Building Coding Assistants at Augment

The research behind Code Completion, Code Retrieval, and Next Edit

Jiayi Wei
March 2025

About Me

- Research Scientist and Founding Member at Augment Code
- Lead the AI research behind Code Completion & Next Edit
- How I got into AI for Code
  - Did my undergrad in Physics in China (2013-2017)
    - Fell in love with programming
    - AlphaGo moment (2016)
  - Did my PhD in CS at UT Austin (2017-2023)
    - PhD Advisor: Isil Dillig (programming languages)
    - Co-advisor: Greg Durrett (NLP)
  - Summer Internship at Facebook (2020)
    - Expression-level code autocomplete using GPT2
  - GitHub Copilot was out (2022)

Your AI pair programmer

GitHub Copilot uses the OpenAI Codex to suggest code and entire functions in real-time, right from your editor.

sentiments.ts | write_sql.go | parse_expenses.py | addresses.rb

```ts
#!/usr/bin/env ts-node

import { fetch } from "fetch-h2";

// Determine whether the sentiment of text is positive
// Use a web service
async function isPositive(text: string): Promise<boolean> {
  const response = await fetch(`http://text-processing.com/api/sentiment/`, {
    method: "POST",
    body: `text=${text}`,
    headers: {
      "Content-Type": "application/x-www-form-urlencoded",
    },
  });
  const json = await response.json();
  return json.label === "pos";
}
```

Copilot

Programming will never be the same…

Program synthesis with AlphaCode

Research

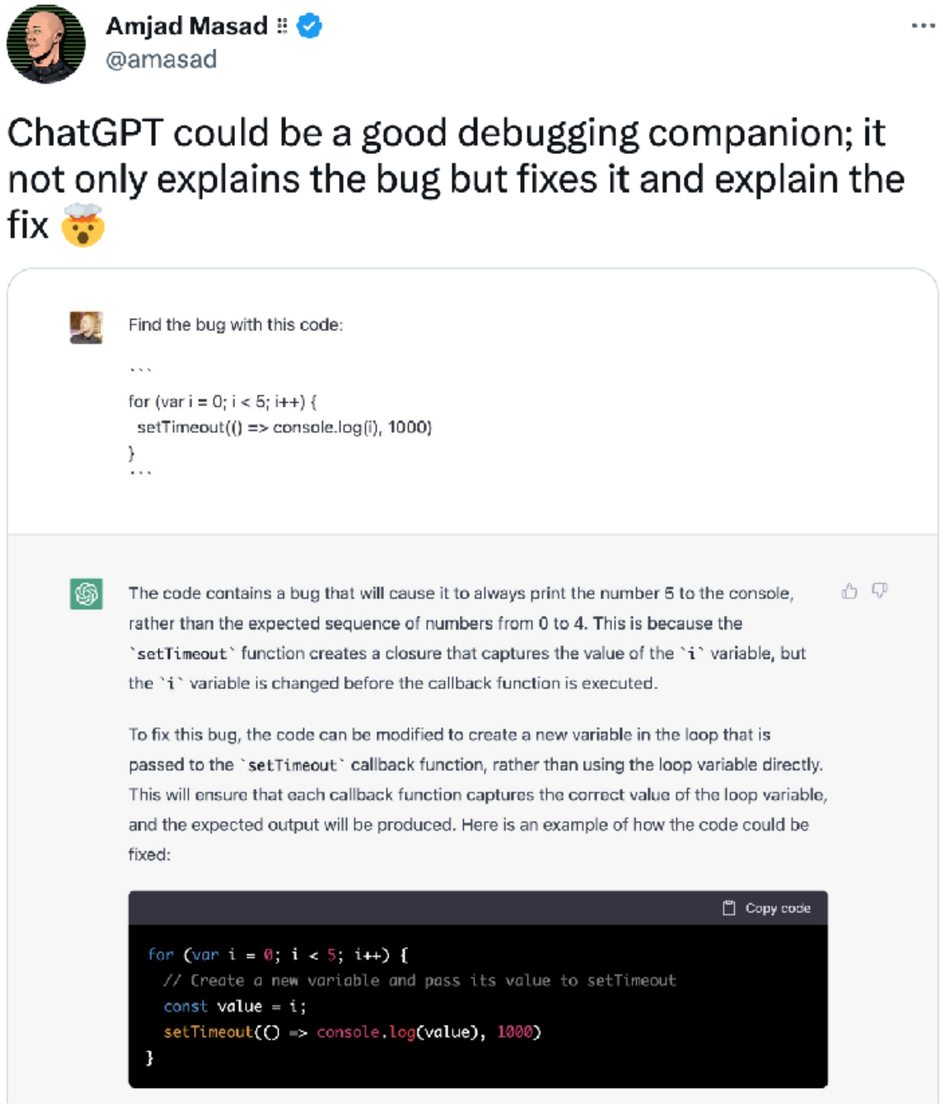
Competitive programming with AlphaCode

December 8, 2022

Note: This blog was first published on 2 Feb 2022. Following the paper's publication in Science on 8 Dec 2022, we've made minor updates to the text to reflect this.

