LLM Code Generation and Test Synthesis

Multi-Agentic Orchestration for Partial Order Estimation of Code Edits

Lee Zong Xun

School of Computing
National University of Singapore

August 28, 2025

Overview

1. Introduction
2. Methodology
3. Approach
4. Evaluation
5. Experimental Results
6. Conclusion

Motivation

Before 2022: Manual Coding

- Developers wrote every line of code manually. Fundamentally human-driven.
- **Tools:** Syntax highlighting, debugging, auto indentation.

In 2022: Revolutionary AI Copilots

- **GitHub Copilot** leveraged **OpenAI's Codex** to introduce AI-assisted coding.
- Developers could:
  - Auto-complete entire functions.
  - Receive explanations for code snippets.
  - Reduce boilerplate coding.

Motivation

Fast forward to 2024...and today

- Unlike Copilots, AI Agents work autonomously.
- They perform **multi-step problem-solving**, including:
  - Identifying issues.
  - Writing and refactoring code.
  - Running tests and debugging.
- Examples:
  - **ManusAI** - Automates entire workflows.
  - **Deep Research** by Google and OpenAI

Modern AI-IDEs

- CursorAI
- Windsurf
- TraeAI

Result? Productivity saw significant improvements, with some companies experiencing up to 10x efficiency gains.

Modern AI-IDEs

AI-IDEs generate high-entropy edits across multiple files at once. This is in contrast to earlier versions where they could only generate predictions locally and one at a time.

- Developers pass control to AI agents to build applications, and complete tasks.
- Inclusive (human-in-the-loop) design enforces that the user should always be in control (or at least be aware of the actions of the AI agents).

Challenges

Can we evaluate the coherence of the sequence of edits that the model suggests?

- No benchmark on what it means to generate logical and coherent edit suggestions.
- Compromises mental flow.
  - Developers need to manually adjust AI-generated code.
  - Need to context switch frequently.

Objective

To enable the evaluation of LLMs in next edit suggestions, we first need to recover the original commit process.

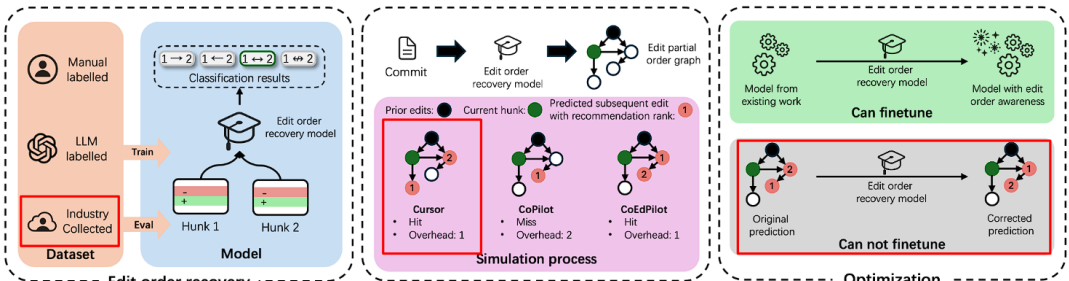


Figure: Research Plan

Problem Formulation

Objective: Determine the partial ordering of code edits in a set $\mathbf{H}$.

- Given two edits $h_i, h_j \in \mathbf{H}$, define their sequential relationship:

$$f(h_i, h_j) = \begin{cases} h_i \rightarrow h_j, & h_i \text{ must occur before } h_j \\ h_i \leftarrow h_j, & h_j \text{ must occur before } h_i \\ h_i \leftrightarrow h_j, & \text{no strict order} \\ h_i \times h_j, & \text{independent edits} \end{cases}$$

Construct a Directed Acyclic Graph (DAG) to model dependencies among edits.

Partial Order Reduction (POR)

Challenge: The number of possible orderings grows exponentially.

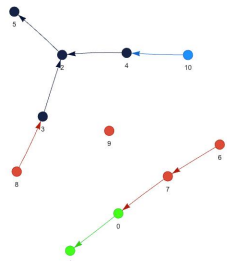


Figure: Partial order reduction reduces the search space from 13,860 to 36.

Solution: Apply POR to prune unnecessary interleavings of independent edits while preserving essential dependencies.

