SWE-Ext: Extending and Scaling Augmented Data for Mid-Training of Repository-Level Coding Tasks

Anonymous ACL submission

Abstract

Repository-level benchmarks such as SWE-Bench have highlighted the challenges of scaling language models to complex software engineering tasks. However, the open-sourced real data with verifiable environments from which we can get trajectories to perform rejection fine-tuning, is scarce. In such a situation, we propose training by augmented data as a mid-training stage. Previous augmented data primarily focused on monolingual issue resolving and feature implementation. In this work, we introduce **SWE-Ext**, an effort to scale and extend augmented data for repository-level coding tasks. SWE-Ext broadens existing data along two key dimensions: **multilingual coverage** (spanning 10 languages) and an **auxiliary code completion task**. We uncover distinct transfer mechanisms: data from other programming languages provides transferable signals that generally enhance localization and editing capabilities in single-language (Python) settings, while code completion data strengthens code editing capabilities, particularly for feature implementation tasks requiring substantial new code generation. Augmented data and these extensions yield consistent improvements on Python repository-level benchmarks like SWE-Bench and FEA-Bench, and assist post-training performance on them. Our method offers a simple yet effective way to leverage more open-source data for advancing repository-level code models.

1 Introduction

Repository-level coding tasks have emerged as a critical frontier in code generation, shifting focus from isolated script-level challenges to comprehensive software engineering workflows in full repositories. Early benchmarks emphasized standalone function synthesis, such as HumanEval (Chen et al., 2021) for Python problems and MBPP (Austin et al., 2021) for basic algorithms. However, these evaluations often ignored broader repository contexts, including dependencies, multi-file interactions, and real-world specifications (Xu et al., 2022). Recent advancements have introduced more holistic benchmarks, with SWE-Bench (Jimenez et al., 2024) exemplifying repository-level issue resolution by evaluating models on GitHub pull requests (PRs) tied to verifiable test outcomes. Complementary efforts like FEA-Bench (Li et al., 2025) target feature implementation.

To enhance model capabilities on these demanding tasks, substantial work has explored training strategies, which mainly fall into two categories: **augmented data** for direct **supervised fine-tuning (SFT)** and **verified data** for **rejection fine-tuning (RFT)**. Augmented data methods construct training instances directly from real-world GitHub PRs, leveraging ground-truth information without requiring execution environments (Xie et al., 2025; Wang et al., 2025). For instance, SWE-Fixer (Xie et al., 2025) curates PR-based data for instruction tuning, while MCTS-Refine (Wang et al., 2025) synthesizes reasoning chains via search algorithms. In contrast, verified data approaches build executable environments to collect agent trajectories based on rejection sampling, filtering for those that pass unit tests or achieve successful outcomes (Jain et al., 2025; Pan et al., 2024; Yang et al., 2025b). Techniques like R2E (Jain et al., 2025) and SWE-Gym (Pan et al., 2024) generate validated trajectories in sandboxes for real PRs, and SWE-Smith (Yang et al., 2025b) extends synthetic tasks with verification. While verified data ensures high-quality supervision through execution feedback, it demands significant resources for environment setup and scaling. Augmented data, conversely, offers greater flexibility and avoids these complexities, enabling broader exploration of data domains. So, we propose training using augmented data as an individual mid-training stage.

Despite their advantages, existing augmented

datasets remain limited in scope, predominantly focusing on Python repositories and monolingual issue resolution or feature implementation (Xie et al., 2025). This narrow focus underutilizes the vast diversity of open-source GitHub data, restricting model generalization across languages and task types. In this work, we introduce **SWE-Ext**, a scalable pipeline to extend and augment repository-level coding data along two orthogonal dimensions: multilingual coverage and auxiliary task inclusion (e.g., code completion). Starting from high-quality PRs crawled from GitHub Archive [1] across ten programming languages (Python, Go, JavaScript, Ruby, PHP, Java, TypeScript, C#, C++, and C), we construct datasets of four complementary subtasks: (1) file localization to identify relevant files, (2) component localization to pinpoint functions or methods, (3) code editing to generate patches for code repositories, and (4) code completion derived from short-description PRs to provide code in-filling on fixed-positions. Our contributions are as follows:

- We propose augmented data for mid-training of repository-level coding tasks and develop a scalable data extension pipeline that broadens the utility of GitHub PRs, enabling extension across multilingual and completion-based domains while preserving baseline effectiveness.
- We demonstrate consistent performance gains on repository-level benchmarks like SWE-Bench and FEA-Bench, with multilingual and completion extensions yielding up to +1.4% and +2.5% improvements on mid-training of 32B models, respectively, and up to +5.4% on 7B models. The mid-training stage can raise performance by up to +5.2% after post-training.
- We provide preliminary validation of strong cross-task and cross-language transfer effects, showing that multilingual or completion data can boost single-language downstream performance and highlighting the value of diverse augmentation for generalization.

2 Problem Definition

Notation. Let $\mathcal{R}$ denote a software repository represented as a finite collection of files $\{f_i\}_{i=1}^{N}$ together with auxiliary structures (ASTs, file-level skeletons, import graph, etc.). Let q denote a problem statement (e.g., a PR description, an issue body, a failing test, or a natural-language specification). Let $\mathcal{L}$ be the universe of addressable program positions (file identifiers $\times$ component spans). A *localization* is a subset $\ell \subseteq \mathcal{L}$. The space of edits (patches) is denoted $\mathcal{P}$; an edit (patch) $\delta \in \mathcal{P}$ is an operator $p : \mathcal{R} \to \mathcal{R}'$ that applies insert/replace/delete/move operations to positions in $\mathcal{P}$. Finally, $L(\mathcal{R}', q)$ is a task loss measuring how well the edited repository $\mathcal{R}'$ satisfies q. Note that $P(\cdot)$ denotes a probability distribution.

Repository-level coding task. Given $(\mathcal{R}, q)$, the objective of a repository-level coding agent is to produce a patch $\delta \in \mathcal{P}$ that minimizes the task loss:

$$\hat{\delta} \;=\; \arg\min_{\delta \in \mathcal{P}} L\big(\delta(\mathcal{R}),\, q\big), \tag{1}$$

or equivalently, under a probabilistic modeling view,

$$\hat{\delta} \;=\; \arg\max_{\delta \in \mathcal{P}} P\big(\delta \mid \mathcal{R}, q\big). \tag{2}$$

Decomposition into localization and editing. Because modern LLMs cannot feasibly consume the entirety of large repositories at once, we decompose the posterior over patches by marginalizing over possible localizations:

$$P(\delta \mid \mathcal{R}, q) \;=\; \sum_{\ell \in \mathcal{L}} P(\ell \mid \mathcal{R}, q)\, P(\delta \mid \mathcal{R}, q, \ell). \tag{3}$$

This decomposition separates (i) a *localization* model that identifies candidate regions ℓ of interest, and (ii) an *editing* model that produces a concrete patch conditional on the selected regions. In practice, localization and editing are performed in separate iterative phases: the system first iterates to identify a sufficiently accurate set of relevant files or components, and only then initiates an editing phase where patches can be generated, potentially with feedback.

Code completion as a special case. Code completion corresponds to the special case where the localization $\ell^\star$ is given *a priori* (a fixed position or contiguous span) and the patch is constrained to be a continuation or replacement at that position:

$$\hat{\delta} \;=\; \arg\max_{\delta \in \mathcal{P}} P\big(\delta \mid \mathcal{R}, q, \ell^\star\big). \tag{4}$$

[1] https://www.gharchive.org

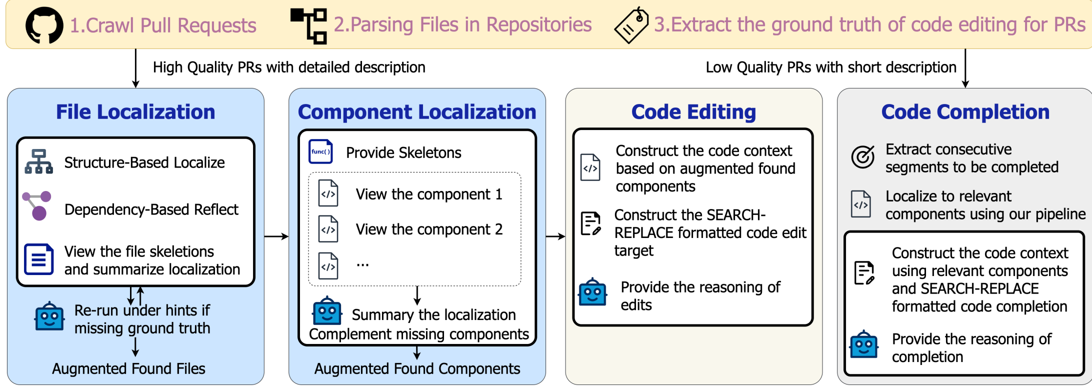

Figure 1: Data collection pipeline for 4 types of tasks in SWE-Ext. We crawl and process pull requests from GH Archive, and then process well-described PRs for the localization-editing task and process other PRs for the completion task.

172 173 ## 3 SWE-Ext: Extend and Scale the Repository-Level Coding Data

174 ### 3.1 Data Collection

175 **Source data and ground truth.** We crawl pull
176 requests (PRs) from GH Archive (recent 10 years)
177 and retain only high-quality candidate samples. We
178 keep PRs whose repository has both (i) star count
179 > 100 and (ii) total PR count > 100, and we ex-
180 clude PRs that modify only non-code files or that
181 were not merged. For each retained PR we col-
182 lect metadata (PR description, linked issue text
183 if any, timestamps) and patch (diff). We parse
184 repository files with tree-sitter parsers for the fol-
185 lowing languages: Python, Go, JavaScript, Ruby,
186 PHP, Java, TypeScript, C#, C++, and C. From each
187 file we extract a compact *file skeleton* (top-level
188 classes, functions/methods). Using the patch, we
189 record the ground-truth modified files $\mathcal{F}_{\text{gt}}$ and the
190 ground-truth modified components $\mathcal{C}_{\text{gt}}$. To limit
191 tractability, we retain only PRs with $||\mathcal{F}_{\text{gt}}|| \leq 5$
192 and $||\mathcal{C}_{\text{gt}}|| \leq 10$. The problem statement q is the
193 concatenation of the PR description and the linked
194 issue body (when available). Formally, each re-
195 tained example is stored as a tuple

196
$$s = (\mathcal{R},\, q,\, \mathcal{F}_{\text{gt}},\, \mathcal{C}_{\text{gt}},\, \delta_{\text{gt}}), \tag{5}$$

197 where δ_{gt} is the ground-truth patch.

198 **File localization.** To approximate $P(\ell \mid \mathcal{R}, q)$
199 at file granularity, we construct file-localization
200 data. We employ a three-stage pipeline driven
201 by an expert model: (i) rank files according to
202 the file tree of the problem statement; (ii) expand
203 via file-level dependency to form a candidate set;
204 and (iii) re-rank using file skeletons and the expert
205 model to produce a final top-n set $\mathcal{S}$. If $\mathcal{F}_{\text{gt}} \not\subseteq \mathcal{S}$,
206 we re-run the expert with rationalization hints so
207 that S always covers the ground truth. The result-
208 ing dataset pairs $(\mathcal{R}, q, \mathcal{S})$ provide supervision for
209 learning to predict high-recall file sets that con-
210 tain the true patch locations, thereby reducing the
211 effective search space for downstream editing.

212 **Component localization.** Once file localization
213 is complete, we build component-level (function/-
214 class level) localization data to approximate $P(c \mid$
215 $\mathcal{R}, q, \ell)$. We adapt iterative selection methods from
216 CoSIL (Jiang et al., 2025) to allow the expert model
217 to sequentially inspect and select components from
218 candidate files. The process terminates once up to
219 $K = 10$ components are selected. If some ground-
220 truth components are missing, they are inserted into
221 the iterative process and final list. Each example
222 $(\mathcal{R}, q, \mathcal{F}, \mathcal{S}_{\text{comp}})$ thus can serve as a high-recall la-
223 bel for a policy that selects components relevant
224 to q. This supports models in learning to identify
225 fine-grained patch locations and to maximize recall
226 under constrained inspection budgets.