Solving novel problems and setting a new milestone in competitive programming

Doing a lot more with ChatGPT



A limited view of programs

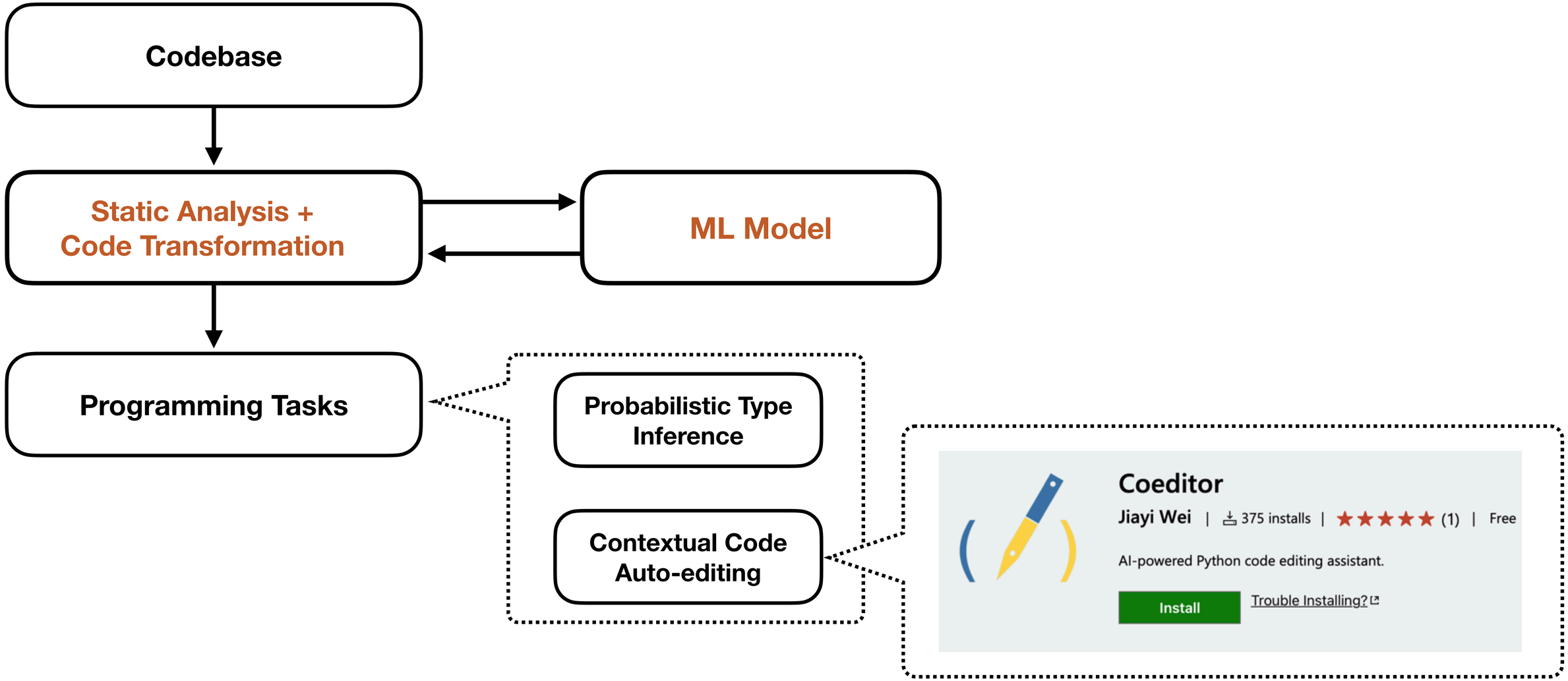
Competition-Level Code Generation with AlphaCode:

> Compared to decoder-only architectures commonly used for language modeling and generation, an encoder-decoder architecture allows a bidirectional description representation (tokens at the beginning of the description can attend to tokens at the end) and the extra flexibility to untie the encoder structure from the decoder. Because problem descriptions are on average twice as long as their corresponding human solutions, we use an asymmetric architecture with 1536 tokens for the encoder but only 768 tokens for the decoder. We further found that using a shallow encoder and a deep decoder significantly improves the efficiency of training without hurting problem solve rate. The exact architectures for our models are listed in Table 3. The 9B and 41B models were trained using model parallelism, with 1 key and value head per shard. We built our model using JAX (Bradbury et al., 2018) and Haiku (Hennigan et al., 2020), and trained them on TPUv4 accelerators using bfloat16 precision.

CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation:

> We build CodeT5 based on Huggingface's T5 (Raffel et al., 2020) PyTorch implementation[3] and employ two sizes of CodeT5-small (60M) and CodeT5-base (220M). We set the maximum source and target sequence lengths to be 512 and 256, respectively. We use the mixed precision of FP16 to accelerate the pre-training. We set the batch size to 1024 and employ the peak learning rate of 2e-4 with linear decay. We pre-train the model with the

My PhD Thesis: Machine Learning + Static Analysis



My 2 years at Augment

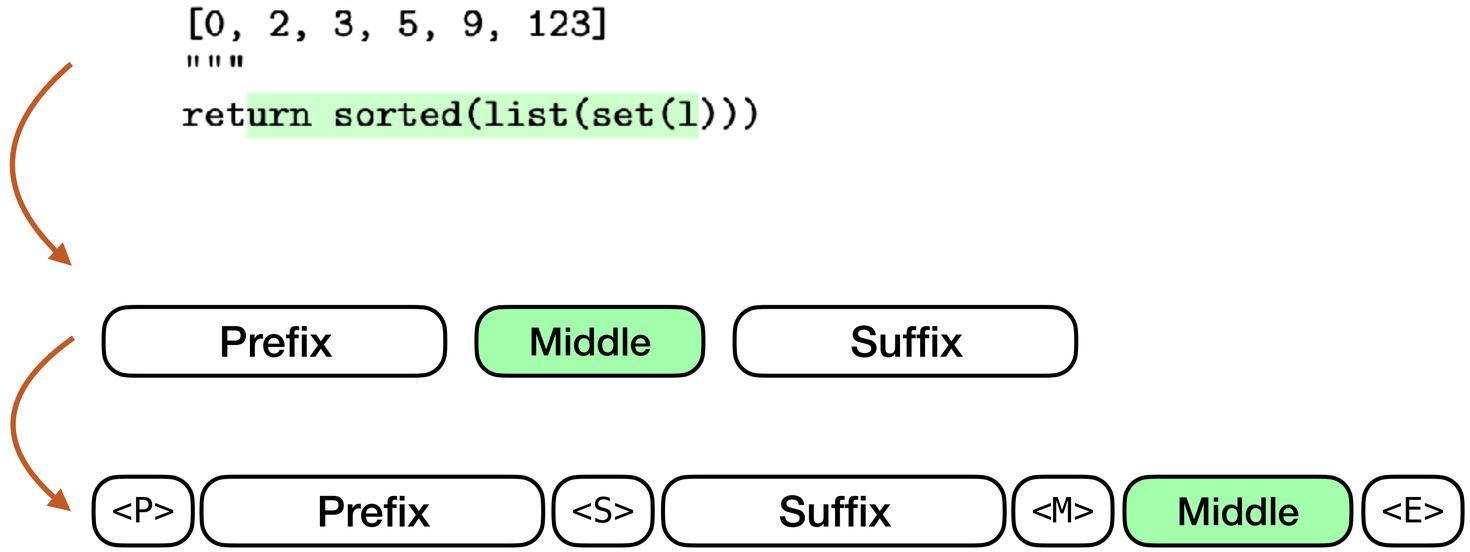
- Joined Augment Code as a founding member (2023-present)
- More about Augment
  - Building coding assistants focusing on enterprise developers and large codebases
  - https://www.augmentcode.com/
- Project 1: Improve FIM for our completion model (~4 months)
- Project 2: Signature retrieval (~8 months)
  - Used some ideas from our type inference work
- Project 3: Next Edit (~1 year)
  - Used lots of ideas from our Coeditor work