Key Components of POR

- **Independence:** Edits are independent if their execution order does not affect the system state.

$$t_1(s) \xrightarrow{t_2} s' \iff t_2(s) \xrightarrow{t_1} s' \text{ (Commutativity)}$$

and execution of one transition does not disable the other (Non-interference):

$$t_1 \text{ is enabled in state } s \implies t_2 \text{ is also enabled in } t_1(s).$$

- **Dependency:** If two edits influence each other, their order must be preserved.
- **Equivalence Classes:** Equivalent execution paths are merged to reduce complexity.
- **Reduction Criterion:** Every equivalence class of executions contains at least one representative path that is explored.

$$\forall t \in T, \exists t' \in T_{\text{red}}(s) : \quad t \sim t'.$$

Dependency Analysis

Key Observations:

- A single edit region may contain multiple edit semantics, which may require multiple rounds of parsing before full processing.
- The order of edits should be arranged based on their dependencies to ensure correct sequencing.
- Independent edits can be applied concurrently with no regards to consequences.
- Edits within the same scope are more likely to occur sequentially due to their local dependencies and proximity.
- Inter-file dependencies are often mediated through shared components (e.g., imports, global variables, or shared libraries).

Observation 1

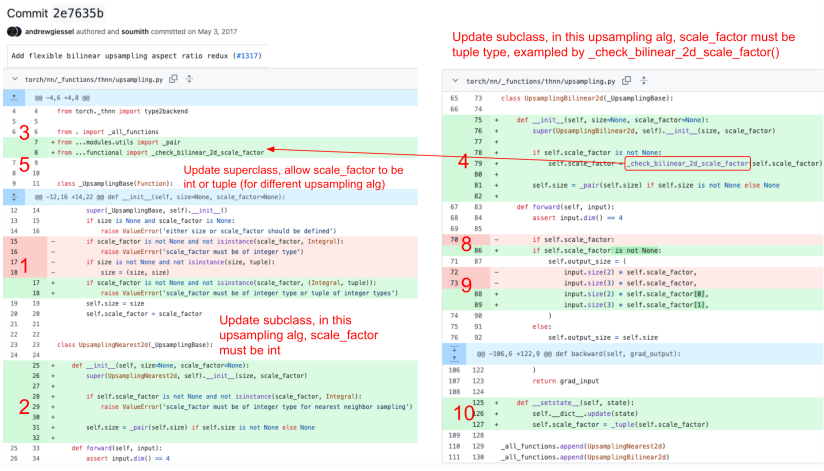


Figure: One edit region may contain multiple edit semantic, that may be visited twice

