SWE-Bench-CL: Continual Learning for Coding Agents

Thomas Joshi
Columbia University
ttj2108@columbia.edu

Shayan Chowdhury
Columbia University
sc4040@columbia.edu

Fatih Uysal
Columbia University
fu2137@columbia.edu

Abstract

Large Language Models (LLMs) have achieved impressive results on static code-generation benchmarks, but real-world software development unfolds as a continuous stream of evolving issues, fixes, and feature requests. We introduce **SWE-Bench-CL**, a novel continual learning benchmark built on the human-verified SWE-Bench Verified dataset OpenAI and Princeton-NLP [2024]. By organizing GitHub issues into chronologically ordered sequences that reflect natural repository evolution, SWE-Bench-CL enables direct evaluation of an agent's ability to accumulate experience, transfer knowledge across tasks, and resist catastrophic forgetting. We complement the dataset with (i) a preliminary analysis of inter-task structural similarity and contextual sensitivity, (ii) an interactive, LangGraph-based evaluation framework augmented with a FAISS-backed semantic memory module, and (iii) a suite of specialized continual learning metrics—including average accuracy, forgetting, forward/backward transfer, tool-use efficiency, and a generalized novel Composite Continual-Learning-Score and CL-F_β score—to capture the stability-plasticity trade-off. We outline a rigorous experimental protocol comparing memory-enabled and memory-disabled agents across diverse Python repositories. All code and data are publicly available at `https://github.com/thomasjoshi/agents-never-forget/`, providing the community with a reproducible platform for developing more adaptive and robust AI agents in software engineering.

1 Introduction & Motivation

Large Language Models (LLMs) have achieved remarkable success in a variety of code-related tasks, from autocompletion to generating entire code snippets from natural language descriptions [Chen et al., 2021, Nijkamp et al., 2023]. However, the lifecycle of real-world software projects is inherently dynamic and continuous. Repositories evolve daily: APIs are deprecated, libraries are upgraded, new bugs are discovered and fixed, and novel features are constantly requested. An adept software engineering agent must therefore not only generate correct code for an immediate request but also learn from its experiences, adapt to changes in the codebase, and, crucially, retain knowledge of how to handle past issues as the project grows and shifts. A human software engineer who has resolved 100 bugs in a complex codebase will be more adept at the 101st bug than an engineer new to it. This ability to accumulate experience is critical for **agents that continuously learn**.

However, current leading benchmarks for code intelligence—such as CodeSearchNet [Husain et al., 2020], CodeXGLUE [Lu et al., 2021], and even the robust SWE-Bench dataset [Jimenez et al., 2024]—primarily evaluate models on isolated, static tasks. They typically present data as an unordered collection, lacking the temporal or sequential structure necessary to measure critical continual learning (CL) properties like adaptation to evolving contexts, knowledge retention over time, or the mitigation of catastrophic forgetting. These benchmarks often require only one-step retrieval or generation and

Preprint. Under review.

employ evaluation metrics (e.g., BLEU, Exact Match, pass@k) that do not quantify an agent's ability to learn continuously or transfer knowledge effectively across related tasks. Furthermore, as we will demonstrate (Section 5), evaluating sequentially structured, derived benchmarks like SWE-Bench-CL with harnesses designed for their static predecessors presents significant alignment challenges, further motivating the need for evaluation frameworks specifically designed for continual learning agents.

To bridge this significant gap, we introduce **SWE-Bench-CL**, a continual learning reformulation of the human-verified SWE-Bench Verified dataset OpenAI and Princeton-NLP [2024]—which itself is a refinement of the original SWE-Bench dataset Jimenez et al. [2024]. SWE-Bench-CL structures software engineering tasks (GitHub issues) from various repositories into chronologically ordered sequences, each designed to simulate a developer's ongoing engagement with a project. This temporal structuring allows for the direct assessment of an agent's ability to learn from a stream of tasks, adapt to new problems, and remember past solutions.