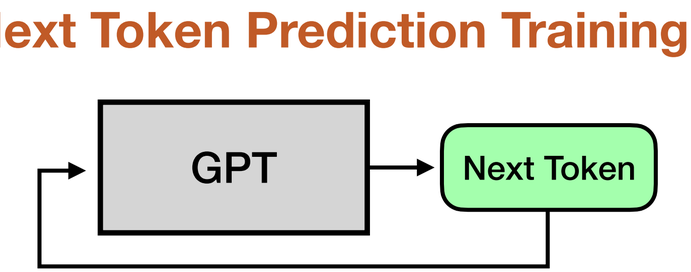
How to train a Code Completion model

The fill-in-the-middle (FIM) task

- "Efficient Training of Language Models to Fill in the Middle", Bavarian et al 2022

```
def unique(l: list):
    """Return sorted unique elements in a list
    >>> unique([5, 3, 5, 2, 3, 3, 9, 0, 123])
    [0, 2, 3, 5, 9, 123]
    """
    return sorted(list(set(l)))
```





Issues with standard FIM training

Model sometimes give unlikely suggestions

Bad

```
MODEL: starcoder_16b
EXAMPLE: fim_argparse.py ✗ | model=starcoder_16b, temp=0, sample=0
import argparse

parser = argparse.ArgumentParser()
parser.add_argument(
    "--model",
    type=str,
    default="my_model",
    help="which model to serve",
)
parser.add_argument(
    "--disable_completions",
    action="store_true",
    help="disable completions",
)
parser.add_argument(
    "--disable_completion_requests",
    action="store_true",
    help="disable completion requests",
)
parser.add_argument(
    "--single_line_completions",
    action="store_true",
    help="whether to return single-line completions",
)
```

Good

```
)
[11] ✓ 45.8s
MODEL: new-FIM-starcoder-python-559K
EXAMPLE: fim_argparse.py ✓ | model=new-FIM-starcoder-python-559K, temp=0, sample=0
import argparse

parser = argparse.ArgumentParser()
parser.add_argument(
    "--model",
    type=str,
    default="my_model",
    help="which model to serve",
)
parser.add_argument(
    "--disable_completions",
    action="store_true",
    default=False,
    help="disable completion requests",
)
parser.add_argument(
    "--single_line_completions",
    action="store_true",
    help="whether to return single-line completions",
)

Expected: default=False,
```

Issues with standard FIM training, ex 2

Bad

```
Expected:


================================================================================
EXAMPLE: fim_demo2.py ✗ | model=starcoder_16b, temp=0, sample=0
--------------------------------------------------------------------------------
import dataclasses
import datetime

from git import Sequence

@dataclasses.dataclass
class Person:
    name: str
    dob: datetime.date
    profession: str
    home_address: str
    parents: Sequence["Person"]
    children: Sequence["Person"]

    def __str__(self):
        return f"{self.name} is a {self.profession} and lives at {self.home_address}"

    def age(self):
        today = datetime.date.today()
        age = today.year - self.dob.year
        if today.month < self.dob.month or (today.month == self.dob.month and today.d
            age -= 1
        return f"{self.name} is {age} years old"

    def salary(self):
        raise NotImplementedError()
================================================================================
Expected:


================================================================================
```

Good

```
    def salary(self,
================================================================================
Expected:


================================================================================
EXAMPLE: fim_demo2.py ✗ | model=new-FIM-starcoder-python-559K, temp=0, sample=0
--------------------------------------------------------------------------------
import dataclasses
import datetime

from git import Sequence

@dataclasses.dataclass
class Person:
    name: str
    dob: datetime.date
    profession: str
    home_address: str
    parents: Sequence["Person"]
    children: Sequence["Person"]

    def __str__(self):
        age = datetime.date.today() - self.dob
        return f"{self.name} is {age} years old"

    def salary(self):
        raise NotImplementedError()
================================================================================
Expected:


================================================================================
EXAMPLE: fim_docstring_middle_fix.py ✗ | model=new-FIM-starcoder-python-559K, temp=0, sampl
--------------------------------------------------------------------------------
def get_odd_ints(lower, upper):
```

Issues with standard FIM training, ex 3

Bad

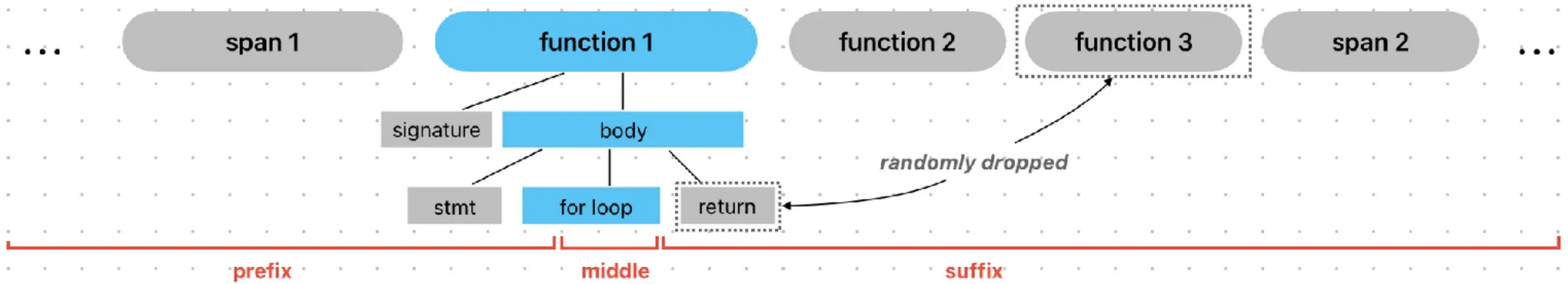
```
    middle_dividers: list[int]
    """Character positions that divide the middle into syntactic units.
    These indices are relative to the middle string."""

    prefix_dividers: list[int]
    """Character positions that divide the prefix into syntactic units.
    These indices are relative to the prefix string."""

    suffix_dividers: list[int]
    """Character positions that divide the suffix into syntactic units.
    These indices are relative to the suffix string."""

    prefix_range: CharRange
    """The character range of the prefix span."""

    suffix_range: CharRange
    """The character range of the suffix span."""

    @property
    def prefix_length(self) -> int:
        return len(self.prefix)

    @property
    def suffix_length(self) -> int:
        return len(self.suffix)

    @property
    def middle_length(self) -> int:
        return len(self.middle)

    @property
    def prefix_middle_length(self) -> int:
        return self.prefix_length + self.middle_length"""

    def __str__(self) -> str:
        return self.show()
```