Observation 2

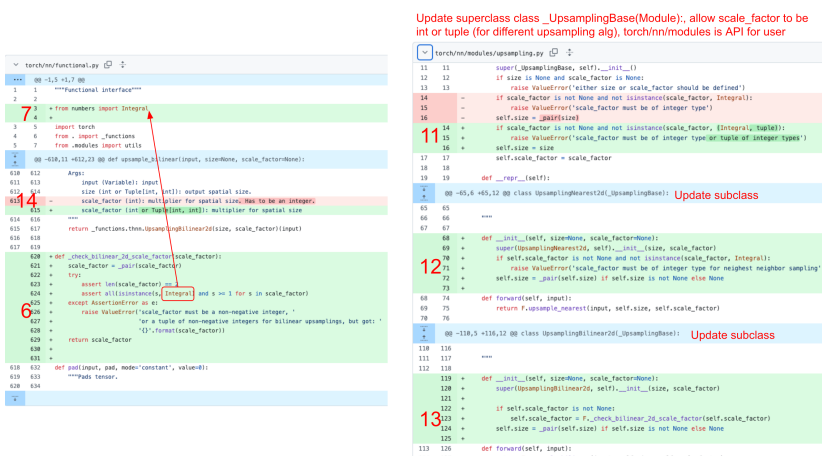


Figure: Arrange edit order by their dependency

Observation 3

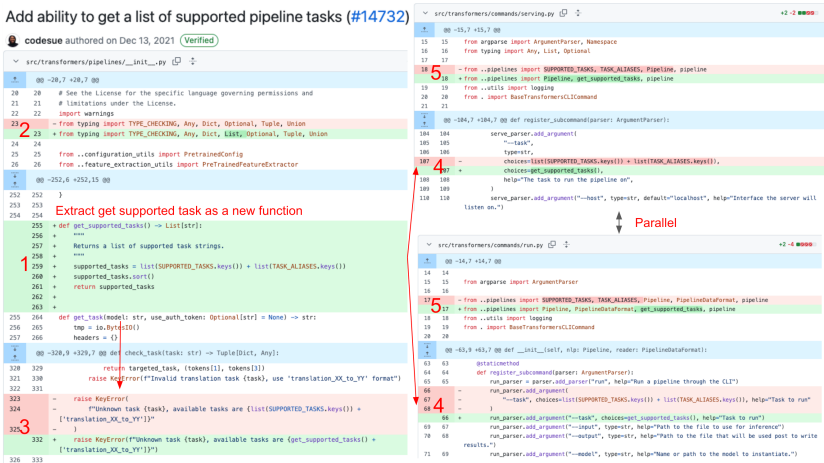


Figure: Parallelism between similar edit compositions

Observation 4

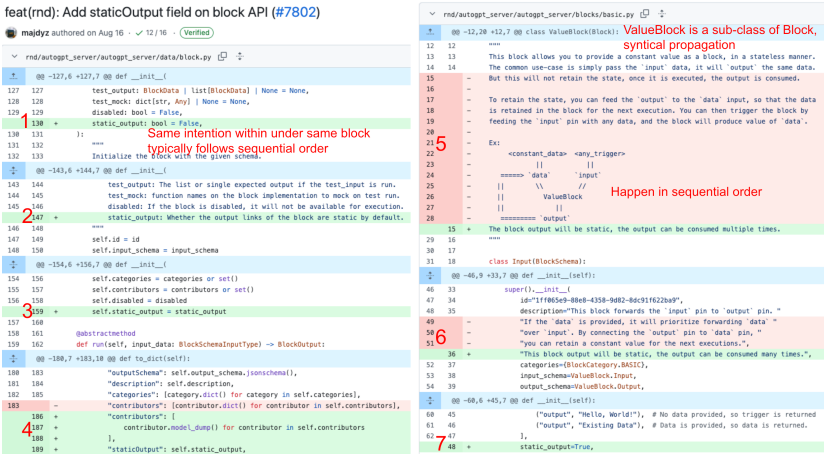


Figure: Edits in the same block are more likely to happen in sequential order

Observation 5

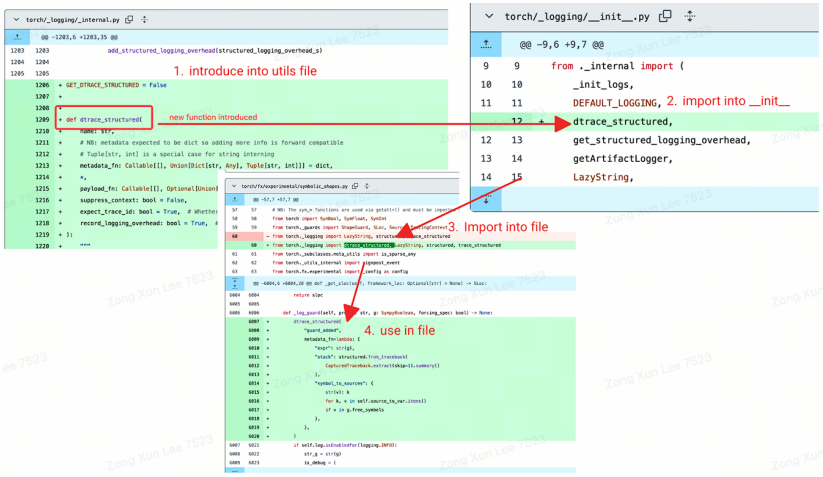


Figure: Inter-file dependencies are often mediated through shared component

Types of Dependencies

Syntactic Arise from structural relationships within the AST.
Example: Edits affecting function definitions, imports, etc.

Semantic Derived from the *behavior* of the code.
Example: Dependencies based on control flow and scope.

Proximity Inherent to the *position* of the code.

Dependency Derivation

Goal: Estimate the dependencies between edits $h_i, h_j \in \mathbf{H}$ using multiple approaches.

- **Heuristic:** Based on observed patterns and logical rules.
- **Static Analysis:** Uses AST, LSP, and code structure.
- **Agentic:** Dependency inference using multiple worker agents.

Heuristic Approach

Key Idea: Establish edit order based on common patterns.