This paper makes the following primary contributions:

1. **A Novel Benchmark Dataset (SWE-Bench-CL):** In Section 3, we detail the construction and structure of SWE-Bench-CL, a reproducible, temporally organized benchmark designed to measure adaptation and memory retention in coding agents.
2. **Preliminary Dataset Analysis:** In Section 4, we present an analysis of SWE-Bench-CL's structural characteristics, including inter-task similarity and contextual sensitivity. These findings highlight the unique challenges the benchmark poses for continual learning and inform the design of effective evaluation strategies and agent architectures.
3. **A Proposed Agentic Evaluation Framework:** In Section 6, we propose a methodology for evaluating agents on SWE-Bench-CL. This framework centers on an interactive coding agent, built with LangGraph Langchain [2024] and inspired by the SWE-agent project Yang et al. [2024], augmented with a semantic memory module. It was developed to overcome challenges with existing harnesses (Section 5) for greater transparency in assessing continual learning.
4. **Specialized Continual Learning Metrics:** In Section 7, we define a suite of evaluation metrics tailored for assessing continual learning in software engineering, addressing success rate, tool use efficiency, knowledge transfer, and forgetting.
5. **A Rigorous Experimental Protocol:** In Section 8, we outline experiments designed to validate the continual learning metrics, comparing memory-enabled/disabled agents and measuring stability–plasticity trade-offs.

Our goal is to provide the research community with a robust benchmark and a principled evaluation approach to catalyze the development of more adaptive AI agents for software engineering. The codebase is available at `https://github.com/thomasjoshi/agents-never-forget/`.

2 Related Work

The evaluation of LLMs on code has rapidly advanced. Initial benchmarks like HumanEval [Chen et al., 2021] and MBPP [Austin et al., 2021] focused on functional correctness for small problems. CodeXGLUE [Lu et al., 2021] offered broader tasks. SWE-Bench [Jimenez et al., 2024] and its verified version OpenAI and Princeton-NLP [2024] advanced with realistic GitHub issues but assess isolated tasks. Their harnesses, while powerful for static evaluation, can be misaligned with derived datasets like SWE-Bench-CL, which have different base versions and sequential structures, as discussed in Section 5.

Continual Learning (CL) aims to enable systems to learn sequentially without catastrophic forgetting [Parisi et al., 2019, Delange et al., 2021]. While explored in computer vision, CL in complex generative tasks like software engineering is emerging.

LLM-based agents, such as SWE-agent Yang et al. [2024], which use tools and interactive reasoning (e.g., ReAct [Yao et al., 2023]), represent a promising direction for tackling complex software tasks. Memory augmentation, especially Retrieval Augmented Generation (RAG) [Lewis et al., 2021], is often used to provide external knowledge. Our work uniquely combines these threads by proposing SWE-Bench-CL, a benchmark specifically for evaluating the *continual learning* abilities of such agents in the software engineering domain, particularly through the use of task-history-based semantic memory.

3 The SWE-Bench-CL Benchmark

3.1 Motivation and Design Goals

Our primary motivation was to create a benchmark that moves beyond static, one-shot evaluation of coding LLMs and instead assesses their ability to learn and adapt over time within the context of evolving software projects. Traditional benchmarks, while valuable, do not capture an agent's capacity to:

- Accumulate knowledge from previously solved issues.
- Transfer learned patterns or solutions to new, related problems (forward transfer).
- Retain proficiency on older tasks after learning new ones (resist catastrophic forgetting).
- Adapt its problem-solving strategies or tool usage as it gains experience with a codebase.

SWE-Bench-CL is designed to directly measure these attributes by structuring tasks sequentially.

3.2 Dataset Construction

SWE-Bench-CL is a continual learning adaptation of the SWE-Bench Verified dataset OpenAI and Princeton-NLP [2024]. GitHub issues and their corresponding code patches were transformed into a series of learning sequences, each associated with a distinct software repository, designed to simulate a developer's learning trajectory within specific, real-world codebases.

Construction involved selecting repositories with sufficient task instances (≥ 15 tasks) from SWE-Bench Verified. For each repository, a learning sequence was created employing several strategies:

- **Curriculum Learning:** After being primarily ordered by their creation timestamp, tasks are further ordered by difficulty, estimated by human fix time (<15 min, 15 min - 1 hr, 1-4 hr, >4 hr), presenting easier tasks first. This curriculum-based approach is motivated by findings suggesting that training models on tasks of progressively increasing difficulty can lead to better generalization, improved learning efficiency, and enhanced performance [Khajehabdollahi et al., 2024, Bengio et al., 2009], and facilitate more stable and effective learning when fine-tuning LLMs [Long et al., 2025, Shi et al., 2025].
- **Dependency Awareness:** Modified file paths from ground truth patches (`patch`) were extracted for each task. This information was used to identify potential dependencies between tasks (i.e., tasks modifying overlapping sets of files) for the future study of knowledge transfer.