227 **Code editing.** We construct code-editing data di-
228 rectly from δ_{gt}. For each selected localization
229 context $\ell_{\text{ctx}} = \mathcal{S}_{\text{comp}}$, we convert patches into
230 *before/after* code fragments (search–replace pairs
231 δ_{sp}). To enrich supervision, an expert model gen-
232 erates a rationale explaining the patch. Each ex-
233 ample $(\mathcal{R}, q, \ell_{\text{ctx}}, \delta_{sp})$ provides supervision for
234 $P(\delta \mid \mathcal{R}, q, \ell_{\text{ctx}})$. This partition trains models to
235 propose concrete patches given a localized context.

236 **Code completion.** Finally, we derive a comple-
237 tion dataset from PRs whose problem statements
238 are short (fewer than 50 words), which are possibly
239 not complete instructions for code editing. In these

Table 1: Statistics of the dialogue data for 4 tasks in the SWE-Ext dataset. Token counts are computed using `Qwen2.5-Coder`. **Dialogue Len** represents the total token count of the full dialogue after applying a chat template. **Answer Len** is the average token count of all "assistant" responses within the dialogue. The 'Answer Len' are the final macro-averaged values. The **Patch** metric shows the average number of lines changed in the ground truth patches. For **Localizations**, the values indicate the number of ground truth modified locations versus the total number of locations in the augmented data.

Dataset	Unique repos	Samples (#)	Dialogue Len (Avg tok.)	Answer Len (Avg tok.)	Patch (Avg lines)	Localizations (Avg GT/Total)
File-Localization	4 746	56 384	9 787.8	552.1	-	1.66/4.83
Component-Localization	4 702	56 726	8 164.3	59.8	-	2.55/6.32
Code-Editing	4 319	45 858	6 447.7	955.0	32.0	-
Code-Completion	1 642	23 853	5 929.7	430.1	88.0	-

240 cases, we extract newly added functions or contiguous inserted regions and frame them as completion tasks with a fixed localization $\ell^\star$. Each example provides supervision for $P(\delta \mid \mathcal{R}, q, \ell^\star)$. Because completions originate from real patches and are abundant, they supply local-generation signals that improve fluency and act as an effective curriculum for training editing models. 247

248 **Summary.** The construction pipeline yields four complementary datasets, each aligned with a distinct conditional distribution in the factorization of $P(\delta \mid \mathcal{R}, q)$. By supervising different sub-tasks separately, we enable mid-training for repository-level tasks while ensuring that the final system can integrate these capabilities into a single model. For further details, please refer to Appendix D. 255

256 ### 3.2 Dataset Characteristics

257 The SWE-Ext corpus is a *mid-training* dataset constructed from real-world pull requests for repository-level coding tasks. The dataset is organized into four complementary tasks: file localization, component localization, code editing, and code completion. Table 1 reports core statistics. Below, we summarize the most important characteristics and how they relate to model training. 264

265 **Origin and construction.** Table 1 presents the statistics of the SWE-Ext dataset. For the three localization and editing tasks (File-Localization, Component-Localization, and Code-Editing), we retain a canonical pool of approximately 56k to 46k PRs. Crucially, these tasks exhibit high data diversity, being sourced from 4,746 unique GitHub repositories, which strongly demonstrates the broad applicability of our data augmentation strategy. The Code-Completion task collects newly-added functions or contiguous insertion ranges from $\sim$ 23.9k PRs, whose short descriptions make end-to-end supervision weak but whose insertions are realistic completion targets. 278

279 **Augmentation of localizations.** The **Localizations** column in Table 1 highlights our data augmentation strategy: The large candidate set size (e.g., 4.83 files vs. 1.66 ground truth files for File-Localization) confirms that we incorporate relevant, non-ground-truth files or components into the training data. This design is crucial because it makes the dataset more realistic, aligning with scenarios where agents' localization is often relevant but not perfectly precise in real-world applications. 288

289 **Scale, context length, and edit complexity.** The dataset provides long, realistic contexts: average input contexts (**Dialogue Len**) range from ~ 5.9k tokens (completion) up to $\sim$ 9.8k tokens (file-localization), reflecting the long context challenge for real-world repository-level tasks. Average **Answer Len** varies by task ($\approx$ 60–955 tokens). Edit complexity differs across modalities: code-editing patches average $\sim$ 32 lines (explicit before/after pairs), while completion targets are larger on average ($\sim$ 88 lines) because they often correspond to newly-added contiguous code regions. 300

301 **Language coverage.** SWE-Ext covers ten languages: Python, Go, JavaScript, Ruby, PHP, TypeScript, Java, C#, C++, and C. Language distributions (See Figure 3) show consistent dominance of Python and Go across modalities (roughly 27–30% and 15–21% respectively), moderate representation for JavaScript, Ruby and PHP ($\approx$ 9–13%), and smaller but present contributions from Java/TypeScript/C#. Low-resource languages (C, C++) appear at the long tail ($<$ 1–1.1% in most partitions). This multi-language composition enables comprehensive utilization of the available data while reflecting real-world repository populations. 313

4 Experiments

Training. To empirically validate whether our augmented data, derived from real-world GitHub PRs, can enhance the proficiency of large language models on complex repository-level coding tasks, and to explore the synergistic effects of incorporating multilingual and code completion data, we employ a supervised finetuning approach tailored for a multi-turn dialogue format to perform supervised fine-tuning.

Models. Our methodology leverages GPT-4o (`gpt-4o-2024-05-13`) (Hurst et al., 2024) as a data augmentation expert to transform raw GitHub PR data into a multi-task dataset, as detailed in Section 3.1. For the training phase, we utilize `Qwen2.5-Coder-Instruct` (Hui et al., 2024) as our foundational model. For details of training and models, please refer to Appendix B.

Agent System. Our agent system adopts a staged approach. We first perform file and component localization following the same iterative process used for data generation. Subsequent stages, including line-level localization, patch generation, and verification, are executed in an Agentless manner (Xia et al., 2024). We have termed this pipeline **CosAgentless**, signifying the integration of the iterative and fine-grained localization process from CoSIL (Xia et al., 2024) into the standard Agentless inference pipeline.

Mid-Training Data. For a comprehensive ablation study and a fair comparison with prior works, we define three distinct training configurations. First, the **SWE-Ext-Baseline** model is trained on the standard Python-only data for the first three tasks (localization and editing), consistent with existing agent systems. Second, the **SWE-Ext-Multilingual** model is trained on the first three tasks of our multilingual data. Finally, the **SWE-Ext-Completion** model is trained on Python-only data but across all four tasks. This setup enables a clear analysis of the performance gains derived from extending our data along both multilingual and task-specific dimensions.

Post-Training Data. To validate that our mid-training recipe based on augmented data can also boost post-training performance, we use mainly `DeepSeek-V3.2`(Liu et al., 2025) and `Claude-4.5-Sonnet`(Anthropic, 2025c) to perform reject sampling on a verifiable dataset SWE-Gym (Pan et al., 2024). An adjusted OpenHands that uses command-line and string replace tools is utilized as the scaffold. Finally, we get 3,128 OpenHands trajectories as a standard post-training dataset to perform rejection fine-tuning (RFT).

Evaluation. We evaluate our models on two distinct repository-level coding benchmarks: the well-established SWE-bench (Jimenez et al., 2024) for resolving issues and FEA-Bench (Li et al., 2025) for implementing features. Both benchmarks only contain Python task instances and the primary metric is the task pass rate (%resolved). For a more granular analysis, we also report the Top-x hit rates, MRR (Mean Reciprocal Rank), and MPP (Mean Precision at Position)(Manning, 2008) for the file and component localization stages. In our evaluation, we use a modified Agentless framework to perform a single complete attempt per task. A task is considered a failure if the process stalls or errors out, resulting in an empty patch. For the evaluation after RFT by OpenHands trajectories, we use OpenHands agent with maximum rounds of 100.

5 Results

5.1 Software Engineering Benchmarks

Competitive performance as a mid-training method. According to Table 2, although the absolute performance of SWE-Ext models is lower than the strongest systems, our models achieve competitive results among open-weight models even with the baseline augmented data. Most higher-performing models either (i) use significantly larger parameter counts or (ii) are distilled from expert models with higher success rates. In contrast, SWE-Ext only leverages augmented training data without relying on such privileged resources, yet already achieves certain improvements. The results validate our pipeline of data collection.

Consistent improvements with data extension. Table 2 and Table 3 demonstrate that extending augmented data along different dimensions yields consistent gains during mid-training. On SWE-Bench Verified, the multilingual extension improves performance from 31.2% to 32.6% ($+1.4\%$), while the completion extension reaches 32.2%. On FEA-Bench Lite, similar trends are observed: the model trained with extended data outperforms the baseline by up to $+2.5\%$. These results confirm that the proposed data extensions are effective in enhancing model performance.

Model	System	Expert Model	Exec	Recipe	% Resolved
Closed Weight Models					
GPT-4o (Hurst et al., 2024)	Agentless	-	-	-	38.8%
Claude 3.5 Sonnet (Anthropic, 2024)	Agentless	-	-	-	50.8%
Claude 3.7 Sonnet (Anthropic, 2025a)	SWE-agent	-	-	-	58.2%
Claude 4 Sonnet (Anthropic, 2025b)	SWE-agent	-	-	-	72.7%
Claude 4.5 Sonnet (Anthropic, 2025c)	-	-	-	-	**77.2%**
Llama3-SWE-RL-70B (Wei et al., 2025)	Agentless	-	-	RL	41.0%
Open Weight Models					
DeepSeek-V3-671B (Liu et al., 2024)	Agentless	-	-	-	42.0%
DeepSeek-V3.2-685B (Liu et al., 2025)	-	-	-	-	**73.1%**
Kimi K2-1TB (Team et al., 2025)	Agentless	-	-	-	65.8%
Lingma-SWE-GPT-72B (Ma et al., 2024)	SWE-SynInfer	-	-	-	28.8%
Qwen3-235B-A22B (Yang et al., 2025a)	OpenHands	-	-	-	34.4%
SWE-gym-32B (Pan et al., 2024)	OpenHands	Claude 3.5 Sonnet*	✓	RFT	20.6%
SWE-gym-32B (Reproduce)	OpenHands	DeepSeek-V3.2*	✓	RFT	48.0%
R2E-Gym-32B (Jain et al., 2025)	OpenHands	Claude 3.5 Sonnet	✓	RFT	34.4%
SWE-smith-32B (Yang et al., 2025b)	SWE-agent	Claude 3.7 Sonnet*	✓	RFT	40.2%
SWE-fixer-72B (Xie et al., 2025)	SWE-Fixer	-	✗	SFT	32.8%
SoRFT-Qwen-32B (Ma et al., 2025)	Agentless	Claude 3.5 Sonnet	✗	SFT+RL	30.8%
MCTS-Refine-32B (Wang et al., 2025)	Agentless	DeepSeek-V3	✗	SFT	32.4%
SWE-Ext Models					
SWE-Ext-Baseline-32B	CosAgentless	GPT-4o	✗	SFT	31.2%
SWE-Ext-Multilingual-32B	CosAgentless	GPT-4o	✗	SFT	32.6%
SWE-Ext-Completion-32B	CosAgentless	GPT-4o	✗	SFT	32.2%
Ext-Baseline-32B + SWE-Gym RFT	OpenHands	DeepSeek-V3.2*	✓	SFT+RFT	50.4%
Ext-Multilingual-32B + SWE-Gym RFT	OpenHands	DeepSeek-V3.2*	✓	SFT+RFT	49.6%
Ext-Completion-32B + SWE-Gym RFT	OpenHands	DeepSeek-V3.2*	✓	SFT+RFT	53.2%

Table 2: Resolve rates on **SWE-bench Verified**, collected from (Yang et al., 2025b) and Kimi-K2 (Team et al., 2025) technical reports. **Expert Model** indicates the LLMs that generated content during the data construction process. **Exec** indicates whether verifiable data is used for rejective sampling. All performance are results by single attempt of agent systems. *Indicates the primary data for training is *mainly* generated by the specified expert model.

Model	System	% Resolved
Zero-Short Inference		
GPT-4o (Hurst et al., 2024)	Agentless	9.0%
o1 (Jaech et al., 2024)	Agentless	14.0%
DeepSeek-V3-671B (Liu et al., 2024)	Agentless-Lite	11.0%
SWE-Ext Models		
SWE-Ext-Baseline-32B	CosAgentless	10.0%
SWE-Ext-Multilingual-32B	CosAgentless	11.5%
SWE-Ext-Completion-32B	CosAgentless	12.5%

Table 3: Resolve rates for existing solutions on **FEA-Bench Lite**, collected from (Li et al., 2025). All performance are results by single attempt of agent systems.