1. **Edit Priority:** Deletions $h_i \in \mathbf{H}_{\text{del}}$ precede additions $h_i \in \mathbf{H}_{\text{add}}$.
2. **Top-Down Order:** Edits within a single file are applied from top to bottom, yielding an order based on line numbers $\text{line}(h_i) < \text{line}(h_j) \implies h_i \to h_j$.
3. **Import Order:** If h_i involves a module import and h_j references it, we establish $h_i \to h_j$ based on the dependency.
4. **Lint Errors:** Procedurally sample across a set of error-free edits $E = \{e_1, e_2, \ldots, e_n\}$ to iteratively construct a valid program $P = \{p_1, p_2, \ldots, p_m\}$
5. **Identical Code:** Merge each equivalence class $[e] = \{e_i \in \mathcal{E} \mid e_i \sim e\}$ into a single representative edit to minimize redundancy and reduce conflicts.

Example: Edit Priority

Rule: Deletions precede additions.

```
    # Original Code
    def compute(a, b):
        return a + b

    # Edit 1: Delete function
    - def compute(a, b):
    -     return a + b

    # Edit 2: Add a new implementation
    + def compute(a, b):
    +     return a * b  # Multiplication instead of addition
```

Dependency: The deletion must occur before the addition to prevent conflicts.

Static Analysis Approach

Key Idea: Analyze source code structure to derive dependencies.

1. **AST Generation:** Extract syntax structure.
2. **Edit Position & Identifier Extraction:** Track function/variable modifications.
3. **LSP Dependency Tracing:** Use cross-references for inter-file dependencies.
4. **Code Clone Detection:** Identify similar edits.

Example: Static Analysis

Original Code:

```
def add(a, b):
    return a + b
```

AST Representation:

```
FunctionDef
   arguments
   Return
       BinOp
           Name (a)
           Add
           Name (b)
```

Edit position, and identifiers extracted

```
Function: add, Line: 2
Identifier: result, Line: 3, Column: 4
Identifier: a, Line: 3, Column: 13
Identifier: b, Line: 3, Column: 17
Identifier: result, Line: 4, Column: 11
```

LSP: If the function signature is modified, all references to ‘add()‘ must be updated.

Agentic Approach

Key Idea: Use a language model to infer edit relationships.

- **Context Vector:** Capture relevant usage patterns.

$$\mathbf{c}(h_i) = \{\text{usage}(h_i), \text{dependencies}(h_i)\}$$

- **Dependency Scoring:** Rank edit relationships.
- **Multi-Agent Framework**[1]**:** Assign specialized agents to different tasks.

Models Used: GPT-4, Gemini-Flash, DeepSeek v3, Llama.

[1] We will be using the Chain-of-Agents framework

Why Context Selection Matters?

Challenge: Limited context window in LLMs prevents full repository input.

Solution: A systematic process to select the most relevant code snippets and examples.

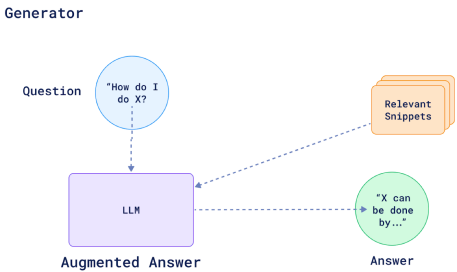


Figure: Retrieval Augmented Generation (RAG) for prompt construction – Input Reduction.

Input Space

Components:

- **Repository (R)**: Contains files $F = \{f_1, f_2, \ldots, f_n\}$.
- **External Vector Database (V)**: Stores examples $E = \{e_1, e_2, \ldots, e_m\}$.

Goal: Extract only the most relevant snippets for an efficient prompt.

Relevance Selection Algorithm

How do we pick relevant examples?

Define a function $\text{rel}(x, q)$ that evaluates the relevance of snippet x to query q:

$$E_{\text{rel}} = \{e \in E \mid \text{rel}(e, q) > \tau_e\}$$

where τ_e are predefined relevance thresholds.

Methods Tested:

- Mean Squared Error (MSE)
- Mean Absolute Error (MAE)
- Cosine Similarity with vector embeddings

Vector Representation

How do we represent edit relationships?

Each edit pair (h_i, h_j) is mapped to a d-dimensional feature vector:

$$\phi(h_i, h_j) = \left[f_1(h_i, h_j), f_2(h_i, h_j), \ldots, f_d(h_i, h_j)\right]$$

Why not code embeddings?

- Code embeddings prioritize syntactic similarity.
- We need **semantic and structural relevance**.