The resulting dataset comprises 8 sequences from distinct repositories, totaling 273 tasks. Each task includes metadata (repository, base commit, creation date, difficulty), the problem statement, developer hints, evaluation details (ground truth patch, test setup patch, `FAIL_TO_PASS` and `PASS_TO_PASS` test case lists), and continual learning context (sequence position, difficulty score, potential dependencies, modified files). High-level statistics are summarized in Table 1. The complete dataset is available in JSON format on our GitHub repository.

Table 1: **SWE-Bench-CL Dataset Statistics per Sequence**

Repository	Tasks	Easy (<15m)	Medium (15m-1h)	Hard (1-4h)	Very Hard (>4h)	Tasks w/ Dependencies (%)
django/django	50	50	0	0	0	25 (50%)
sympy/sympy	50	25	25	0	0	12 (24%)
sphinx-doc/sphinx	44	22	17	4	1	23 (52%)
matplotlib/matplotlib	34	15	19	0	0	13 (38%)
scikit-learn/scikit-learn	32	13	18	1	0	4 (13%)
astropy/astropy	22	4	15	3	0	3 (14%)
pydata/xarray	22	5	15	1	1	13 (59%)
pytest-dev/pytest	19	8	8	3	0	7 (37%)

4 Preliminary Analysis of SWE-Bench-CL: Characteristics and Implications

To better understand the nature of SWE-Bench-CL and the specific challenges it presents for continual learning agents, we conducted preliminary analyses of its structural properties. These analyses inform the design of our proposed evaluation framework and highlight considerations for developing agents capable of continuous learning in software engineering contexts.

4.1 Low Inter-Task Structural Similarity

We investigated the structural overlap between the ground truth patches of different tasks within SWE-Bench-CL using Jaccard similarity of token sets and cosine similarity of TF-IDF embeddings (Figure 1). Our findings indicate a high degree of independence among most issue-patch pairs. The average Jaccard similarity was 0.1114 and the average cosine similarity was 0.1792. Few task pairs exhibited substantial overlap, with only one pair exceeding a cosine similarity of 0.4. Even when stratified by difficulty, within-group similarities remained modest (e.g., Easy-Easy Jaccard = 0.1225), and across-group similarities were lower still (Easy-Hard Jaccard = 0.0353).

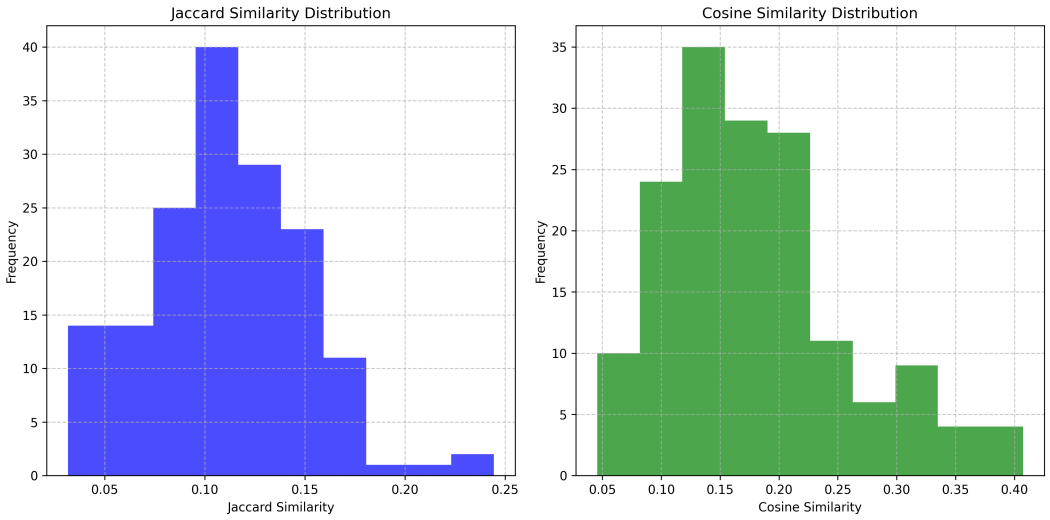

Figure 1: Distribution of Patch-Level Similarity Across Tasks in SWE-Bench-CL. Left: Jaccard similarity. Right: Cosine similarity. Most patch/task pairs exhibit low structural similarity, reinforcing SWE-Bench-CL's role as a high-variance benchmark.