Good

```
    prefix: str

    suffix: str

    middle_range: CharRange
    """The character range of the middle span."""

    file_path: Path
    """The path of the file from which this problem is constructed."""

    middle_node_type: TsNodeType
    """Type of the tree-sitter node that the middle belongs to."""

    middle_dividers: list[int]
    """Character positions that divide the middle into syntactic units.
    These indices are relative to the middle span."""

    def __str__(self) -> str:
        return self.show()

    def __bool__(self) -> Literal[True]:
        return True

    def show(self, prefix_limit: int = 1000, suffix_limit: int = 1000):
        SEP = "-" * 50
        prefix = self.prefix
        if len(prefix) > prefix_limit:
            prefix = "[...]" + prefix[len(prefix) - prefix_limit :]
        suffix = self.suffix
        if len(suffix) > suffix_limit:
            suffix = suffix[:suffix_limit] + "[...]"
        parts = [
            f"FimProblem(file={self.file_path}, middle={self.middle_range}):",
            f"{SEP}Prefix{SEP}",
            prefix,
            f"{SEP}Middle ({self.middle_node_type}){SEP}"
```

Better FIM training using program structure

- Real middle spans are not uniformly randomly distributed
- We can sample more realistic middle spans by taking AST structure into account
- Model consider realistic solutions more if you exclude unrealistic examples from the training data
- In WIP code, multiple code snippets can be missing



Better FIM training: Result

- We see improved code completion performance across different languages
  - including the ones we didn't train on
- The model suggestions are more intuitive and predictable
- Using AST structure, we further taught the model how to `<|pause|>` and `<|skip|>`

```
<fim-prefix>
def some_long_function(...):
  x = [x_expression]
  y = [y_expr_left]<fim-middle>[y_expr_right] <pause>
  z = [z_expression] <pause>
  for value in [x,y,z]:
    # do something with the value
    ... <pause>
  [statement after the for loop]<fim-stop>
  <fim-suffix>
  return [return_expression]
```

```
74  def make_or_load_ground_truth_data(
75      cache: PickleCache,
76      all_data: Sequence[NextEditIntermediateType],
77      max_repo_files: int = 40_000,
78      min_probs_per_session: int = 50,
79      max_workers: int = 16,
80  ) -> tuple[list[tuple[NextEditIntermediateTy…:
81      n_before = len(all_data)
82      all_data = [
83          datum for datum in all_data if      Add: datum.request.blob_names
84      ]
85      n_after = len(all_data)
86      # reorder the data by session id to help reduce pickling during pmap
87      all_data = sorted(all_data, key=lambda d: d.session_id)
88      print(
89          f"Keeping {n_after} ({n_after / n_before:.1%}) examples with <= {max_repo_files}
90      )
```

Coeditor:
Leveraging Repo-level diffs for Code Auto-editing

Jiayi Wei, Greg Durrett, Isil Dillig

April 2024

Coeditor VSCode demo

Contributions

New Task

- Multi-round code auto-editing
- The PyCommits dataset*

Coeditor Model

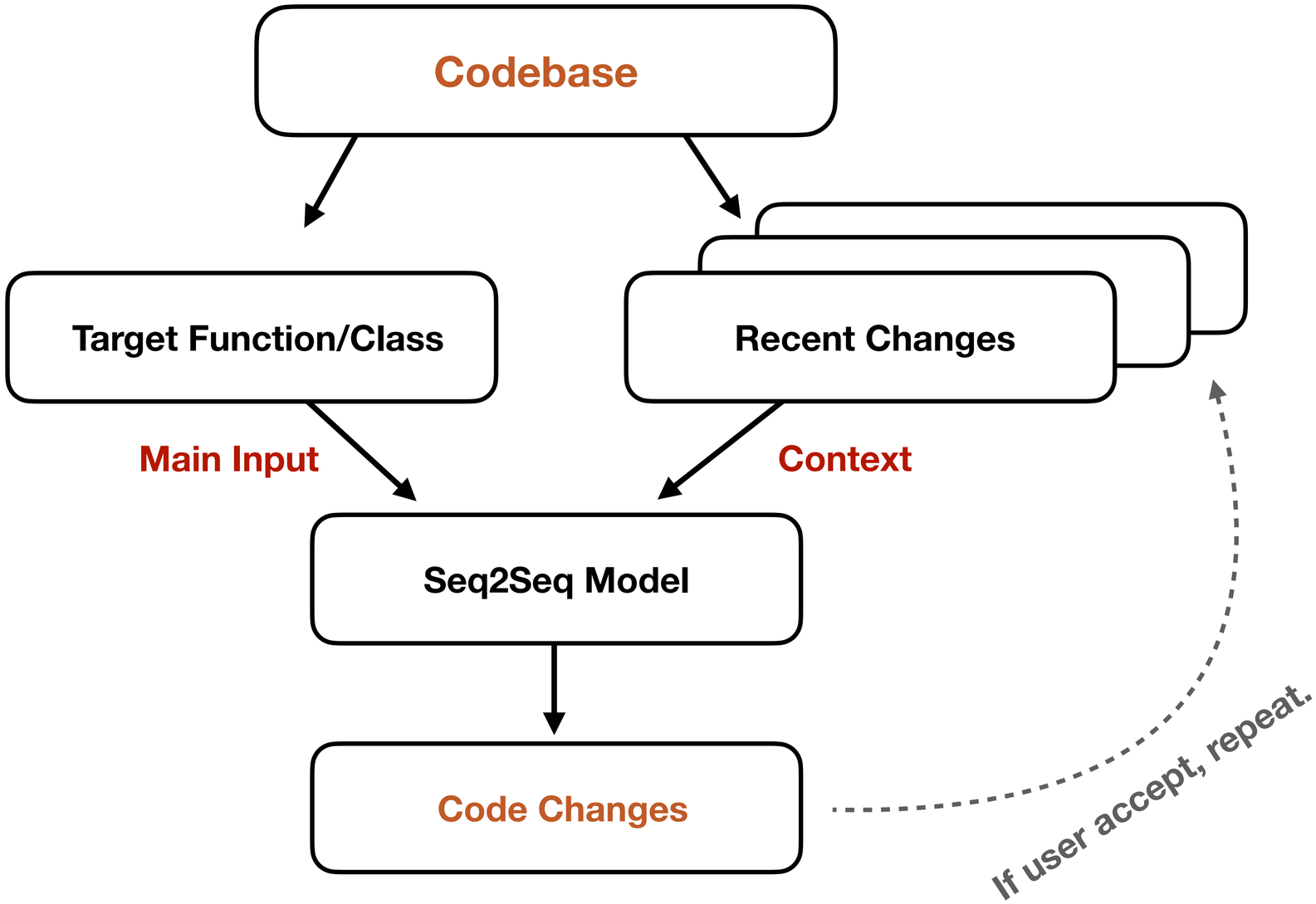
- Encoding/decoding changes using line diffs
- Adapted CodeT5 with sparse attention*
- Retrieval Augmented using static analysis*

Evaluation

- Comparing with completion models
- Multi-round editing*
- Ablation studies*

(*) Not in this talk.