Feature Definitions

Each feature function $f_k(h_i, h_j)$ captures a relationship:

- **File Similarity:** Do both edits occur in the same file?

$$f_1(h_i, h_j) = \mathbb{1}[\text{file}(h_i) = \text{file}(h_j)]$$

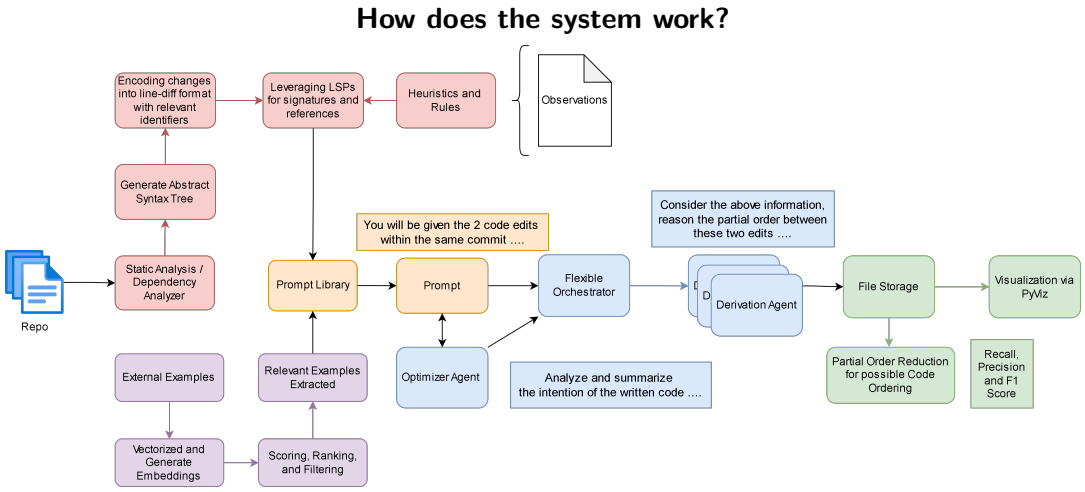
- **Edit Type Matching:** Do they modify the same type of entity?

$$f_2(h_i, h_j) = \mathbb{1}[\text{type}(h_i) = \text{type}(h_j)]$$

- **Lexical Similarity:** Measures textual closeness.

$$f_6(h_i, h_j) = \frac{\text{edit-distance}(h_i, h_j)}{\max(|h_i|, |h_j|)}$$

System Overview



Encoding Code Changes

```
--- src/transformers/commands/run.py
+++ src/transformers/commands/run.py
```

```
@@61,3 61,1 @@ class RunCommand(BaseTransformersCLICommand):  # Line Numbers
>>>> def register_subcommand(parser: ArgumentParser)  # Logical Path

@staticmethod
def register_subcommand(parser: ArgumentParser):
    run_parser = parser.add_parser("run", help="Run a pipeline through the CLI")
-   run_parser.add_argument(
-       "--task", choices=list(SUPPORTED_TASKS.keys()) + list(TTASK_ALIASES.keys()), help="Task
↪   to run"
-   )
+   run_parser.add_argument("--task", choices=get_supported_tasks(), help="Task to run")
```

Annotations:

- **File Path:** `src/transformers/commands/run.py` (Highlighted in red).
- **Line Numbers:** `@@61,3 61,1@@` (Highlighted in blue).
- **Logical Path:** `class RunCommand` and function `register_subcommand()` (Highlighted in blue).