Mid-training benefits for downstream post-training. As depicted in Section 4, we report results under a standard rejection fine-tuning (RFT) pipeline to evaluate the quality of mid-training. When trained with the same SWE-Gym setup, models initialized from SWE-EXT mid-training consistently outperform the reproduced baseline, with gains of up to +5.2% (from 48.0% to 53.2%). Performance across different extensions remains broadly comparable after post-training, suggesting that the primary advantage lies in the augmented-data-based mid-training paradigm, which provides a stronger and more transferable foundation for subsequent fine-tuning.

Multilingual vs. Completion extension. From results on two benchmarks, multilingual extension brings more consistent improvements in issue resolving, suggesting that cross-lingual data helps the model capture generalizable reasoning strategies. In contrast, completion extension is particularly effective for feature implementation, where generating new functionality requires stronger code completion capabilities. The results highlight that different types of target complementary aspects of repository-level development jointly contribute to the overall performance gains.

Ablation insights. By comparing the multilingual extension and the completion-based extension, we find that both augmentation strategies yield consistent gains. This suggests that when the amount of original supervision is limited, expanding the dataset in orthogonal directions, either through multilingual variants or through completion-style tasks, can provide complementary signals and enhance model generalization. This result underscores the value of leveraging diverse forms of augmented data for repository-level coding tasks. Overall, these findings highlight that SWE-Ext contributes a practical approach for mid-training to improve the software engineering capabilities of LLMs.

Model	System	Component-level Localization				File-level Localization			
		Hit@1	Hit@3	MAP	MRR	Hit@1	Hit@3	MAP	MRR
Qwen2.5-Coder-32B	CoSIL	43.0	54.3	46.1	48.9	60.7	77.3	69.8	69.4
Qwen2.5-Coder-32B	CosAgentless	47.4	61.8	47.4	54.8	69.0	86.2	76.1	77.7
SWE-Ext-Baseline	CosAgentless	55.0	65.4	53.0	60.8	72.8	87.6	77.0	80.2
SWE-Ext-Multilingual	CosAgentless	**57.2**	**68.2**	**55.3**	**63.1**	**75.8**	**90.2**	**79.4**	**82.8**
SWE-Ext-Completion	CosAgentless	52.8	62.0	51.3	57.9	72.8	86.6	76.7	79.8

Table 4: Localization performance on the **SWE-bench Verified** test set, consolidating Component-level and File-level results. All scores are reported in percentages (%).

5.2 Localization Analysis

To better understand how augmented data improves final resolution rates, we analyze the intermediate step of *localization* on SWE-bench Verified. Localization accuracy determines whether the correct files or components are identified, and thus directly affects downstream success.

Effect of multilingual extension. Introducing multilingual data consistently improves localization: component Hit@1 increases from 55.0% to 57.2%, and file Hit@1 from 72.8% to 75.8% (Table 4). This shows that data from other programming languages provides transferable knowledge that strengthens the model's ability to locate relevant code regions. As a result, localization accuracy and overall resolution both improve, even when the target task is restricted to Python repositories.

Effect of completion extension. In contrast, extending with completion data leads to a different effect: component Hit@1 decreases from 55.0% to 52.8%, while file-level metrics remain nearly unchanged. This is expected, since completion data focuses on the code editing objective rather than localization. Formally, such data trains the model according to Eq. 4, which optimizes the synthesis of code edits δ given a fixed ground-truth location $\ell^\star$, without providing supervision for selecting ℓ. Despite the degradation in localization, the strengthened editing capability yields higher overall resolution rates, demonstrating that the positive impact on code edit outweighs the negative impact on localization.

Implication. These results reveal distinct mechanisms: multilingual extension simultaneously improves localization and editing by enabling transfer across languages, whereas completion extension primarily enhances code editing ability, especially in scenarios that require substantial additions of new content, such as feature implementation.

5.3 Scaling Analysis

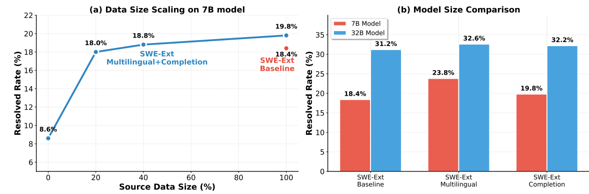

Figure 2: Scaling law analysis on SWE-bench. (a) Performance scaling with training data size follows an approximate logarithmic curve. (b) Model size comparison across three configurations demonstrates consistent improvements from 7B to 32B models.

To validate the effectiveness and robustness of our approach, we conduct experiments across both model sizes and training data volumes. We evaluate our method on `Qwen2.5-Coder-7B-Instruct` and systematically sample 20%, 40%, and 100% of collected data to examine scaling behaviors. The results are presented in Figure 2.

Consistent improvement across different model sizes. Our experiments reveal that although the 32B model substantially outperforms its 7B version, the relative performance trends between different extension strategies remain remarkably stable: multilingual extension consistently provides the largest gains (+5.4% for 7B, +1.4% for 32B over baseline), while completion extension shows modest improvements. This consistency across model scales validates the robustness of our data construction pipeline.

Extension strategies enhance data scaling. Figure 2(a) shows that applying our extension strategies yields clear logarithmic scaling patterns. The largest gains (+9.4%) appear when moving from 0% to 20% of the data, with diminishing returns thereafter. Compared with the SWE-Ext-Baseline, which simply discards non-Python samples or PRs with omitted description and achieves only 18.4% resolution rate, our approach lifts the scaling curve.

By converting previously unusable data into effective training data through multilingual and completion extensions, we improve both efficiency and the scaling potential of augmented data.

6 Related Work

6.1 Repository-level Coding Task

Early benchmarks for code generation focused on function-level tasks, such as HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021) for simple algorithms. These evaluations emphasized isolated synthesis but overlooked repository contexts. Subsequent benchmarks introduced class- and multi-file challenges, including ClassEval (Du et al., 2023), BigCodeBench (Zhuo et al., 2024), and LiveCodeBench (Jain et al., 2024). Evaluation of real problems in code repositories is advanced with SWE-bench (Jimenez et al., 2024) for GitHub issue resolution, alongside variants like DevEval (Li et al., 2024b) and EvoCodeBench (Li et al., 2024a) that align with real repositories. FEA-Bench (Li et al., 2025) targets feature additions via pull requests. Our work aims to enhance the capabilities of large language models in repository-level coding tasks through post-training with augmented real-world data.

6.2 Software Engineering Agents

Although basic code large language models (Hui et al., 2024) can achieve good performance on many code benchmarks through direct generation, for repository-level tasks involving numerous files and complex edits, the common practice is to incorporate agent frameworks (Yao et al., 2023). Agents for repository-level tasks often employ iterative processes for issue resolution. SWE-agent (Yang et al., 2024) uses agent-computer interfaces for navigation and editing, while AutoCodeRover (Zhang et al., 2024) integrates fault localization for repairs. CodePlan (Bairi et al., 2024) focuses on planning modifications. Localization-specific agents include CoSIL (Jiang et al., 2025), and LocAgent (Chen et al., 2025) for multi-hop reasoning via heterogeneous graphs. Agentless (Xia et al., 2024) simplifies to a three-phase process without complex tooling. OpenHands (Wang et al., 2024) uses iterative tool calls to complete a task. Our data construction method draws from the commonalities of these agents, employing a pipeline to build data for four subtasks, thereby enhancing model capabilities in repository-level coding.

6.3 Training Data for Repository-level Coding

Traditional instruction tuning methods enhance general code generation, such as Wizard-Coder (Luo et al., 2023), and WaveCoder (Yu et al., 2024). However, constructing training data for repository-level tasks differs significantly, as it requires building input-output and reasoning processes to simulate complex interactions.

Mainstream approaches involve building executable environments to collect agent trajectories for rejection sampling and supervised fine-tuning. R2E (Jain et al., 2025) and SWE-Gym (Pan et al., 2024) create runtime environments to gather verified trajectories from limited task instances. SWE-Smith (Yang et al., 2025b) extends synthetic data to produce more verifiable tasks under constrained environments. Other methods focus on reinforcement learning. SoRFT (Ma et al., 2025) employs subtask-oriented rejection sampling and PPO.

Due to the diversity requirements of repositories, some works augment or synthesize data without constructing environments. SWE-Fixer (Xie et al., 2025) gathers data with chain-of-thought, and MCTS-Refine (Wang et al., 2025) builds reasoning chains via MCTS. In relation to these efforts, while data construction for repository-level tasks predominantly focuses on Python, the proposed pipeline extends the scope by systematically gathering multilingual data of different subtasks, demonstrating that out-of-distribution data like code completion can further enhance performance.

7 Conclusion

In this work, we proposed augmented data as an explicit *mid-training* stage for repository-level coding tasks and introduced **SWE-Ext**, a scalable pipeline to extend such data beyond existing settings. By broadening the utilization of GitHub data along two orthogonal dimensions, multilingual coverage and an auxiliary code completion task, we expand the scope of usable augmented data without relying on execution environments. Experiments on SWE-Bench and FEA-Bench show consistent gains at the mid-training stage, with different extensions providing complementary benefits, and further indicate that augmented-data-based mid-training offers a stronger initialization for downstream fine-tuning. Our findings highlight mid-training with diverse augmented data as a simple yet effective strategy for improving repository-level code models under realistic data constraints.

Limitations

This work focuses on augmented data as a mid-training stage for repository-level coding tasks. While SWE-Ext demonstrates consistent gains, several limitations remain. First, due to limited API resources, the scale of data synthesis is constrained, and we are unable to exhaustively explore all combinations of augmentation dimensions or expand to larger volumes of augmented data. Second, due to restricted computational resources, our experiments are conducted on a limited set of base models and training configurations, primarily at the 32B scale, and the observed trends may not fully generalize to substantially smaller or larger models. Finally, although we consider multilingual and completion-based extensions, other forms of augmentation, such as refactoring-oriented supervision or test generation, are not explored in this work. We leave systematic investigation of larger-scale data synthesis and broader augmentation strategies to future work.

Ethics Statement

This work uses data derived exclusively from publicly available open-source repositories on GitHub. As such, it does not involve personal or private user data and poses minimal privacy concerns. We respect the original licensing terms of all source repositories and only use content that is publicly accessible.

During data construction, large language models are used to assist with data augmentation and language refinement. While model-generated content may occasionally contain biased, insecure, or incorrect code patterns, the data is restricted to software engineering contexts and does not involve sensitive domains. Models trained on SWE-Ext may still generate code with potential security vulnerabilities or unintended behaviors, which is a general risk for code generation systems. We therefore advise users to carefully review and test generated code before deployment in real-world systems.

Large language models were also used to assist with code completion and language polishing during the preparation of this manuscript. All research ideas, experimental designs, and methodological decisions were independently developed by the authors, who take full responsibility for the content and conclusions presented in this work.

References

Anthropic. 2024. Introducing Claude 3.5 Sonnet.

Anthropic. 2025a. Introducing Claude 3.7 Sonnet.

Anthropic. 2025b. Introducing Claude 4.

Anthropic. 2025c. Introducing Claude 4.5 Sonnet.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and 1 others. 2021. Program synthesis with large language models. *ArXiv preprint*, abs/2108.07732.

Ramakrishna Bairi, Atharv Sonwane, Aditya Kanade, Arun Iyer, Suresh Parthasarathy, Sriram Rajamani, B Ashok, and Shashank Shet. 2024. Codeplan: Repository-level coding using llms and planning. *Proceedings of the ACM on Software Engineering*, 1(FSE):675–698.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, and 1 others. 2021. Evaluating large language models trained on code. *ArXiv preprint*, abs/2107.03374.

Zhaoling Chen, Xiangru Tang, Gangda Deng, Fang Wu, Jialong Wu, Zhiwei Jiang, Viktor Prasanna, Arman Cohan, and Xingyao Wang. 2025. Locagent: Graph-guided llm agents for code localization. *arXiv preprint arXiv:2503.09089*.

Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. 2023. Classeval: A manually-crafted benchmark for evaluating llms on class-level code generation. *ArXiv preprint*, abs/2308.01861.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, and 1 others. 2024. Qwen2. 5-coder technical report. *ArXiv preprint*, abs/2409.12186.

Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, and 1 others. 2024. Gpt-4o system card. *ArXiv preprint*, abs/2410.21276.

Aaron Jaech, Adam Kalai, Adam Lerer, Adam Richardson, Ahmed El-Kishky, Aiden Low, Alec Helyar, Aleksander Madry, Alex Beutel, Alex Carney, and 1 others. 2024. Openai o1 system card. *ArXiv preprint*, abs/2412.16720.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2024. Livecodebench: Holistic and contamination free evaluation of large language models for code. *ArXiv preprint*, abs/2403.07974.

Naman Jain, Jaskirat Singh, Manish Shetty, Liang Zheng, Koushik Sen, and Ion Stoica. 2025. R2e-gym: Procedural environments and hybrid verifiers for scaling open-weights swe agents. *arXiv preprint arXiv:2504.07164*.

Zhonghao Jiang, Xiaoxue Ren, Meng Yan, Wei Jiang, Yong Li, and Zhongxin Liu. 2025. Cosil: Software issue localization via llm-driven code repository graph searching. *arXiv preprint arXiv:2503.22424*.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. 2024. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*.

Jia Li, Ge Li, Xuanming Zhang, Yihong Dong, and Zhi Jin. 2024a. Evocodebench: An evolving code generation benchmark aligned with real-world code repositories. *ArXiv preprint*, abs/2404.00599.

Jia Li, Ge Li, Yunfei Zhao, Yongmin Li, Huanyu Liu, Hao Zhu, Lecheng Wang, Kaibo Liu, Zheng Fang, Lanshen Wang, and 1 others. 2024b. Deveval: A manually-annotated code generation benchmark aligned with real-world code repositories. *ArXiv preprint*, abs/2405.19856.

Wei Li, Xin Zhang, Zhongxin Guo, Shaoguang Mao, Wen Luo, Guangyue Peng, Yangyu Huang, Houfeng Wang, and Scarlett Li. 2025. Fea-bench: A benchmark for evaluating repository-level code generation for feature implementation. *arXiv preprint arXiv:2503.06680*.

Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, and 1 others. 2024. Deepseek-v3 technical report. *ArXiv preprint*, abs/2412.19437.

Aixin Liu, Aoxue Mei, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, and 1 others. 2025. Deepseek-v3. 2: Pushing the frontier of open large language models. *arXiv preprint arXiv:2512.02556*.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. Wizardcoder: Empowering code large language models with evol-instruct. *ArXiv preprint*, abs/2306.08568.

Yingwei Ma, Rongyu Cao, Yongchang Cao, Yue Zhang, Jue Chen, Yibo Liu, Yuchen Liu, Binhua Li, Fei Huang, and Yongbin Li. 2024. Lingma swe-gpt: An open development-process-centric language model for automated software improvement. *arXiv preprint arXiv:2411.00622*.

Zexiong Ma, Chao Peng, Pengfei Gao, Xiangxin Meng, Yanzhen Zou, and Bing Xie. 2025. Sorft: Issue resolving with subtask-oriented reinforced fine-tuning. *arXiv preprint arXiv:2502.20127*.

Christopher D Manning. 2008. *Introduction to information retrieval*. Syngress Publishing,.

Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. 2024. Training software engineering agents and verifiers with swe-gym. *arXiv preprint arXiv:2412.21139*.

Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. 2024. Hybridflow: A flexible and efficient rlhf framework. *arXiv preprint arXiv: 2409.19256*.

Kimi Team, Yifan Bai, Yiping Bao, Guanduo Chen, Jiahao Chen, Ningxin Chen, Ruijue Chen, Yanru Chen, Yuankun Chen, Yutian Chen, and 1 others. 2025. Kimi k2: Open agentic intelligence. *arXiv preprint arXiv:2507.20534*.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, and 1 others. 2024. Openhands: An open platform for ai software developers as generalist agents. *arXiv preprint arXiv:2407.16741*.

Yibo Wang, Zhihao Peng, Ying Wang, Zhao Wei, Hai Yu, and Zhiliang Zhu. 2025. Mcts-refined cot: High-quality fine-tuning data for llm-based repository issue resolution. *arXiv preprint arXiv:2506.12728*.

Yuxiang Wei, Olivier Duchenne, Jade Copet, Quentin Carbonneaux, Lingming Zhang, Daniel Fried, Gabriel Synnaeve, Rishabh Singh, and Sida I Wang. 2025. Swe-rl: Advancing llm reasoning via reinforcement learning on open software evolution. *arXiv preprint arXiv:2502.18449*.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2024. Agentless: Demystifying llm-based software engineering agents. *ArXiv preprint*, abs/2407.01489.

Chengxing Xie, Bowen Li, Chang Gao, He Du, Wai Lam, Difan Zou, and Kai Chen. 2025. Swe-fixer: Training open-source llms for effective and efficient github issue resolution. *arXiv preprint arXiv:2501.05040*.

Frank F. Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. 2022. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, MAPS 2022, page 1–10, New York, NY, USA. Association for Computing Machinery.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, and 1 others. 2025a. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*.

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. Swe-agent: Agent-computer interfaces enable automated software engineering. *ArXiv preprint*, abs/2405.15793.

John Yang, Kilian Lieret, Carlos E Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. 2025b. Swe-smith: Scaling data for software engineering agents. *arXiv preprint arXiv:2504.21798*.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. React: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*.

Zhaojian Yu, Xin Zhang, Ning Shang, Yangyu Huang, Can Xu, Yishujie Zhao, Wenxiang Hu, and Qiufeng Yin. 2024. WaveCoder: Widespread and versatile enhancement for code large language models by instruction tuning. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 5140–5153, Bangkok, Thailand. Association for Computational Linguistics.

Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. Autocoderover: Autonomous program improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2024, page 1592–1604, New York, NY, USA. Association for Computing Machinery.

Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, and 1 others. 2023. Pytorch fsdp: experiences on scaling fully sharded data parallel. *arXiv preprint arXiv:2304.11277*.

Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, and 1 others. 2024. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. *ArXiv preprint*, abs/2406.15877.

A Details of Collected Data

A.1 Distribution of Programming Languages

The proposed augmented data for mid-training can be constructed easily across any repository using different programming languages. The distribution of programming languages of our collected data is shown in Figure 3.

A.2 Filtering pull requests

To ensure the collection of high-quality data during the pull request (PR) scraping phase, we applied the following exclusion criteria:

- PRs originating from repositories with fewer than 100 stars or a total of fewer than 100 PRs, as this often indicates low repository quality or inconsistent contribution patterns.
- PRs that were not merged into the main branch, indicating that the proposed code changes were not accepted.
- PRs whose patches failed to apply cleanly to the codebase.
- PRs where `tree-sitter` encountered parsing errors in the changed files.
- PRs whose changed files exclusively consisted of non-code file extensions, including: `[".json", ".png", "csv", ".txt", ".md", ".jpg", ".jpeg", ".pkl", ".yml", ".yaml", ".toml"]`.

A.3 Filtering Training Data

After the construction of the SWE-Ext training data, further filtering is required to ensure data quality. First, we remove data samples that contain expert model API call errors, which are commonly caused by exceeding the context window length. Second, we remove samples with obvious errors for each data type:

- **File Localization data**: We remove samples where the ground truth edited files are not fully contained within the final prediction or are not included in the Phase 2 results.
- **Component Localization data**: We remove samples where the ground truth edited components are not fully contained in the final output.
- **Code Edit and Code Completion data**: We remove samples where the “search” code (the code to be changed) is not present in the provided context, as well as those with incomplete search-replace pairs.

To ensure the model is trained on complete conversational data, we also filter out any samples with a length exceeding 32768 tokens. Finally, we remove any dialogue data that corresponds to pull requests included in our evaluation datasets, including SWE-bench Verified and FEA-Bench, to prevent data leakage.

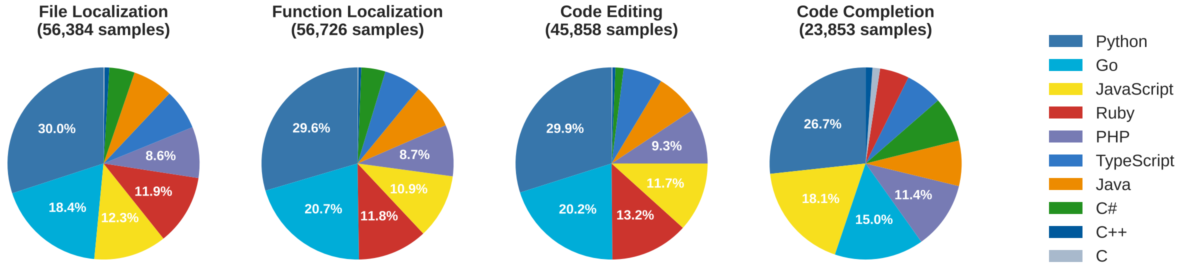

Figure 3: Language distribution of dialogues for 4 sub-tasks in SWE-Ext dataset.

B Training Settings

From our preliminary experiments, we found that using `Qwen2.5-Coder` as the base model yields strong performance, consistent with prior works on constructing training datasets (Yang et al., 2025b; Jain et al., 2025; Wang et al., 2025). We adopt `Qwen2.5-Coder-32B-Instruct` and `Qwen2.5-Coder-7B-Instruct` as the backbone models for fine-tuning. For the training framework, we use the multi-turn dialogue SFT framework provided by `verl` (Sheng et al., 2024), which is implemented on top of FSDP (Zhao et al., 2023).

For the 32B model, we fine-tune on two GPU workstations, each equipped with 2*8 NVIDIA A100 GPUs. Training is performed in `float32` precision with CPU offloading enabled. We set the maximum sequence length to 32,768. The training configuration includes a sequence parallel size of 4, a learning rate of 5×10^{-6}, a training batch size of 128, and training for 3 epochs. The total training time is approximately 2, 7, and 3 days on the SWE-EXT-BASELINE, SWE-EXT-MULTILINGUAL, and SWE-EXT-COMPLETION datasets, respectively.

For the 7B model, we fine-tune on a single workstation with 8 NVIDIA A100 GPUs. The setup remains the same as the 32B model, except that the sequence parallel size is increased to 4. The training time is similar to that of the 32B model.

C Evaluation Settings

All execution-based evaluations were conducted on a cloud computing instance equipped with a 32-core AMD EPYC 7763 processor @ 2.45GHz and 256 GB RAM, running Ubuntu 22.04.5 LTS with Linux kernel 6.8.0-1027-azure. It is important to note that in our evaluation environment, we encountered compatibility issues with certain test cases where 5 out of 500 gold patches in SWE-bench Verified and 9 out of 200 gold patches in FEA-Bench Lite failed to execute successfully due to environment-specific dependencies or configuration conflicts. As a result, the reported performance metrics may represent a slight underestimate of the true capabilities of our approach.

D Prompts of CosAgentless and Details of Augmented Data

In this section, we detail the prompts used and the specifics of how they construct our four task datasets. All of our data is built upon real-world GitHub pull requests and augmented based on their extracted ground truth results, which is why we refer to it as "**augmented data**". This data augmentation process consists of transforming the raw pull requests into a dialogue-based format, following the steps of our CosAgentless pipeline.

D.1 File localization

Phase 1: Initial File Localization. The first phase of the task is to select a top-5 list of relevant files based on the problem statement (`problem_statement`) and the provided file tree (`structure`). The prompt used for this task is shown in Figure 4.

Phase 2: Dependency Analysis and Reflection. The second phase analyzes the dependencies of the files selected in the previous stage based on their import relationships `import_content`. These relationships are derived from code segments identified by regular expressions. The model is then prompted to reflect its selection and choose a new list of up to 10 relevant files, as shown in Figure 5.

Phase 3: Refined File Localization. This phase leverages the results of parsing code repository. The model is provided with the structural skeletons (`file_internal_structure`, including class and function definitions) of the files identified in the previous stage. Using the skeletons of the files, the model makes a final selection, outputting a refined

Figure 4: Prompt for Refined File Localization in the pipeline of data construction.