Setting: Multi-round code auto-editing



Encoding code changes…

…as line diffs

```
[...]
            if ref_size_sum + len(ref) <= args.max_total_ref_
                ref_selected.append(ref)
                ref_size_sum += len(ref)
<add>       input_tks, output_tks = process_edit(edit, args)
        ex_cost = retrieval_cost_model(
            ref_size=sum(len(x) for x in ref_selected),
<add>            query_size=len(input_tks),
<del>            query_size=len(input_tks_list[i]),
<add>            output_size=len(output_tks),
<del>            output_size=len(output_tks_list[i]),
        )
        ref_selected.sort(key=lambda x: id2ref_name[id(x)])
        row = {
<add>            "input_tks": input_tks,
<del>            "input_tks": input_tks_list[i],
<add>            "output_tks": output_tks,
<del>            "output_tks": output_tks_list[i],
            "ref_selected": ref_selected,
<add>           "cost": ex_cost,
        }
        if ex_cost > cost_limit:
            warnings.warn("Batch cost limit is too small.")
        if ex_cost + current_cost <= cost_limit:
[...]
```

Decoding code changes

```
[...]
            ref_size_sum = 0
            ref_selected = list[TokenSeq]()
            for ref in all_refs:
<1>             if ref_size_sum + len(ref) <= args.max_total_
<2>                 ref_selected.append(ref)
<3>                 ref_size_sum += len(ref)
<4><add>        input_tks, output_tks = process_edit(edit, ar
<5>         ex_cost = retrieval_cost_model(
<6>             ref_size=sum(len(x) for x in ref_selected),
<7>             query_size=len(input_tks_list[i]),
<8>             output_size=len(output_tks_list[i]),
<9>         )
<10>        ref_selected.sort(key=lambda x: id2ref_name[id(x)
<11>        row = {
<12>            "input_tks": input_tks_list[i],
<13>            "output_tks": output_tks_list[i],
<14>            "ref_selected": ref_selected,
<15><add>           "cost": ex_cost,
<16>        }
<17>        if ex_cost > cost_limit:
<18>            warnings.warn("Batch cost limit is too small.
<19>        if ex_cost + current_cost <= cost_limit:
[...]
```

Input: adds placeholder tokens to indicate code region to edit.

```
<6>
<7> <add>           query_size=len(input_tks),
    <del>
<8> <add>           output_size=len(output_tks),
    <del>
[...]
<12> <add>          "input_tks": input_tks,
     <del>
<13> <add>          "output_tks": output_tks,
     <del>
<14>
```

Output: specifies further changes at each placeholder token (if any).

Decoding code changes

```
[...]
            ref_size_sum = 0
            ref_selected = list[TokenSeq]()
            for ref in all_refs:
<1>             if ref_size_sum + len(ref) <= args.max_total_
<2>                 ref_selected.append(ref)
<3>                 ref_size_sum += len(ref)
<4><add>        input_tks, output_tks = process_edit(edit, ar
<5>         ex_cost = retrieval_cost_model(
<6>             ref_size=sum(len(x) for x in ref_selected),
<7>             query_size=len(input_tks_list[i]),
<8>             output_size=len(output_tks_list[i]),
<9>         )
<10>        ref_selected.sort(key=lambda x: id2ref_name[id(x)
<11>        row = {
<12>            "input_tks": input_tks_list[i],
<13>            "output_tks": output_tks_list[i],
<14>            "ref_selected": ref_selected,
<15><add>           "cost": ex_cost,
<16>        }
<17>        if ex_cost > cost_limit:
<18>            warnings.warn("Batch cost limit is too small.
<19>        if ex_cost + current_cost <= cost_limit:
[...]
```

Input: adds placeholder tokens to indicate code region to edit.

```
<7> <add>           query_size=len(input_tks),
    <del>
<8> <add>           output_size=len(output_tks),
    <del>
```

```
<12> <add>          "input_tks": input_tks,
     <del>
<13> <add>          "output_tks": output_tks,
     <del>
```

Output: specifies further changes at each placeholder token (if any).