Multi-Agent System Overview

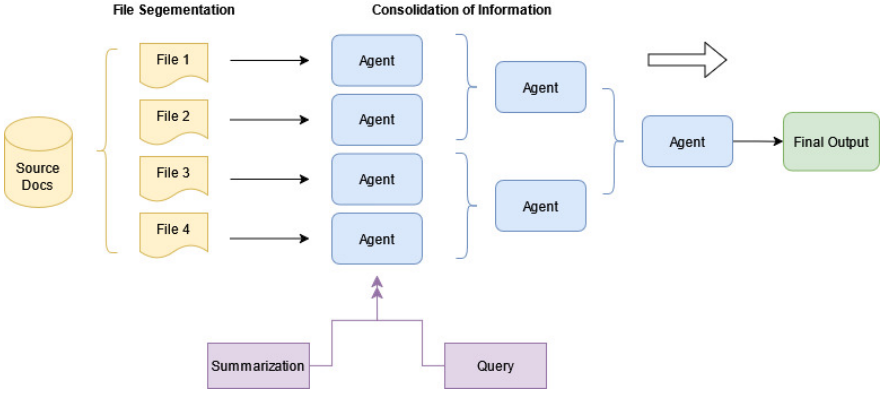


Figure: Agentic Workflow

Consensus Mechanism

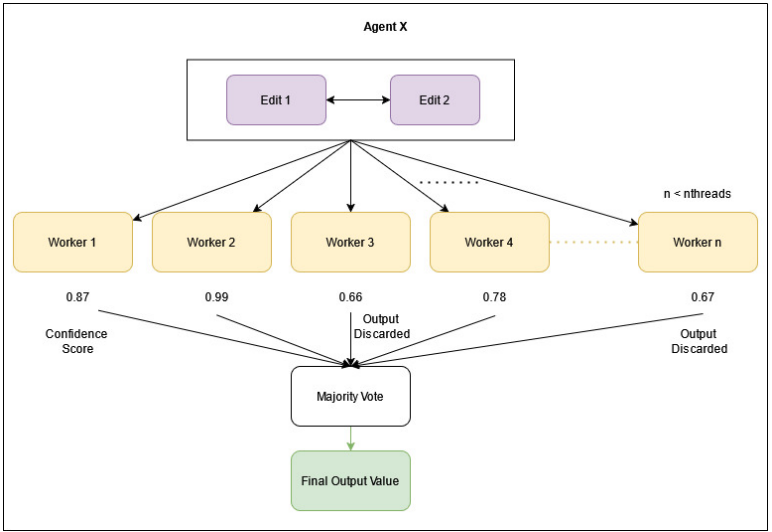
Why is consensus needed?

- Each agent has multiple workers.
- Different workers may suggest conflicting edit orders.
- We need a structured way to merge results.

Methods Used:

- **Voting-Based Resolution** - Majority vote for edit order.
- **Confidence-Weighted Merging** - Prioritize high-confidence relations.
- **Cycle Detection** - Ensure edit sequences remain valid.

Voting Example for Conflict Resolution



Dataset Construction

- 50 commits sourced from popular Python repositories on GitHub.
- Chosen to reflect diverse real-world development tasks.

Attribute	Value
Total Commits	50
Total Edits	364
Total Partial Orders	289
Avg. Edits per Commit	7.28
Avg. Lines Deleted per Edit	1.03
Avg. Lines Inserted per Edit	2.17

Sampling Strategy

To create a representative dataset, each commit C_i is scored by:

- **Edit Type Distribution**: Feature additions, bug fixes, refactoring, etc.
- **File Coverage**: Preference for multi-file commits.
- **Structural Impact**: Significant logic-altering edits prioritized.

Commits selected if their score $S(C_i)$ meets threshold θ:

$$S(C_i) \geq \theta$$

Ensures balanced diversity and relevance.

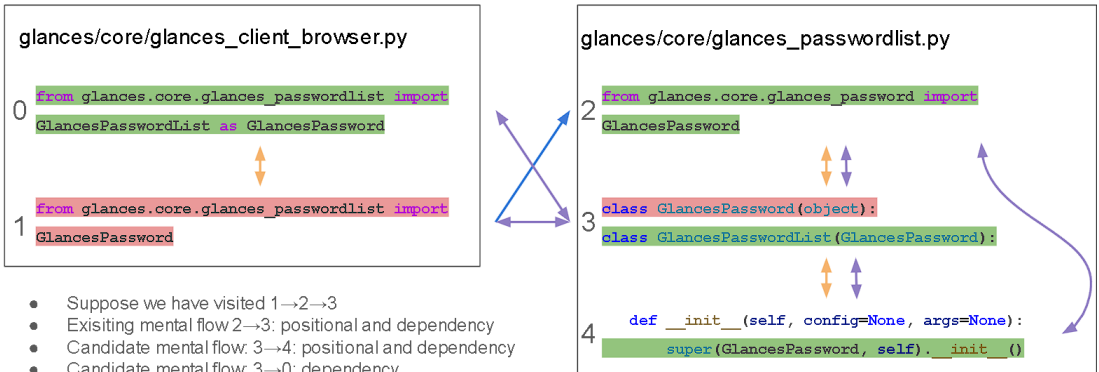
Reconstructing Edit Sequences

Objective

Reconstruct developer intent via edit sequences:

- Pairwise dependencies form Directed Acyclic Graphs (DAGs).
- Edits ordered respecting identified dependencies (partial orders).