Prompt for Initial File Localization

````
The problem description is as follows:
```
### GitHub Problem Description ###
{problem_statement}

###

### Candidate Files ###
{structure}

###
```
Let's locate the relevant file step by step using reasoning.
In order to locate accurately, you can pre-select {pre_select_num} files, and finally confirm {top_n} file names.

Based on the available information, confirm and provide complete name of the top-5 most likely relevant files that need
      to be edited for the problem.
You should output your reasoning process first.
Since your final answer will be processed automatically, please give your final answer of relevant files in the format as
       follows.
The returned files should be separated by new lines ordered by most to least important and wrapped with ```.
```
file1.py
file2.py
file3.py
file4.py
file5.py
```
Replace the 'file1.py' with the actual file path.
For example,
```
sklearn/linear_model/__init__.py
sklearn/base.py
```
````

list of at most 5 files as the final localization result. The prompt template used in this phase is shown in Figure 6.

The CosAgentless inference pipeline is structured as a sequential execution of the three phases detailed above. However, during the construction of our ground-truth-based augmented dataset, we employ a specific self-correction process: if the expert model fails to recall all the files in the ground truth code edit in its initial attempt, we provide the ground truth files as an explicit prompt to guide a re-prediction.

D.2 Component localization

The prompt for our component localization process is an enhanced version of the one presented in (Jiang et al., 2025), specifically adapted to generalize to a broader set of programming languages and repository-level coding tasks.

System Prompt. Within the system prompt shown in 7, we define a set of tool-use capabilities that enable the model to inspect the code of a specified component.

Iterative Localization Initialization. The iterative localization process begins with an initialization step where the model is prompted to identify a single, relevant component, as shown in Figure 8. This component serves as the initial point of inspection for the codebase.

Iterative Component Inspection. This process involves the iterative inspection of components. At each step, the model leverages the full dialogue history to inform its decision. And the instruction is shown in Figure 9. By making sequential function calls, it dynamically identifies the next component or code segment to examine, progressively refining its understanding of the problem and the codebase. `file_internal_structure` are the skeletons of found files in the file localization stage.

For each component `component_retrieved` viewed during an iteration, the model is prompted to assess its relevance to the given problem, as depicted in Figure 10. If the component is relevant to the problem description, the model is then directed to consider its internal call and dependency relationships to select the next component to be

viewed. Conversely, if a component is determined to be irrelevant, no further analysis is conducted on it.

Final Component Localization Output. The final component localization output is generated upon the termination of the iterative process. This occurs when the model either invokes the `exit()` function or reaches the maximum iteration limit. The model is then instructed to summarize its full inspection history and produce a final set of predicted components relevant to the `problem_statement`, as shown in Figure 11.

During the construction of our augmented data from GitHub pull requests, we incorporate a feedback mechanism. If the final localization output does not fully recall all of the locations specified by the ground truth code edit, we explicitly introduce the omitted locations back into the iterative inspection process. This allows the model to correct its course and regenerate a complete and accurate final localization result.

D.3 Code Editing

Final Code Editing. Leveraging the code at the locations identified by our component localization process, we construct a context block denoted as `top_n_content`. This contextual information is then provided to the large language model to facilitate the code editing task, as depicted in Figure 12.

Code Editing Output Format. The model's output is structured according to a specific format. The code edits are presented as **SEARCH-REPLACE** pairs, where the old code is provided alongside the corresponding new code. In addition, the model is required to generate a detailed editing plan and a clear rationale for the proposed changes. The response template is shown in Figure 13.

For the construction of our code editing dataset, we first transform the ground truth code edits into a structured format of **SEARCH-REPLACE** pairs. Subsequently, we utilize the expert model to generate the reasoning for these edits, populating the designated `plan` field of the response.

D.4 Code Completion

The code completion data is constructed from pull requests that lack a complete problem description (< 50 words). When generating these samples, we prioritize targeting a complete, newly added function within the patch. If no such function exists, the target becomes a contiguous block of new code. The code in the context (`top_n_content`) is then composed of the target file along with other relevant components identified via our CosAgentless localization pipeline. The line requiring completion is replaced with the token `[TODO]`, thereby converting the code completion task as an editing task centered on the `[TODO]` token. Consequently, the prompt and the subsequent augmented data generation steps align with those used for our code editing task.

E Data Examples

In this section, we present several short data examples for reference. Each turn of the dialogue is formatted as follows:

```
>>>>>>>>>>>>>>>>>>>>{role}
{content}
```

We select one example for each task from the constructed dataset. The file localization examples are shown in Figures 14–19. The component localization examples are presented in Figures 20–24. Examples of code editing and code completion data are illustrated in Figures 25–27 and Figures 28–29, respectively.

Figure 5: Prompt for Dependency Analysis and Reflection in the pipeline of data construction.

Prompt for Dependency Analysis and Reflection

````
Please look through the following problem description and repository structure and provide a list of files that one would
       need to edit to solve the software development problem.
I have already find 5 relevant files. According to the import relations, construct the call graph first.

### Problem Description ###
{problem_statement}

###

### Repository Structure ###
{structure}

###

### Files To Be Explored ###
{pre_files}

###

### Import Relations ###
{import_content}
###


Based on the import relationships, please analyze which files in the repository depend on which other files within the
      same repository. Ignore any libraries or modules that are imported from outside the current repository. Present the
       results in the following format:

file1.py -> file6.py, file7.py
file2.py ->
file3.py -> xxx/file8.py
file4.py -> file2.py, file9.py
file5.py -> xxx/file10.py, xxx/file11.py

Note: Solving the problem not only requires determining where to modify the code, but also identifying which other code
      to refer to in order to understand and invoke.

Based on the files listed above and the import relations, reconfirm and provide the complete names of the top 10 most
      likely relevant files, considering both where changes need to be made and which files are important to refer to.

Please think step by step and give you reasoning process first. Finally, provide full path and return top 10 files.

The final returned files should be separated by new lines ordered by most to least important and wrapped with ```
For example:
```
file1.py
file2.py
file3.py
file4.py
file5.py
file6.py
file7.py
file8.py
file9.py
file10.py
```
Note: file1.py indicates the top-1 file, file2.py indicates the top-2 file, and so on. Do not include test files.
````

Figure 6: Prompt for Refined File Localization in the pipeline of data construction.

Prompt for Refined File Localization

```
Please look through the following problem description and repository structure and provide a list of files that one would
       need to edit to solve the software development problem.
I have already find 10 relevent files according to the file structure and dependency.
I will further give the internal structures of these 10 files.
Please rank them again and reflect the result according to the internal structures and dependency.

### Problem Description ###
{problem_statement}

###

### Import Relations ###
{import_content}

###


### Files To Be Ranked ###
{file_internal_structure}

###

Please think step by step and give you reasoning process first. Finally, provide full path and return top 5 files.

The returned files should be separated by new lines ordered by most to least important and wrapped with ```
For example:
```
file1.py
file2.py
file3.py
file4.py
file5.py
```
Note: file1.py indicates the top-1 file, file2.py indicates the top-2 file, and so on. Do not include test files.
```

Figure 7: System Prompt for Component localization in the pipeline of data construction.

System Prompt for Component localization

```
You will be presented with a repository-level coding problem with repository file structure to access the source code of
      the software.
Since the modification is based on the code repository, the modified locations may include files, classes, and functions,
       and the modifications may be in the form of addition, deletion, or update.
Your task is to locate the top-5 most likely edit locations based on the problem description and the information you
      retrieve using given functions.
Function calls you can use are as follows:
* get_code_of_class('file_name', 'class_name') -> Get the code of a specified class in the given file and python project.
       'file_name' -> The name of the file. 'class_name' -> The name of the class. *
* get_code_of_class_function('file_name', 'class_name', 'func_name') -> Get the code of a specified function in the given
       class, file, and python project. 'file_name' -> The name of the file. 'class_name' -> The name of the class. '
      func_name' -> The name of the function. *
* get_code_of_file_function('file_name', 'func_name') -> Get the code of a specified function in the given file and
      python project. 'file_name' -> The name of the file. 'func_name' -> The name of the function. *
* get_toplevel_code('file_name') -> Get all the code in a given file that is not part of a class or function definition.
      This is useful for viewing imports, global variables, constants, and any top-level script logic. 'file_name' -> The
       name of the file. *
* exit() -> Exit function calling to give your final answer when you are confident of the answer. *
You have {max_try} chances to call function.
```

Figure 8: Prompt for the Initialization for Iterative Localization in the pipeline of data construction.

Prompt for the Initialization for Iterative Localization

```
### Problem Description ###
{problem_statement}

###
Let's locate the relevant elements (function/class) step by step using reasoning and function calls.
I have pre-identified top-5 relevant files. There stuctures are as follows:
{bug_file_list}
The formal parameter 'file_name' takes the value in "file:"
The formal parameter 'class_name' takes the value in "class:"
The formal parameter 'func_name' takes the value in "static functions:" and "class functions: "
Avoid making multiple identical calls to save overhead.
You must strictly follow the structure I give to call different tools.
For static functions, you can use 'get_code_of_file_function', and for class functions, you can use '
      get_code_of_class_function'.
In order to locate accurately, you can pre-select {pre_select_num} locations, then check them through function calls, and
       finally confirm {top_n} file names.
Don't make the first function call in this message.
```

Figure 9: Prompt for Iterative Component Inspection in the pipeline of data construction.

Prompt for Iterative Component Inspection

````
Now call a function in this format 'FunctionName(Argument)' in a single line without any other word or signal (such as
      ```).
Don't call the same function you've previous called, because this may waste your context length.
{file_internal_structure}
````

Figure 10: Prompt for Judgement for Relevant Components in the pipeline of data construction.

Prompt for Judgement for Relevant Components

````
You will be presented with a repository-level coding problem with repository file structure to access the source code of
      the software.

Your task is to locate the top-5 most likely edit locations based on the problem description.

### Problem Description ###
{problem_statement}

###

Here is a result of a function/class code retrived by '{content}'.
Please check if the code is related to the problem and if the code should be added into context.
<code>
{component_retrieved}
</code>
Return True if the code is related to the problem and should be added into context, otherwise return False.
Since your answer will be processed automatically, please give your answer in the format as follows.
The returned content should be wrapped with ```.
```
True
```
or
```
False
```
````
````

Figure 11: Prompt for Final Component Localization in the pipeline of data construction.

Prompt for Final Component Localization

```
{file_internal_structure}
Based on the available information, reconfirm and provide complete names of the most likely edit locations (10 locations
      at most).
Before make the final decision, please check whether the function name is correct or not, for static functions, don't add
       class name.
{bug_file_list}

Please provide the complete set of locations as either a class name, a function name, or a file name.
The returned files should be separated by new lines ordered by most to least important and wrapped with ```
Since your answer will be processed automatically, please give your answer in the exapmle format as follows.
```
top1_file_fullpath.py
function: Class1.Function1

top2_file_fullpath.py
function: Function2

top3_file_fullpath.py
class: Class3

top4_file_fullpath.py
function: Class4.Function4

top5_file_fullpath.py
function: Function5

top6_file_fullpath.py
global

top7_file_fullpath.py
function: Class7.Function7
```
Replace the 'Top_file_fullpath.py' with the actual file path, the 'Class' with the actual class name and the 'Function'
      with the actual function name. 'global' means the code is not in a class or function.
For example,
```
sklearn/linear_model/__init__.py
function: LinearRegression.fit
```
```

Figure 12: Prompt for Code Editing in the pipeline of data construction.

Prompt for Code Editing

````
We are currently solving the following task within our repository. Here is the task description.


### Task Description ###
{problem_statement}

###

Below are some code segments, each from a relevant file. One or more of these files may need to be edited to solve the
      task.


--- BEGIN FILE ---
```
{top_n_content}
```
--- END FILE ---

Please first localize the positions to edit based on the task statement, and then output the files that need to be
      deleted, modified or added.
'- file' means deleting the file;
'* file' means modificating the file;
'+ file' means adding the file.
The file should list like below:
```
- file1.py
* file2.py
* file3.py
+ file4.py
```

To solve the task, you should then generate *SEARCH/REPLACE* edits.