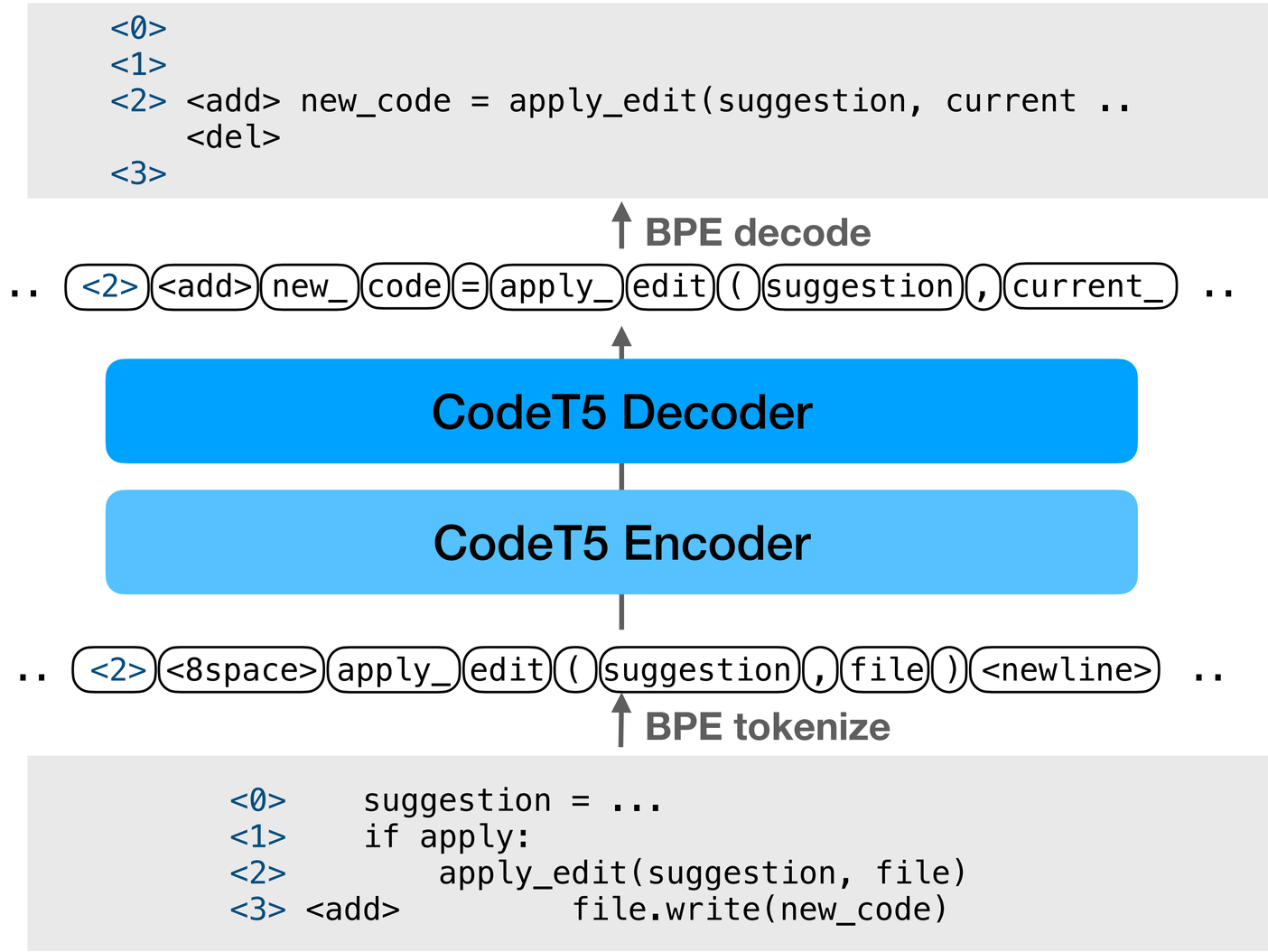
Decoding code changes

```
[...]
            ref_size_sum = 0
            ref_selected = list[TokenSeq]()
            for ref in all_refs:
<1>             if ref_size_sum + len(ref) <= args.max_total_
<2>                 ref_selected.append(ref)
<3>                 ref_size_sum += len(ref)
<4><add>        input_tks, output_tks = process_edit(edit, ar
<5>         ex_cost = retrieval_cost_model(
<6>             ref_size=sum(len(x) for x in ref_selected),
<7> <add>           query_size=len(input_tks),
    <del>
<8> <add>           output_size=len(output_tks),
<9>         )
<10>        ref_selected.sort(key=lambda x: id2ref_name[id(x)
<11>        row = {
<12> <add>          "input_tks": input_tks,
    <del>
<13> <add>          "output_tks": output_tks,
    <del>
<14>            "ref_selected": ref_selected,
<15><add>           "cost": ex_cost,
<16>        }
<17>        if ex_cost > cost_limit:
<18>            warnings.warn("Batch cost limit is too small.
<19>        if ex_cost + current_cost <= cost_limit:
[...]
```

💡 Masked Span Infilling! 💡

Model Architecture: CodeT5

- We finetune CodeT5-base (220M params), which was pre-trained on masked span infilling



Simulating Partial Changes

Past Changes

Editing Scope

Model Output

```
1   Model Input:
2   <add> def apply_edit(edit: Edit, text: str) -> str:
3   <del> def apply_edit(edit: Edit, file: File):
4   <add>     "Apply the edit to the given string"
5   <del>     "Apply the edit to the content of given file"
6             # omitted...
7
8   def suggest_edit(
9       file: Path,
10      line: int,
11      apply: bool = False
12  ):
13  <mask_0>    suggestion =...
14  <mask_1>    if apply:
15  <mask_2>        apply_edit(suggestion, file)
16  <mask_3>
17
18  Model Output:
19  <mask_0>
20  <mask_1>
21  <mask_2> <add>          new_code = apply_edit(suggestion, current_code)
22           <del>
23  <mask_3> <add>          file.write(new_code)
```

