfulQA -> ensure the truth percent
- Toxicity
  - ToxiGen -> reduce hate speech
- Bias
  - BOLD datasets



# Safety Consideration

- Red teaming: avoid hacking
  - Avoid false refusal: the safety can't go too high
- Results:
  - After fine-tuning using the three datasets, a clear optimization is obtained
  - Ranked #2 safety performance

- Evaluation such as “Dual intent prompts”.
- Some possible ways to reduce the risk? Collect data? Simulation?

# Comparison

- Compared with traditional NLP methods:
  - More important in context, especially right context
- InCoder:
  - Based on InCoder Transformer
  - Infilling: Casual Masking
  - Different sequence
- Code Llama:
  - Based on Llama 2 model
  - Infilling: multi-task filling
  - Pipelines for generating different models, better fine-tuning

# Summary

- Code Llama: another zero-shot code synthesis model
- Different pipelines
- Safety Consideration

42

Any questions?

A thin vertical line is positioned to the right of the text "Any questions?". In the background, there are several light-colored, curved lines that sweep across the right side of the slide, creating a sense of movement and design.

Thank you!

## References

1. Fried, D., Aghajanyan, A., Lin, J., Wang, S., Wallace, E., Shi, F., Zhong, R., Yih, W.-t., Zettlemoyer, L., Lewis, M. (2023). InCoder: A Generative Model for Code Infilling and Synthesis. arXiv preprint arXiv:2204.05999.
2. Rozière, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X. E., Adi, Y., Liu, J., Sauvestre, R., Remez, T., Rapin, J., Kozhevnikov, A., Evtimov, I., Bitton, J., Bhatt, M., Ferrer, C. C., Grattafiori, A., Xiong, W., Défossez, A., Copet, J., Azhar, F., Touvron, H., Martin, L., Usunier, N., Scialom, T., Synnaeve, G. (2023). Code Llama: Open Foundation Models for Code. arXiv preprint arXiv:2308.12950.
3. Armen Aghajanyan, Bernie Huang, Candace Ross, Vladimir Karpukhin, Hu Xu, Naman Goyal, Dmytro Okhonko, Mandar Joshi, Gargi Ghosh, Mike Lewis, and Luke Zettlemoyer. CM3: A causal masked multimodal model of the Internet. arXiv preprint arXiv:2201.07520, 2022a.



44

Thank you!