Intuition



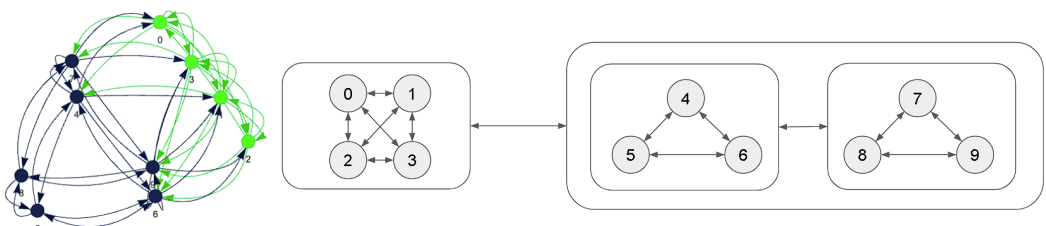
- Suppose we have visited 1→2→3
- Exisiting mental flow 2→3: positional and dependency
- Candidate mental flow: 3→4: positional and dependency
- Candidate mental flow: 3→0: dependency
-

$$P(3 \to 4 | 1 \to 2 \to 3) = \frac{Sim(2 \to 3, 3 \to 4)}{\Sigma_i Sim(2 \to 3, 3 \to i)} > P(3 \to 0 | 1 \to 2 \to 3)$$

Constructing Equivalence Classes

Two edits should be in an equivalence class if they:

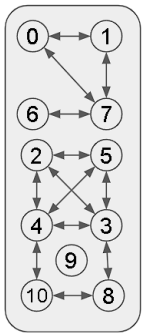
- Are clones of one another in the same/different file
- Share same context
  - Within the same scope and logical path
  - Has bi-directional dependency
  - Import-use / def-use relationship

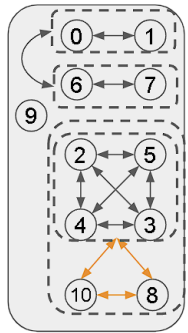


Hierarchical Graph Statistics

- Average number of levels 1.76
- Average edge reduction rate from converting to hierarchical graph: 16.99%
- Average number of nodes in each equivalence class: 2.74

26 edges





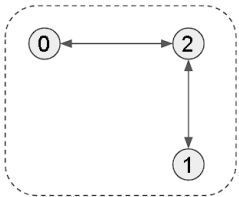
- 2 levels
- 24 edges
- Avg. 3.6 node in cluster

Does it make sense?

- **Input:** Hierarchical edit partial order graph, Ground-truth edit sequence
- **Recall:** How many ground-truth edit orders do not violate the hierarchical graph?
- **Derived restrictions:**
  - The subsequent edit node cannot be an upper-level node unless all nodes at the current level have been visited.
    (e.g., $2 \rightarrow 8$ is forbidden when 3/4/5 is unvisited)
  - The subsequent edit node must be connected to the current node, unless all other nodes at the current level have been visited.
    (e.g., $0 \rightarrow 9$ is allowed while 0/1/6/7 is visited)
- **Results:** 96.41% (25 commits)

Failed Cases

- The partial order graph has false negative edges: missing edge 0 $\iff$ 1



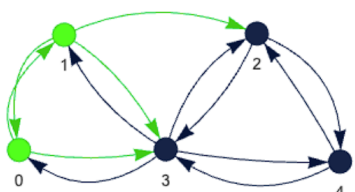

```
41  41          library_paths: Dict[str, List[str]] = {}
42  42          ldpaths: List[str] = []
43  43          ccver: Tuple[int, int]
0   44      +   vcs_rev: str = ''
44  45
45  46          # glfw stuff
46  47          all_headers: List[str] = []
        @@ -52,11 +53,13 @@
52  53
53  54          def __init__(
54  55              self, cc: List[str] = [], cppflags: List[str] = [], cflags: List[str] = [], ldflags: List[str] = [],
55      -           library_paths: Dict[str, List[str]] = {}, ldpaths: Optional[List[str]] = None, ccver: Tuple[int, int] =
    (0, 0)
1   56      +       library_paths: Dict[str, List[str]] = {}, ldpaths: Optional[List[str]] = None, ccver: Tuple[int, int] =
    (0, 0),
    57      +       vcs_rev: str = ''
56  58          ):
57  59              self.cc, self.cppflags, self.cflags, self.ldflags, self.library_paths = cc, cppflags, cflags, ldflags,
    library_paths
58  60              self.ldpaths = ldpaths or []
59  61              self.ccver = ccver
2   62      +       self.vcs_rev = vcs_rev
60  63
61  64          def copy(self) -> 'Env':
```