Implications This low structural redundancy suggests that agents cannot rely on simple surface-level pattern matching between task solutions. It amplifies the challenge of catastrophic forgetting, as knowledge from one task may not be directly reinforced by subsequent, dissimilar tasks. Effective forward transfer will likely depend on learning more abstract problem-solving strategies or leveraging explicit memory of semantically (rather than structurally) similar past experiences. This characteristic validates the need for a benchmark like SWE-Bench-CL that explicitly tests for retention and transfer across distinct problems. It also motivates the inclusion of a semantic memory module in our proposed evaluation agent. While global overlap is low, some repositories (e.g., `django/django`) showed localized reuse in common modules, suggesting opportunities for intra-repository transfer.

4.2 Contextual Sensitivity and Prompt Poisoning

To assess how LLM-based agents might be affected by potentially irrelevant contextual information (e.g., from a RAG system retrieving sub-optimal memories), we performed a "prompt poisoning" experiment (Algorithm 1). For a target task B, we compared the semantic drift $(1 - \cos(\text{solution}_{\text{clean}}, \text{solution}_{\text{poisoned}}))$ in generated solutions when the prompt for B was prepended with an unrelated issue-patch pair from task A.

Algorithm 1 Prompt Poisoning Drift Analysis

Require: SWE-Bench-CL, task difficulty labels, cosine_similarity(), LLM

1: For $(d_{\text{src}}, d_{\text{tgt}})$ difficulty pairs:
2: Sample N unrelated task pairs (A, B) with difficulty$(A) = d_{\text{src}}$, difficulty$(B) = d_{\text{tgt}}$.
3: For each (A, B):
4: $r_{\text{clean}} \leftarrow \text{LLM}(\text{prompt}(B))$
5: $r_{\text{poisoned}} \leftarrow \text{LLM}(\text{prompt}(A) + \text{prompt}(B))$
6: Record $drift \leftarrow 1 - \cos(r_{\text{clean}}, r_{\text{poisoned}})$.
7: Aggregate mean drift.

As shown in Figure 2, even structurally dissimilar prompts from easier tasks induced consistently high semantic drift (average ≈ 0.45) in solutions for more difficult target tasks. While differences between target difficulty groups were not always statistically significant due to sample sizes, the overall high drift indicates that LLM outputs are sensitive to contextual inputs, even if those inputs are not directly relevant to the immediate task.

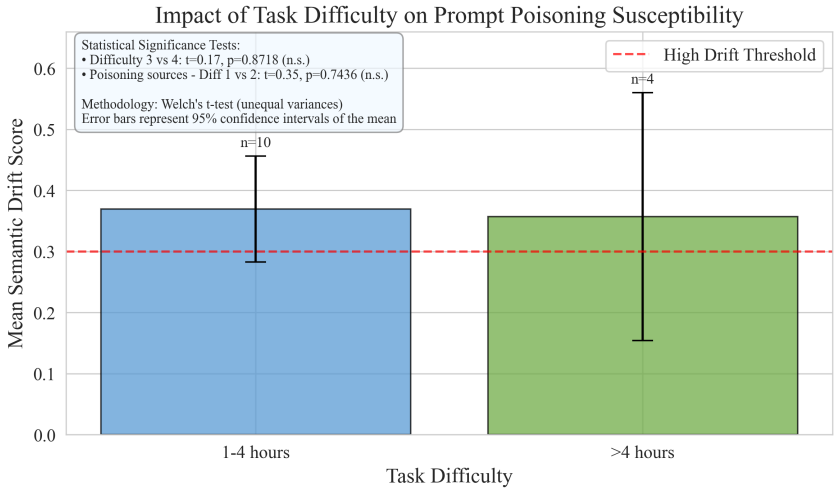

Figure 2: Mean semantic drift induced by prompt poisoning across SWE-Bench-CL tasks, grouped by difficulty of the target task ($1-4$ hours vs. >4 hours). Drift is computed as $1-\cos(\text{clean}, \text{poisoned})$. Red dashed line marks the "high drift" threshold (0.3). Error bars represent 95% confidence intervals.