Simulating Partial Changes

Past Changes

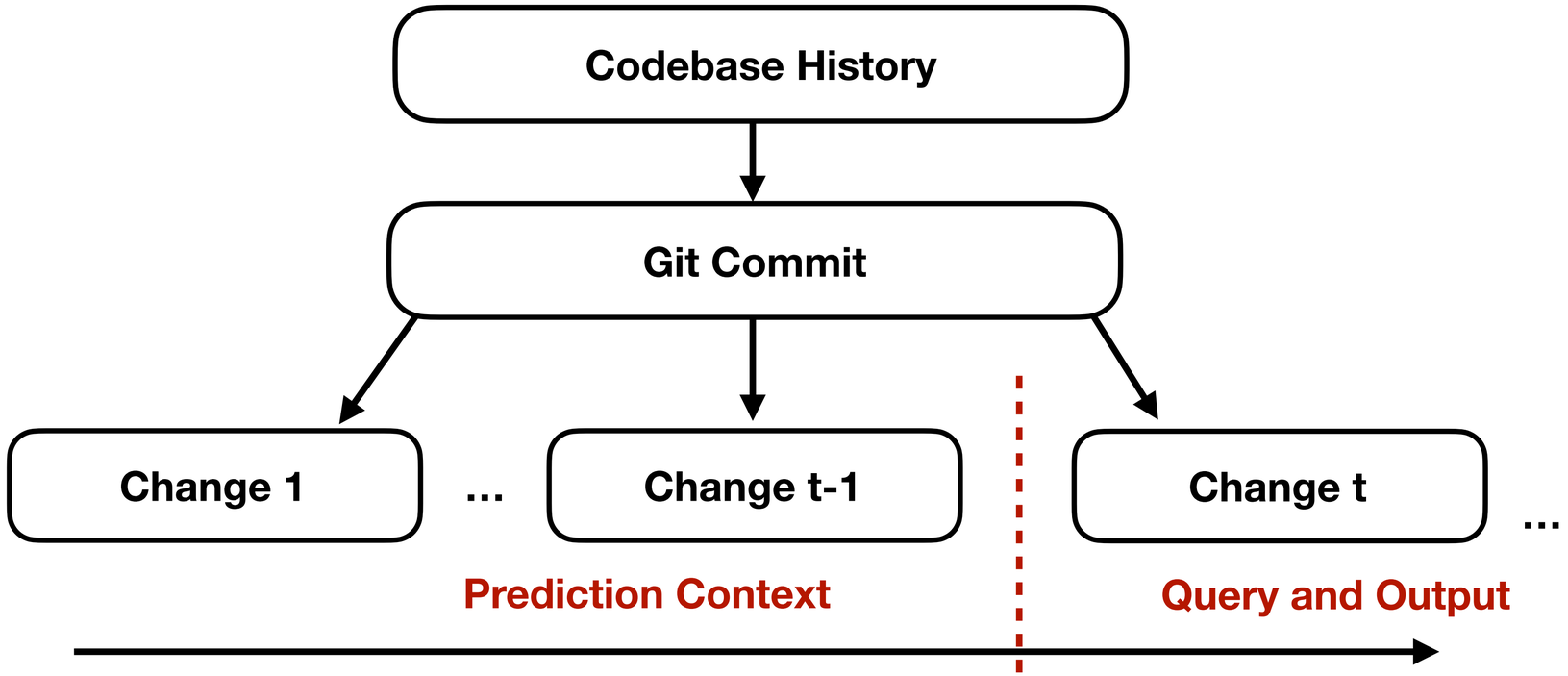
Editing Scope

Model Output

```
1   Model Input:
2   <add> def apply_edit(edit: Edit, text: str) -> str:
3   <del> def apply_edit(edit: Edit, file: File):
4   <add>     "Apply the edit to the given string"
5   <del>     "Apply the edit to the content of given file"
6             # omitted...
7
8   def suggest_edit(
9       file: Path,
10      line: int,
11      apply: bool = False
12  ):
13  <mask_0>    suggestion =...
14  <mask_1>    if apply:
15  <mask_2>        apply_edit(suggestion, file)
16  <mask_3> <add>          file.write(new_code)
17
18  Model Output:
19  <mask_0>
20  <mask_1>
21  <mask_2> <add>          new_code = apply_edit(suggestion, current_code)
22            <del>
23  <mask_3>
```

Constructing the PyCommits Dataset

- Split the changes in each commit by syntactic scopes
- Within a commit, assume a linear modification order



Sorted by location and file imports

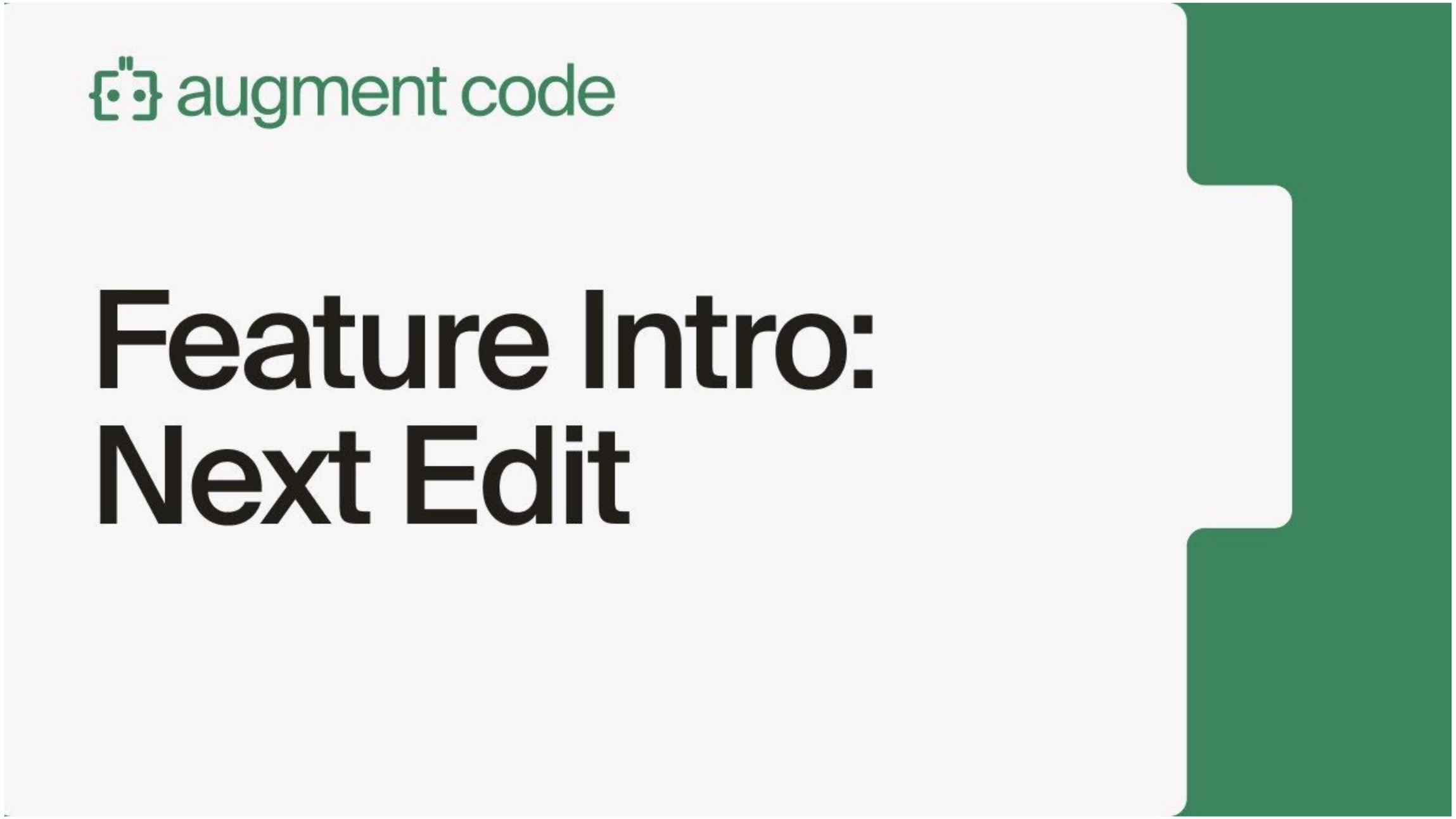
Comparison with Code Completion Models

- Test data: use last changed line from real commits as completion target
- Code completion models only see current version of the code
- Coeditor also sees same-commit changes

Table 3: Performance on 5000 code completion instances extracted from edits (PyCommits-OneLine). Add EM and Replace EM are the (enhanced) exact-match accuracies on addition and replacement change, respectively.