Every *SEARCH/REPLACE* edit must use this format:
1. The file path
2. The start of search block: <<<<<<< SEARCH
3. A contiguous chunk of lines to search for in the existing source code
4. The dividing line: =======
5. The lines to replace into the source code
6. The end of the replace block: >>>>>>> REPLACE

Here is an example:

```python
### mathweb/flask/app.py
<<<<<<< SEARCH
from flask import Flask
=======
import math
from flask import Flask
>>>>>>> REPLACE
```

Please note that the *SEARCH/REPLACE* edit REQUIRES PROPER INDENTATION. If you would like to add the line '        print(
      x)', you must fully write that out, with all those spaces before the code!
Wrap the *SEARCH/REPLACE* edit in blocks ```python...```.
When multiple edits should be done, please output *SEARCH/REPLACE* edit one by one and give your reasoning process before
       each *SEARCH/REPLACE* block.
````

Figure 13: Template for Code Editing Output in the pipeline of data construction.

Template for Code Editing Output

```
The plan of solving this software task:
{plan}
The files that should be edited include:
{files}

Here are my edits for code.
{search_replaces}
```

Figure 14: An Example of File Localization.

An Example of File Localization: Part I

````
>>>>>>>>>>>>>>>>>>>> system:

You will be presented with a repository-level coding problem with repository file structure to access the source code of
      the software.
Your task is to locate the most likely relavant files that need to be edited based on the problem description.


>>>>>>>>>>>>>>>>>>>> user:

The problem description is as follows:
```
### GitHub Problem Description ###
<request>
Adding fill parameters to pairwise and windowed
I found these handy for when I needed lookahead/lookbehind, but didn't want to add extra processing code before/after the
       processing loop. Here's a sketch of one way I've used it:

```python
for record, next_record in pairwise_iter(records, fill=None):
  if next_record is not None:
     record.link(next_record)
  record.process()
```

Without fill, I have to call `records[-1].process()` after the loop, assuming it's a list and not an iterable.
</request>

###

### Candidate Files ###
boltons/
    __init__.py
    cacheutils.py
    debugutils.py
    deprutils.py
    dictutils.py
    easterutils.py
    ecoutils.py
    excutils.py
    fileutils.py
    formatutils.py
    funcutils.py
    gcutils.py
    ioutils.py
    iterutils.py
    jsonutils.py
    listutils.py
    mathutils.py
    mboxutils.py
    namedutils.py
    pathutils.py
    queueutils.py
    setutils.py
    socketutils.py
    statsutils.py
    strutils.py
    tableutils.py
    tbutils.py
    timeutils.py
    typeutils.py
    urlutils.py
docs/
    conf.py
misc/
    bench_omd.py
    linkify_changelog.py
    table_html_app.py
setup.py

###
```


Let's locate the relevant file step by step using reasoning.
In order to locate accurately, you can pre-select 7 files, and finally confirm 5 file names.
````

Figure 15: An Example of File Localization.

An Example of File Localization: Part II

````
Based on the available information, confirm and provide complete name of the top-5 most likely relevant files that need
      to be edited for the problem.
You should output your reasoning process first.
Since your final answer will be processed automatically, please give your final answer of relevant files in the format as
       follows.
The returned files should be separated by new lines ordered by most to least important and wrapped with ```.
```
file1.py
file2.py
file3.py
file4.py
file5.py
```
Replace the 'file1.py' with the actual file path.
For example,
```
sklearn/linear_model/__init__.py
sklearn/base.py
```


>>>>>>>>>>>>>>>>>>>> assistant:
To solve the problem of adding fill parameters to `pairwise` and `windowed` functions, we need to identify where these
      functions are likely implemented. The problem description suggests that these functions are related to iterating
      over collections, which is a common task in utility modules dealing with iteration or functional programming.

Given the directory structure, the most relevant file is likely to be `iterutils.py` because:
- The name `iterutils.py` suggests it contains utilities for iteration, which aligns with the functions `pairwise` and `
      windowed` that deal with iterating over elements in a collection.

Let's consider the reasoning for selecting the files:

1. **boltons/iterutils.py**: This file is the most likely candidate as it is named to suggest it contains utilities for
      iteration, which is directly related to the `pairwise` and `windowed` functions mentioned in the problem.

2. **boltons/funcutils.py**: This file might contain functional utilities, which could include higher-order functions or
      iterators. If `pairwise` and `windowed` are implemented using functional programming paradigms, they might be here.

3. **boltons/listutils.py**: This file could potentially contain utilities for list operations. If `pairwise` and `
      windowed` are specifically designed for lists, they might be implemented here.

4. **boltons/typeutils.py**: This file might contain utilities related to type handling, which could include generic
      functions for handling iterables of various types.

5. **boltons/strutils.py**: Although less likely, if `pairwise` and `windowed` have any specific implementations for
      strings or string-like iterables, they might be here.

Now, let's finalize the top 5 files based on the reasoning:

```
boltons/iterutils.py
boltons/funcutils.py
boltons/listutils.py
boltons/typeutils.py
boltons/strutils.py
```

>>>>>>>>>>>>>>>>>>>> user:

Please look through the following problem description and repository structure and provide a list of files that one would
       need to edit to solve the software development problem.
I have already find 5 relevant files. Accroding to the import relations, construct the call graph first.

### Problem Description ###
<request>
Adding fill parameters to pairwise and windowed
I found these handy for when I needed lookahead/lookbehind, but didn't want to add extra processing code before/after the
       processing loop. Here's a sketch of one way I've used it:

```python
for record, next_record in pairwise_iter(records, fill=None):
  if next_record is not None:
    record.link(next_record)
  record.process()
```
````

Figure 16: An Example of File Localization.

An Example of File Localization: Part III

```
Without fill, I have to call `records[-1].process()` after the loop, assuming it's a list and not an iterable.
</request>

###

### Repository Structure ###
boltons/
    __init__.py
    cacheutils.py
    debugutils.py
    deprutils.py
    dictutils.py
    easterutils.py
    ecoutils.py
    excutils.py
    fileutils.py
    formatutils.py
    funcutils.py
    gcutils.py
    ioutils.py
    iterutils.py
    jsonutils.py
    listutils.py
    mathutils.py
    mboxutils.py
    namedutils.py
    pathutils.py
    queueutils.py
    setutils.py
    socketutils.py
    statsutils.py
    strutils.py
    tableutils.py
    tbutils.py
    timeutils.py
    typeutils.py
    urlutils.py
docs/
    conf.py
misc/
    bench_omd.py
    linkify_changelog.py
    table_html_app.py
setup.py

###

### Files To Be Explored ###
['boltons/iterutils.py', 'boltons/funcutils.py', 'boltons/listutils.py', 'boltons/typeutils.py', 'boltons/strutils.py']

###

### Import Relations ###
file: boltons/iterutils.py
 ['import os', 'import math', 'import time', 'import codecs', 'import random', 'import itertools']

file: boltons/funcutils.py
 ['from __future__ import print_function', 'import sys', 'import re', 'import inspect', 'import functools', 'import
      itertools', 'from types import MethodType, FunctionType']

file: boltons/listutils.py
 ['from __future__ import print_function, division', 'import operator', 'from math import log as math_log', 'from
      itertools import chain, islice']

file: boltons/typeutils.py
 ['import sys', 'from collections import deque']

file: boltons/strutils.py
 ['from __future__ import print_function', 'import re', 'import sys', 'import uuid', 'import zlib', 'import string', '
      import unicodedata', 'import collections', 'from gzip import GzipFile']

###


Based on the import relationships, please analyze which files in the repository depend on which other files within the
      same repository. Ignore any libraries or modules that are imported from outside the current repository. Present the
       results in the following format:
```

Figure 17: An Example of File Localization.

An Example of File Localization: Part IV

```
file1.py -> file6.py, file7.py
file2.py ->
file3.py -> xxx/file8.py
file4.py -> file2.py, file9.py
file5.py -> xxx/file10.py, xxx/file11.py

Note: Solving the problem not only requires determining where to modify the code, but also identifying which other code
      to refer to in order to understand and invoke.

Based on the files listed above and the import relations, reconfirm and provide the complete names of the top 10 most
      likely relevant files, considering both where changes need to be made and which files are important to refer to.

Please think step by step and give you reasoning process first. Finally, provide full path and return top 10 files.

The final returned files should be separated by new lines ordered by most to least important and wrapped with ```
For example:
```
file1.py
file2.py
file3.py
file4.py
file5.py
file6.py
file7.py
file8.py
file9.py
file10.py
```
Note: file1.py indicates the top-1 file, file2.py indicates the top-2 file, and so on. Do not include test files.


>>>>>>>>>>>>>>>>>>>> assistant:
To solve the problem, we need to analyze the import relationships and determine which files are relevant for adding the "
      fill" parameter to the `pairwise` and `windowed` functions. Here's the step-by-step reasoning process:

1. **Identify the Core Functionality**: The problem is about adding a "fill" parameter to the `pairwise` and `windowed`
      functions. These functions are likely to be part of the `iterutils.py` file, as they deal with iteration utilities.

2. **Analyze Import Relations**: We need to understand the dependencies between the files to see if any other files might
       be relevant for understanding or modifying the `pairwise` and `windowed` functions.

3. **Check Import Statements**:
   - `boltons/iterutils.py` imports standard libraries like `os`, `math`, `time`, `codecs`, `random`, and `itertools`.
      It does not import any other files from the `boltons` package.
   - `boltons/funcutils.py` imports `itertools` and other standard libraries, but does not import `iterutils`.
   - `boltons/listutils.py` imports `itertools` and other standard libraries, but does not import `iterutils`.
   - `boltons/typeutils.py` and `boltons/strutils.py` do not import `iterutils` or any other relevant files.

4. **Determine Relevant Files**: Since `iterutils.py` does not import or is not imported by any other files in the list,
      it is likely self-contained for the functionality of `pairwise` and `windowed`. However, understanding how these
      functions might interact with other utilities could be useful, so we consider the files that import `itertools` as
      they might have similar iteration-related utilities.

5. **Prioritize Files**: Based on the above analysis, `iterutils.py` is the most relevant file for direct modification.
      The other files (`funcutils.py`, `listutils.py`, `typeutils.py`, `strutils.py`) might provide additional context
      or similar implementations that could be useful for reference.

6. **Top 10 Files**: Since the problem is specific to iteration utilities, the most relevant file is `iterutils.py`. The
      other files are included for reference and understanding of similar patterns or utilities.

Based on this reasoning, here are the top 10 files, ordered by relevance:

```
boltons/iterutils.py
boltons/funcutils.py
boltons/listutils.py
boltons/typeutils.py
boltons/strutils.py
boltons/cacheutils.py
boltons/debugutils.py
boltons/deprutils.py
boltons/dictutils.py
boltons/easterutils.py
```
```

Figure 18: An Example of File Localization.

An Example of File Localization: Part V

````
>>>>>>>>>>>>>>>>>>>> user:

Please look through the following problem description and repository structure and provide a list of files that one would
       need to edit to solve the software development problem.
I have already find 10 relevent files according to the file structure and dependency.
I will further give the internal structures of these 10 files.
Please rank them again and reflect the result according to the internal structures and dependency.