These findings highlight a vulnerability for memory-augmented agents: naively retrieving and incorporating past experiences can degrade performance if the retrieval mechanism is not highly discerning or if the agent cannot effectively gate irrelevant information. This motivates the need for sophisticated semantic retrieval in our proposed agent's memory system and underscores the importance of evaluating how agents *use* memory, not just whether they have access to it.

4.3 Informing Agent Design and Fine-Tuning Strategies

The structural properties of SWE-Bench-CL directly inform design choices for both agentic evaluation frameworks and potential fine-tuning strategies developing continually learning coding LLMs:

- **Agent Memory Design:** The prevalence of distinct tasks suggests that an effective memory system for agents evaluated on SWE-Bench-CL should prioritize semantic similarity (e.g., issue type, error patterns) over superficial structural similarity for retrieving past experiences.
- **Evaluation of Context Use:** Agentic evaluation frameworks should not only measure task success but also analyze how agents interact with and are influenced by retrieved context, especially given the observed sensitivity to prompt poisoning.
- **Curriculum for Fine-Tuning:** The dataset's inherent curriculum structure (chronological and difficulty-based ordering) provides a natural sequence for fine-tuning LLMs. A fine-tuning protocol could leverage this sequential nature to adapt a base LLM to specific repositories or general software engineering patterns over time, by sequentially fine-tuning on tasks and evaluating for knowledge accumulation and forgetting. This is complementary to the RAG-based memory in our proposed test-time agent.

These preliminary analyses confirm that SWE-Bench-CL provides a rich and challenging environment for developing and evaluating agents that aim to continuously learn.

5 Empirical Evaluation & Motivation for an Agentic Framework

To empirically ground the need for a specialized evaluation approach, we attempted to evaluate agents on SWE-Bench-CL tasks using the official SWE-Bench evaluation harness [Jimenez et al.,

2024]. This harness provides **pre-configured Docker "dump containers"** primarily for the original SWE-Bench and SWE-Bench Lite datasets, and does *not* natively support the SWE-Bench Verified subset from which SWE-Bench-CL is derived. We generated patches with a range of LLMs (e.g., `CodeLlama-13B` Roziere et al. [2023], `Gemma-3-12B`, Gemma-Team et al. [2025], `Mistral-7B` Jiang et al. [2023] via Ollama, and Google's `Gemini-2.0-Flash`) using a standardized prompt (see Appendix A) under two main conditions: a baseline (independent tasks) and a memory-enabled condition (prompt augmented with semantically retrieved information from previously attempted tasks).

This endeavor highlighted several critical incompatibilities:

- **Mismatch of Containers and Task Definitions:** The harness's Docker containers and associated test execution scripts are tightly coupled to the specific file layouts, patch formats, and codebase states of the original SWE-Bench instances. SWE-Bench-CL, being built upon SWE-Bench Verified and restructured chronologically, often presents tasks where these assumptions no longer hold. Even **minor differences** can lead to **runtime errors** or **silent evaluation failures**.
- **Inconclusive and Unreliable Results:** After adapting the harness for SWE-Bench-CL, performance metrics were extremely low and highly variable, with frequent harness execution issues. As shown in Figure 3, overall pass rates remained consistently low (generally below 8.5%, often significantly lower), with the memory-enabled condition typically performing on par with or slightly worse. Similarly, Character Levenshtein distances to ground truth patches (Figure 4) were persistently high. This made it impossible to draw reliable conclusions.