Model	Parameters	Context	Exact Match Rate (%)		
			Add	Replace	Overall
InCoder1B	1.3B	2K	29.0	25.2	26.2
InCoder6B	6.7B	2K	34.0	30.4	31.3
SantaCoder	1.1B	2K	31.0	28.1	28.8
StarCoder7B	7B	8K	37.9	33.7	34.8
text-davinci-003	175B	4K	40.2	39.3	39.5
Coeditor	220M	16K	**47.1**	**64.9**	**60.4**

The Next Edit feature: background mode



The Next Edit feature: global mode

(starts around 2:00)

Improvements over Coeditor

- Predicts when and where to make the next change
  - v.s. Coeditor requires the user to specify which snippet to edit next
- Works on any programming language
  - v.s. Coeditor only works on Python
- Can edit lines that have been already been modified
  - v.s. Coeditor is trained to only edit unmodified lines
  - This requires feeding model a more granular edit history that distinguish newer edits from older ones
- Switched to a much larger model trained on much more data

The 3 big challenges when building Next Edit

- Figuring out **what** tasks the user is trying to accomplish
- Figuring out **where** to make those changes
- Figuring out **how** to make the edits

Figuring Out What Tasks the User Is Trying to Accomplish

Challenges:

- **Non-linear Editing Histories**
  - copy a block of code, paste it elsewhere, and immediately modify it significantly.
  - make multiple edits in rapid succession, frequently switching between files or functions
  - undo and redo changes as they experiment with different implementations.
  - These non-linear workflows create a messy trail of changes that can mislead a model trying to infer the developer's true intent.
- **Unintended Biases**
  - For example, the model might learn to avoid touching parts of the codebase that already contain recent changes. This is problematic because those areas might be precisely where further edits are needed.
- **Intention Hallucination**
  - Models might hallucinate intentions, suggesting changes that are not directly related to the user's recent edits.
  - This occurs when the model tries to be overly proactive, aiming to cover all possible relevant edits (high recall), which can result in noisy and disruptive suggestions.
  - There's a delicate balance between being "helpful but noisy" and "accurate but passive."

Figuring Out What Tasks the User Is Trying to Accomplish

Solution:

- **Simulating Common Editing Scenarios:**
  - developed a sophisticated algorithm to simulate realistic editing scenarios that reflect common developer behaviors by analyzing commit messages, initial commit states, and final commit states
- **Optimizing Diff Granularity:**
  - taught our models to read fine-grained editing events while carefully optimizing the granularity of diffs presented in the prompt
  - too fine-grained -> model may be distracted by the noise in the user’s editing history
  - too coarse, model may struggle to distinguish newer changes from older ones
- **Avoiding Undoing User's Recent Changes:**
  - model sometimes had a strong tendency to undo the user's recent changes
  - made special efforts to improve our training samples to discourage this behavior
  - add inference-time filter to prevent undoing edits

Figuring Out Where to Make Those Changes

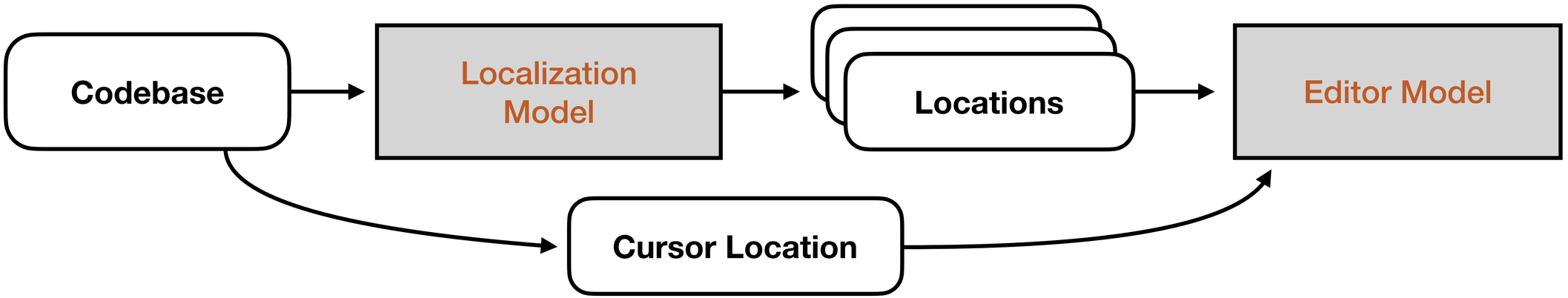
Challenges:

- **Scalability**
  - The localization mechanism needs to be scalable to handle large-scale codebases efficiently without consuming excessive resources.
- **Speed**
  - It needs to be extremely fast to support highly interactive usage patterns, providing immediate suggestions as the user makes new changes.
- **Relevance**
  - It must accurately identify relevant locations without overwhelming the user with unnecessary suggestions or missing important ones.

Figuring Out Where to Make Those Changes

Solution:

- **Edit Localization with a Trained Retriever**
  - We trained a fast retriever model specifically designed to identify code locations likely to require updates
- **Editing Surrounding Code first**
  - The code around the user's cursor is always added to the list of candidate locations and processed first without waiting the localization model



Figuring Out How to Make the Edits

Challenges:

- **Complex Edits Beyond Cursor Insertions**
  - We need to adapt the Coeditor’s T5-based architecture to be compatible with the newest open-source LLMs, which all use a decoder-only architecture.
- **Latency Constraints**
  - Generating edits should be fast enough for real-time usages.
  - It should be really cheap for the model to reject a given location.
- **Codebase Awareness**
  - Suggestions need to match the project's coding standards, conventions, and correctly use custom APIs.

Figuring Out How to Make the Edits

Solution:

- **Novel Diff Decoding Scheme**
  - We taught the model to output a specialized diff format that is both compact and unambiguously applicable to the original code.
  - This format minimizes the number of tokens generated and enables efficient processing of large files.
- **Codebase-Aware Suggestions**
  - We leveraged our powerful Retrieval Augmented Generation (RAG) infrastructure to add codebase-specific context to Next Edit.
  - The model conditions its prediction on both the recent edits and retrieved information that's specific to the given location.
  - This ensures that the edits are not only aligned with what the user is trying to accomplish but also consistent with existing APIs and coding patterns

Summary

- We're now working on scaling up Next Edit to handle larger-scale changes, from enhancing the model's ability to understand broader contexts and dependencies to supporting bulk edits across many files simultaneously.
- We're also exploring deeper integration with our chat functionality, which could provide additional context for edit suggestions and enable more interactive problem-solving workflows.

My learnings

- Focus on impact
  - Avoid working on things that don’t matter in the medium-to-long run.
  - Think about opportunity cost.
- Optimize for iteration speed
  - Don’t try to do things in the most perfect way. Others likely will have a different opinion on what’s best.
  - Use evolvable designs. Be prepared to adapt based on new information.
  - Avoid unnecessary coupling. Less coupling -> more people can work in parallel.
  - Make things super easy to reuse. People tend to only reuse something if it’s easy.
  - Constantly ask yourself: are there ways to move faster?