### Problem Description ###
<request>
Adding fill parameters to pairwise and windowed
I found these handy for when I needed lookahead/lookbehind, but didn't want to add extra processing code before/after the
       processing loop. Here's a sketch of one way I've used it:

```python
for record, next_record in pairwise_iter(records, fill=None):
  if next_record is not None:
     record.link(next_record)
  record.process()
```

Without fill, I have to call `records[-1].process()` after the loop, assuming it's a list and not an iterable.
</request>

###

### Import Relations ###
file: boltons/iterutils.py
 ['import os', 'import math', 'import time', 'import codecs', 'import random', 'import itertools']

file: boltons/funcutils.py
 ['from __future__ import print_function', 'import sys', 'import re', 'import inspect', 'import functools', 'import
      itertools', 'from types import MethodType, FunctionType']

file: boltons/listutils.py
 ['from __future__ import print_function, division', 'import operator', 'from math import log as math_log', 'from
      itertools import chain, islice']

file: boltons/typeutils.py
 ['import sys', 'from collections import deque']

file: boltons/strutils.py
 ['from __future__ import print_function', 'import re', 'import sys', 'import uuid', 'import zlib', 'import string', '
      import unicodedata', 'import collections', 'from gzip import GzipFile']


###


### Files To Be Ranked ###
file: boltons/iterutils.py
        class: ['PathAccessError', 'GUIDerator', 'SequentialGUIDerator']
        static functions:  ['is_iterable', 'is_scalar', 'is_collection', 'split', 'split_iter', 'lstrip', 'lstrip_iter',
      'rstrip', 'rstrip_iter', 'strip', 'strip_iter', 'chunked', '_validate_positive_int', 'chunked_iter', 'chunk_ranges
      ', 'pairwise', 'pairwise_iter', 'windowed', 'windowed_iter', 'xfrange', 'frange', 'backoff', 'backoff_iter', '
      bucketize', 'partition', 'unique', 'unique_iter', 'redundant', 'one', 'first', 'flatten_iter', 'flatten', 'same', '
      default_visit', 'default_enter', 'default_exit', 'remap', 'get_path', 'research', 'soft_sorted', 'untyped_sorted']
        class functions: [
                PathAccessError: ['__init__', '__repr__', '__str__']
                GUIDerator: ['__init__', 'reseed', '__iter__', '__next__', '__next__']
                SequentialGUIDerator: ['reseed', 'reseed', '__next__']
        ]
file: boltons/funcutils.py
        class: ['InstancePartial', 'CachedInstancePartial', 'FunctionBuilder', 'MissingArgument', 'ExistingArgument']
        static functions:  ['inspect_formatargspec', 'get_module_callables', 'mro_items', 'dir_dict', 'copy_function', '
      partial_ordering', 'format_invocation', 'format_exp_repr', 'format_nonexp_repr', 'wraps', 'update_wrapper', '
      _parse_wraps_expected', '_indent', 'total_ordering', 'noop']
        class functions: [
                InstancePartial: ['_partialmethod', '__get__']
                CachedInstancePartial: ['_partialmethod', '__set_name__', '__get__']
                FunctionBuilder: ['_argspec_to_dict', '_argspec_to_dict', '__init__', 'get_sig_str', 'get_invocation_str
      ', 'get_sig_str', 'get_invocation_str', 'from_func', 'get_func', 'get_defaults_dict', 'get_arg_names', 'add_arg', '
      add_arg', 'remove_arg', '_compile']
                MissingArgument: []
                ExistingArgument: []
        ]
````

Figure 19: An Example of File Localization.

An Example of File Localization: Part VI

```
file: boltons/listutils.py
        (...omitted skeleton)
file: boltons/typeutils.py
        (...omitted skeleton)
file: boltons/strutils.py
        (...omitted skeleton)
file: boltons/cacheutils.py
        (...omitted skeleton)
file: boltons/debugutils.py
        (...omitted skeleton)
file: boltons/deprutils.py
        (...omitted skeleton)
file: boltons/dictutils.py
        (...omitted skeleton)
file: boltons/easterutils.py
        (...omitted skeleton)
###

Please think step by step and give you reasoning process first. Finally, provide full path and return top 5 files.

The returned files should be separated by new lines ordered by most to least important and wrapped with ```
For example:
```
file1.py
file2.py
file3.py
file4.py
file5.py
```
Note: file1.py indicates the top-1 file, file2.py indicates the top-2 file, and so on. Do not include test files.

>>>>>>>>>>>>>>>>>>>> assistant:
To solve the problem of adding fill parameters to the `pairwise` and `windowed` functions, we need to identify the files
       that contain these functions and any related utility functions that might be affected by or need to support the
      new feature.

### Step-by-Step Reasoning:

1. **Identify the Functions:**
   - The problem description specifically mentions `pairwise_iter` and implies a similar change might be needed for `
      windowed` functions. We need to locate these functions in the codebase.

2. **Locate the Functions:**
   - From the provided internal structure of the files, we see that `pairwise`, `pairwise_iter`, `windowed`, and `
      windowed_iter` are static functions in `boltons/iterutils.py`.

3. **Assess Dependencies:**
   - Since these functions are in `boltons/iterutils.py`, this file is the primary candidate for modification.
   - We should also consider if any other files might be using these functions or if there are utility functions that
      might need to be updated to support the new `fill` parameter.

4. **Check for Related Utilities:**
   - The `boltons/iterutils.py` file might have other utility functions that interact with `pairwise` and `windowed`. We
       should ensure that any such functions are also updated if necessary.

5. **Consider Other Files:**
   - While other files like `boltons/funcutils.py`, `boltons/listutils.py`, etc., are part of the repository, they do
      not contain the functions in question or directly relate to the problem at hand based on the provided information.

6. **Rank the Files:**
   - Given the focus on `pairwise` and `windowed`, `boltons/iterutils.py` is the most relevant file.
   - Other files are less relevant unless they are found to directly use or extend the functionality of `pairwise` and `
      windowed`.

### Conclusion:

Based on the above reasoning, the top file to edit is `boltons/iterutils.py`. Other files are not directly relevant to
      the problem as described, so they are not included in the top 5 list.

```
boltons/iterutils.py
```
```

Figure 20: An Data Example of Component Localization.

An Data Example of Component Localization: Part I

```
>>>>>>>>>>>>>>>>>>>> system:

You will be presented with a repository-level coding problem with repository file structure to access the source code of
      the software.
Since the modification is based on the code repository, the modified locations may include files, classes, and functions,
       and the modifications may be in the form of addition, deletion, or update.
Your task is to locate the top-5 most likely edit locations based on the problem description and the information you
      retrieve using given functions.

Function calls you can use are as follows:
* get_code_of_class('file_name', 'class_name') -> Get the code of a specified class in the given file and python project.
       'file_name' -> The name of the file. 'class_name' -> The name of the class. *
* get_code_of_class_function('file_name', 'class_name', 'func_name') -> Get the code of a specified function in the given
       class, file, and python project. 'file_name' -> The name of the file. 'class_name' -> The name of the class. '
      func_name' -> The name of the function. *
* get_code_of_file_function('file_name', 'func_name') -> Get the code of a specified function in the given file and
      python project. 'file_name' -> The name of the file. 'func_name' -> The name of the function. *
* get_toplevel_code('file_name') -> Get all the code in a given file that is not part of a class or function definition.
      This is useful for viewing imports, global variables, constants, and any top-level script logic. 'file_name' -> The
       name of the file. *
* exit() -> Exit function calling to give your final answer when you are confident of the answer. *

You have 10 chances to call function.


>>>>>>>>>>>>>>>>>>>> user:
### Problem Description ###
<request>
(PUP-5548) Use `service` in debian provider to interact with all init systems
In the Debian family, the `service` command provides an
abstraction around available init systems, allowing it to be used
to properly determine service state regardless of which init system
is active on the system.

This commit updates the Debian provider's `statuscmd` method to use
`service` at all times, allowing the provider to properly query
services regardless of OS version or init system in place.
</request>

###

Let's locate the relevant elements (function/class) step by step using reasoning and function calls.
I have pre-identified top-5 relevant files. There stuctures are as follows:
file: lib/puppet/provider/service/debian.rb
        class: []
        static functions:  ['disable', 'enabled?', 'get_start_link_count', 'enable', 'statuscmd']
        class functions: [
        ]
file: lib/puppet/provider/service/base.rb
        class: []
        static functions:  ['getps', 'getpid', 'status', 'statuscmd', 'start', 'startcmd', 'stop', 'stopcmd']
        class functions: [
        ]
file: lib/puppet/type/service.rb
        class: ['Puppet']
        static functions:  []
        class functions: [
                Puppet: ['retrieve', 'retrieve', 'sync', 'refresh']
        ]
file: lib/puppet/provider/service/systemd.rb
        class: []
        static functions:  ['instances', 'disable', 'get_start_link_count', 'enabled?', 'debian_enabled?', 'status', '
      enable', 'mask', 'unmask', 'restartcmd', 'startcmd', 'stopcmd']
        class functions: [
        ]
file: lib/puppet/provider/service/init.rb
        class: []
        static functions:  ['defpath', 'excludes', 'instances', 'get_services', 'hasstatus=', 'initscript', 'paths', '
      search', 'startcmd', 'stopcmd', 'restartcmd', 'texecute', 'statuscmd', 'is_init?']
        class functions: [
        ]
```

Figure 21: An Data Example of Component Localization.

An Data Example of Component Localization: Part II

```
The formal parameter 'file_name' takes the value in "file:"
The formal parameter 'class_name' takes the value in "class:"
The formal parameter 'func_name' takes the value in "static functions:" and "class functions: "
Avoid making multiple identical calls to save overhead.
You must strictly follow the structure I give to call different tools.
For static functions, you can use 'get_code_of_file_function', and for class functions, you can use '
      get_code_of_class_function'.
In order to locate accurately, you can pre-select 7 locations, then check them through function calls, and finally
      confirm 5 file names.
Don't make the first function call in this message.


>>>>>>>>>>>>>>>>>>>> assistant:
To address the problem, we need to focus on the `statuscmd` method in the Debian provider, as the request specifically
      mentions updating this method to use the `service` command. Based on the provided file structure, the relevant
      files and functions to consider are:

1. **lib/puppet/provider/service/debian.rb**
   - Function: `statuscmd`

2. **lib/puppet/provider/service/base.rb**
   - Function: `statuscmd`

3. **lib/puppet/provider/service/systemd.rb**
   - Function: `statuscmd`

4. **lib/puppet/provider/service/init.rb**
   - Function: `statuscmd`

These files contain the `statuscmd` function, which is likely related to the service status command functionality. The
      problem description specifically mentions the Debian provider, so the `debian.rb` file is of primary interest.
      However, examining the `statuscmd` function in other related files might provide additional context or necessary
      changes.

Let's proceed with checking the code of these functions to identify the necessary modifications. I will start by
      examining the `statuscmd` function in `lib/puppet/provider/service/debian.rb`.

>>>>>>>>>>>>>>>>>>>> user:

Now call a function in this format 'FunctionName(Argument)' in a single line without any other word or signal (such as
      ```).
Don't call the same function you've previous called, because this may waste your context length.

>>>>>>>>>>>>>>>>>>>> assistant:
get_code_of_file_function('lib/puppet/provider/service/debian.rb', 'statuscmd')

>>>>>>>>>>>>>>>>>>>> user:
I have already checked this function/class is related to the task.

>>>>>>>>>>>>>>>>>>>> user:
  def statuscmd
    os = Facter.value(:operatingsystem).downcase

    if os == 'debian'
      majversion = Facter.value(:operatingsystemmajrelease).to_i
    else
      majversion = Facter.value(:operatingsystemmajrelease).split('.')[0].to_i
    end

    if ((os == 'debian' && majversion >= 8) || (os == 'ubuntu' && majversion >= 15))
      # SysVInit scripts will always return '0' for status when the service is masked,
      # even if the service is actually stopped. Use the SysVInit-Systemd compatibility
      # layer to determine the actual status. This is only necessary when the SysVInit
      # version of a service is queried. I.e, 'ntp' instead of 'ntp.service'.
      (@resource[:hasstatus] == :true) && ["systemctl", "is-active", @resource[:name]]
    else
      super
    end
  end
```
```

Figure 22: An Data Example of Component Localization.

An Data Example of Component Localization: Part III

```
>>>>>>>>>>>>>>>>>>>> user:

Now call a function in this format 'FunctionName(Argument)' in a single line without any other word or signal (such as
      ```).
Don't call the same function you've previous called, because this may waste your context length.

You can check the functions that this part of code calls.

>>>>>>>>>>>>>>>>>>>> assistant:
get_code_of_file_function('lib/puppet/provider/service/base.rb', 'statuscmd')

>>>>>>>>>>>>>>>>>>>> user:
I have already checked this function/class is related to the task.

>>>>>>>>>>>>>>>>>>>> user:
  def statuscmd
  end

>>>>>>>>>>>>>>>>>>>> user:

Now call a function in this format 'FunctionName(Argument)' in a single line without any other word or signal (such as
      ```).
Don't call the same function you've previous called, because this may waste your context length.

You can check the functions that this part of code calls.

>>>>>>>>>>>>>>>>>>>> assistant:
get_code_of_file_function('lib/puppet/provider/service/systemd.rb', 'statuscmd')

>>>>>>>>>>>>>>>>>>>> user:
The call of get_code_of_file_function('lib/puppet/provider/service/systemd.rb', 'statuscmd') encountered error. Maybe you
       retrieved a location which does not exist.

>>>>>>>>>>>>>>>>>>>> user:

Now call a function in this format 'FunctionName(Argument)' in a single line without any other word or signal (such as
      ```).
Don't call the same function you've previous called, because this may waste your context length.

>>>>>>>>>>>>>>>>>>>> assistant:
get_code_of_file_function('lib/puppet/provider/service/init.rb', 'statuscmd')

>>>>>>>>>>>>>>>>>>>> user:
I have already checked this function/class is related to the task.

>>>>>>>>>>>>>>>>>>>> user:
  def statuscmd
    (@resource[:hasstatus] == :true) && [initscript, :status]
  end

>>>>>>>>>>>>>>>>>>>> user:

Now call a function in this format 'FunctionName(Argument)' in a single line without any other word or signal (such as
      ```).
Don't call the same function you've previous called, because this may waste your context length.

You can check the functions that this part of code calls.


>>>>>>>>>>>>>>>>>>>> assistant:
get_code_of_file_function('lib/puppet/provider/service/init.rb', 'initscript')

>>>>>>>>>>>>>>>>>>>> user:
I have already checked this function/class is not related to the task.

>>>>>>>>>>>>>>>>>>>> user:
  def initscript
    @initscript ||= self.search(@resource[:name])
  end
```

Figure 23: An Data Example of Component Localization.