Constrained Ordering Algorithm

Use Depth-First Traversal with Derived Restrictions. Define $\Pi(\mathcal{S})$ to be an ordered set containing the equivalence classes in the correct order. Then, the constrained group ordering is:

$$\mathcal{P} = \bigcup_{\pi \in \Pi(\mathcal{S})} \text{Flatten}(\pi)$$

Output: A Sequence of Edit Order



→ [1,2,4,3,0]

Evaluation Metrics

Given predicted sequence P and ground truth $\mathbf{C}$:

- **Perfect Match**: Exact matching sequence.
- **Minimum Swaps**: Minimal swaps needed to reach valid ground-truth order.
- **Edit and Kendall Tau Distances**: Measures sequence alignment.

Evaluation metrics reported:

- Precision, Recall, F1 Score
- Swap counts and distances

Results Summary

Metric	Score
Precision	0.84
Recall	0.92
F1 Score	0.87
Avg Swaps	2.2 (Median: 2, Max: 6)
Avg Edit Distance	6
Avg Kendall Tau Distance	10
Perfect Matches	82%

Table: Performance Metrics

False Positives

- Tend to treat similar keywords as dependency

```
430  431
431       -   return Env(cc, cppflags, cflags, ldflags, library_paths, ccver=ccver, ldpaths=ldpaths)
     432  +   return Env(cc, cppflags, cflags, ldflags, library_paths, ccver=ccver, ldpaths=ldpaths, vcs_rev=vcs_rev)
432  433

910  913       cmd = [go, 'build', '-v']
911       -   ld_flags = [f"-X 'kitty.VCSRevision={get_vcs_rev_define()}'"]
     914  +   vcs_rev = args.vcs_rev or get_vcs_rev()
     915  +   ld_flags = [f"-X 'kitty.VCSRevision={vcs_rev}'"]
912  916       if for_freeze:
```

False Positives

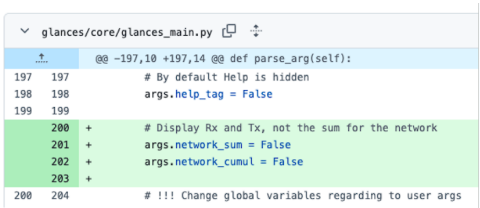
- Contain logic jump
  - *import json* $\rightarrow$ *to_json* $\rightarrow$ *json*

```
1    1      from functools import partial
     2    + import json
2    3
```

```
832  845        table_meta["select_data_query"],
833       -     table_meta["training_options"],
     846  +     to_json(table_meta["training_options"]),
834  847        table_meta["label"],
```

False Negatives

- Existing methods: only when 2 edit hunks within same file, or share dependency, will be sent to llm.
- Here 2 edit hunks share a dependency that cannot be identified by LSP, hence no partial order detected.



glances/core/glances_main.py

```
@@ -197,10 +197,14 @@ def parse_arg(self):
197  197          # By default Help is hidden
198  198          args.help_tag = False
199  199
     200  +       # Display Rx and Tx, not the sum for the network
     201  +       args.network_sum = False
     202  +       args.network_cumul = False
     203  +
200  204          # !!! Change global variables regarding to user args
```

glances/outputs/glances_curses.py

```
@@ -217,11 +217,11 @@ def __catchKey(self):
217  217              # 's' > Show/hide sensors stats (Linux-only)
218  218              self.args.disable_sensors = not self.args.disable_sensors
219  219          elif self.pressedkey == ord('t'):
220       -          # 't' > View network traffic as combination
221       -          self.network_stats_combined = not self.network_stats_combined
     220  +          # 't' > View network traffic as sum Rx+Tx
     221  +          self.args.network_sum = not self.args.network_sum
222  222          elif self.pressedkey == ord('u'):
```

Limitations of the Approach

Despite promising results, several limitations exist:

1. **Static and Heuristic-Based Analysis**
   - Does not fully capture implicit dependencies from runtime or dynamic behaviors.
2. **Metric Limitations**
   - Potential underestimation of generalization due to incomplete ground-truth coverage.
3. **Dataset Coverage:**
   - May not generalize fully across different programming languages or paradigms.

Summary of Contributions

- Formalized a framework for reconstructing developers' code edit sequences.
- Developed a multi-agent orchestration system to infer logical ordering of edits.
- Demonstrated the practicality and effectiveness using real-world data from GitHub.

Thank You!

Q&A