An Data Example of Component Localization: Part IV

```
>>>>>>>>>>>>>>>>>>>> user:

Now call a function in this format 'FunctionName(Argument)' in a single line without any other word or signal (such as
      ```).
Don't call the same function you've previous called, because this may waste your context length.

Don't check the functions that this part of code calls.

>>>>>>>>>>>>>>>>>>>> assistant:
get_toplevel_code(lib/puppet/provider/service/debian.rb)

>>>>>>>>>>>>>>>>>>>> user:
I have already checked this function/class is related to the task.

>>>>>>>>>>>>>>>>>>>> user:
Manage debian services. Start/stop is the same as InitSvc, but enable/disable
is special.
Puppet::Type.type(:service).provide :debian, :parent => :init do
  desc <<-EOT
    Debian's form of `init`-style management.

    The only differences from `init` are support for enabling and disabling
    services via `update-rc.d` and the ability to determine enabled status via
    `invoke-rc.d`.

  EOT

  commands :update_rc => "/usr/sbin/update-rc.d"
  # note this isn't being used as a command until
  # http://projects.reductivelabs.com/issues/2538
  # is resolved.
  commands :invoke_rc => "/usr/sbin/invoke-rc.d"

  defaultfor :operatingsystem => :cumuluslinux
  defaultfor :operatingsystem => :debian, :operatingsystemmajrelease => ['5','6','7']

  # Remove the symlinks
...
end
...

>>>>>>>>>>>>>>>>>>>> user:

Now call a function in this format 'FunctionName(Argument)' in a single line without any other word or signal (such as
      ```).
Don't call the same function you've previous called, because this may waste your context length.

You can check the functions that this part of code calls.


>>>>>>>>>>>>>>>>>>>> assistant:
exit()

>>>>>>>>>>>>>>>>>>>> user:

Based on the available information, reconfirm and provide complete names of the most likely edit locations (10 locations
      at most).
Before make the final decision, please check whether the function name is correct or not, for static functions, don't add
       class name.
```

Figure 24: An Data Example of Component Localization.

An Data Example of Component Localization: Part V

````
Please provide the complete set of locations as either a class name, a function name, or a file name.
The returned files should be separated by new lines ordered by most to least important and wrapped with ```
Since your answer will be processed automatically, please give your answer in the exapmle format as follows.
```
top1_file_fullpath.py
function: Class1.Function1

top2_file_fullpath.py
function: Function2

top3_file_fullpath.py
class: Class3

top4_file_fullpath.py
function: Class4.Function4

top5_file_fullpath.py
function: Function5

top6_file_fullpath.py
global

top7_file_fullpath.py
function: Class7.Function7
```
Replace the 'Top_file_fullpath.py' with the actual file path, the 'Class' with the actual class name and the 'Function'
      with the actual function name. 'global' means the code is not in a class or function.
For example,
```
sklearn/linear_model/__init__.py
function: LinearRegression.fit
```


>>>>>>>>>>>>>>>>>>>> assistant:
```
lib/puppet/provider/service/debian.rb
global

lib/puppet/provider/service/debian.rb
function: statuscmd

lib/puppet/provider/service/base.rb
function: statuscmd

lib/puppet/provider/service/init.rb
function: statuscmd
```
````

Figure 25: An Data Example of Code Editing.

An Data Example of Code Editing: Part I

```
>>>>>>>>>>>>>>>>>>>>> user:

We are currently solving the following task within our repository. Here is the task description.


### Task Description ###
<request>
Add port mapping on run
CF issue : https://github.com/jolicode/docker-starter/issues/31
</request>

<discussion>
--------------------
Mapping ports on "run" command
Hi there,

Is there any reason to avoid mapping ports on "run" command ?
https://github.com/jolicode/docker-starter/blob/c319756cc1ab1de8e4a3f29bea306695752f84c2/fabfile.py#L179

I just needed to add --service-ports option to the fabfile.py to run my container with port mapping embed.

:kissing_heart:

**********
since we use docker-compose, it's better to set the port mapping in it.
But if you need something special, you can use port mapping via the run command.

Feel free to modify your fabfile. This docker-starter is not a lib, just a starter that must be customized according to
      your needs.
--service-ports does actually the mapping defined in docker-compose.yaml.
**********

</discussion>

###

Below are some code segments, each from a relevant file. One or more of these files may need to be edited to solve the
      task.


--- BEGIN FILE ---
```
fabfile.py
...
env.extra_domains = []
# This is the host directory containing your PHP application
env.project_directory = 'application'

services_to_build_first = [
    'php-base',
    'builder',
]

def with_builder(func):
    @wraps(func)

...(omitted file content)

```
--- END FILE ---
```

Figure 26: An Data Example of Code Editing.

An Data Example of Code Editing: Part II

````
Please first localize the positions to edit based on the task statement, and then output the files that need to be
      deleted, modified or added.
'- file' means deleting the file;
'* file' means modificating the file;
'+ file' means adding the file.
The file should list like below:
```
- file1.py
* file2.py
* file3.py
+ file4.py
```

To solve the task, you should then generate *SEARCH/REPLACE* edits.

Every *SEARCH/REPLACE* edit must use this format:
1. The file path
2. The start of search block: <<<<<<< SEARCH
3. A contiguous chunk of lines to search for in the existing source code
4. The dividing line: =======
5. The lines to replace into the source code
6. The end of the replace block: >>>>>>> REPLACE

Here is an example:

```python
### mathweb/flask/app.py
<<<<<<< SEARCH
from flask import Flask
=======
import math
from flask import Flask
>>>>>>> REPLACE
```

Please note that the *SEARCH/REPLACE* edit REQUIRES PROPER INDENTATION. If you would like to add the line '        print(
      x)', you must fully write that out, with all those spaces before the code!
Wrap the *SEARCH/REPLACE* edit in blocks ```python...```.
When multiple edits should be done, please output *SEARCH/REPLACE* edit one by one and give your reasoning process before
       each *SEARCH/REPLACE* block.


>>>>>>>>>>>>>>>>>>>> assistant:
The plan of solving this software task:

To address the task of adding port mapping on the "run" command, we need to modify the `docker_compose_run` function in
      the `fabfile.py`. The discussion suggests using the `--service-ports` option to enable port mapping as defined in
      the `docker-compose.yml` file. We will add a parameter to the `docker_compose_run` function to allow this option to
       be toggled.

The files that should be edited include:
* fabfile.py

Here are my edits for code.
````

Figure 27: An Data Example of Code Editing.

An Data Example of Code Editing: Part III

```
The motivation of this edit:
We need to modify the `docker_compose_run` function to include an optional parameter for port mapping. By adding a `
      port_mapping` parameter, we can control whether the `--service-ports` option is included in the Docker run command.
       This change allows users to enable port mapping when running a service, which is useful for exposing ports as
      defined in the `docker-compose.yml` file.

```python
### fabfile.py
<<<<<<< SEARCH
        ))


def docker_compose_run(command_name, service="builder", user="app", no_deps=False, workdir=None):
    args = [
        'run ',
        '--rm ',
=======
        ))


def docker_compose_run(command_name, service="builder", user="app", no_deps=False, workdir=None, port_mapping=False):
    args = [
        'run ',
        '--rm ',
>>>>>>> REPLACE
<<<<<<< SEARCH
    if no_deps:
        args.append('--no-deps ')

    if workdir is not None:
        args.append('-w %s ' % _shell_escape(workdir))

=======
    if no_deps:
        args.append('--no-deps ')

    if port_mapping:
        args.append('--service-ports ')

    if workdir is not None:
        args.append('-w %s ' % _shell_escape(workdir))

>>>>>>> REPLACE
```
```

Figure 28: An Data Example of Code Completion.

An Data Example of Code Completion: Part I

```
>>>>>>>>>>>>>>>>>>>> user:

We are currently solving the following task within our repository. Here is the task description.


### Task Description ###
Please help me to complete the code at the position of '[TODO]', in the b2/sync/scan_policies.py
    def __init__(
        self,
        exclude_dir_regexes=tuple(),
        exclude_file_regexes=tuple(),
        include_file_regexes=tuple(),
    ):

###

Below are some code segments, each from a relevant file. One or more of these files may need to be edited to solve the
      task.


--- BEGIN FILE ---
(...omitted content)

class ScanPoliciesManager(object):
    """
    Policy object used when scanning folders for syncing, used to decide
    which files to include in the list of files to be synced.
    """

    def __init__(
        self,
        exclude_dir_regexes=tuple(),
        exclude_file_regexes=tuple(),
        include_file_regexes=tuple(),
    ):
        # [TODO]

    def should_exclude_file(self, file_path):

(...omitted content)
```
```
--- END FILE ---

Please first localize the positions to edit based on the task statement, and then output the files that need to be
      deleted, modified or added.
'- file' means deleting the file;
'* file' means modificating the file;
'+ file' means adding the file.
The file should list like below:
```
```
- file1.py
* file2.py
* file3.py
+ file4.py
```
```

To solve the task, you should then generate *SEARCH/REPLACE* edits.

Every *SEARCH/REPLACE* edit must use this format:
1. The file path
2. The start of search block: <<<<<<< SEARCH
3. A contiguous chunk of lines to search for in the existing source code
4. The dividing line: =======
5. The lines to replace into the source code
6. The end of the replace block: >>>>>>> REPLACE

Here is an example:
```

Figure 29: An Example of Code Completion.

An Example of Code Completion: Part II

````
```python
### mathweb/flask/app.py
<<<<<<< SEARCH
from flask import Flask
=======
import math
from flask import Flask
>>>>>>> REPLACE
```

Please note that the *SEARCH/REPLACE* edit REQUIRES PROPER INDENTATION. If you would like to add the line '        print(
      x)', you must fully write that out, with all those spaces before the code!
Wrap the *SEARCH/REPLACE* edit in blocks ```python...```.
When multiple edits should be done, please output *SEARCH/REPLACE* edit one by one and give your reasoning process before
       each *SEARCH/REPLACE* block.

>>>>>>>>>>>>>>>>>>>> assistant:
The plan of solving this software task:
The task requires completing the `__init__` method of the `ScanPoliciesManager` class in the `b2/sync/scan_policies.py`
      file. The method should initialize the class attributes using the provided regular expression tuples. The class
      uses `RegexSet` objects to manage and check regular expressions. Therefore, the `__init__` method should create `
      RegexSet` instances for each of the regular expression tuples: `exclude_dir_regexes`, `exclude_file_regexes`, and `
      include_file_regexes`.

The files that should be edited include:
* b2/sync/scan_policies.py

Here are my edits for code.

The motivation of this edit:
The `ScanPoliciesManager` class is designed to manage policies for scanning directories and files, determining which
      should be included or excluded based on regular expressions. The `__init__` method needs to initialize the class
      attributes `_exclude_dir_set`, `_exclude_file_set`, and `_include_file_set` using the `RegexSet` class. This setup
       allows the `should_exclude_file` and `should_exclude_directory` methods to function correctly by checking file and
       directory paths against the compiled regular expressions.

```python
### b2/sync/scan_policies.py
<<<<<<< SEARCH
        exclude_file_regexes=tuple(),
        include_file_regexes=tuple(),
    ):
        # [TODO]

    def should_exclude_file(self, file_path):
        """
=======
        exclude_file_regexes=tuple(),
        include_file_regexes=tuple(),
    ):
        self._exclude_dir_set = RegexSet(exclude_dir_regexes)
        self._exclude_file_set = RegexSet(exclude_file_regexes)
        self._include_file_set = RegexSet(include_file_regexes)

    def should_exclude_file(self, file_path):
        """
>>>>>>> REPLACE
```
````