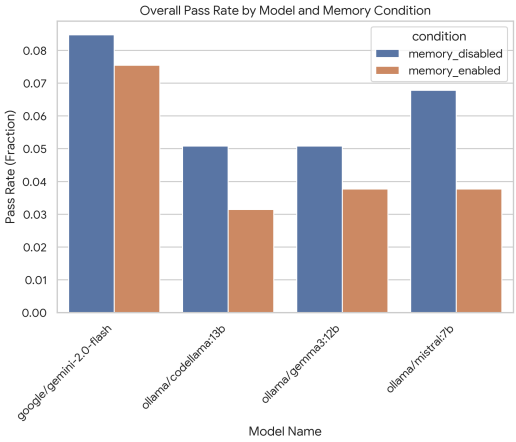

Figure 3: Overall Pass Rate on SWE-Bench-CL tasks using the standard SWE-Bench harness. Low pass rates highlight mismatch with static tooling.

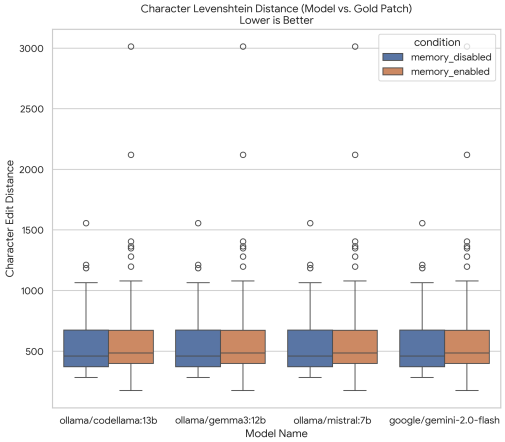

Figure 4: Character Levenshtein Distance (Model vs. Gold Patch) for SWE-Bench-CL tasks. High distances indicate evaluation challenges.

The generally **poor performance of the memory-enabled condition** (Figure 3) is plausible: high failure rates due to harness incompatibility likely populated "memory" with information from predominantly failed prior attempts, creating a **"garbage-in, garbage-out"** scenario.

These findings underscore a fundamental limitation of evaluating evolving, derived benchmarks with static, patch-only frameworks: they cannot flexibly accommodate variations differing from their original target. This critical gap motivated our development of the proposed agentic evaluation framework (Section 6), which treats evaluation as an interactive process for a more reliable assessment of problem-solving and learning capabilities on SWE-Bench-CL.

6 Proposed Agentic Evaluation Framework

To effectively evaluate performance on the SWE-Bench-CL benchmark, we propose an interactive, agent-based framework integrating semantic memory, inspired by SWE-agent Yang et al. [2024].

6.1 Rationale for a Custom Agent-Based Evaluation Framework

Evaluating continual learning in software engineering necessitates more than static code generation. Initial attempts to use the standard SWE-bench harness [Jimenez et al., 2024] with our custom dataset and CL setup faced challenges like opaque error messages, hindering debugging and analysis. To achieve greater transparency, control over agent-environment interaction, and enhanced debuggability for CL experiments—particularly for integrating complex agentic behaviors like semantic memory—we propose a custom agentic framework providing clearer insights into:

- **Tool Use:** Agent's ability to select and use tools effectively.
- **Multi-Step Reasoning:** Iterative sequences of actions, observations, and plan adjustments.
- **Feedback Incorporation:** Interpreting tool feedback (e.g., linter errors, test failures) and adapting.
- **Simulating Experience:** Agent interacts with the environment and remembers past interactions.

6.2 Agent Architecture and Agent-Computer Interface (ACI)

Implemented using LangGraph Langchain [2024] for stateful, graph-based execution, the agent is model-agnostic (configurable for OpenAI, Anthropic, Google, and local Ollama models) with standardized generation parameters (e.g., low `temperature`, `max_tokens`). Agent state is tracked in a Pydantic model (`AgentState`). Drawing inspiration from SWE-agent Yang et al. [2024], we developed an Agent-Computer Interface (ACI) and structured interaction patterns, while preserving the LangGraph foundation and integrating our novel semantic memory system.

Based on SWE-agent Yang et al. [2024], our ACI provides LLMs with tools to interact with the software environment: **Navigation/Search** (`find_file`, `search`), **File Viewing** (`file_viewer` with windowed view), **Editing** (`edit` tool with `flake8` linting before applying edits), and **Execution** (`run_tests` for shell commands). These tools accept necessary context from `AgentState` and return concise, structured, LM-parsable feedback.

6.3 Semantic Memory System

To explicitly model and evaluate continual learning, we introduce a `MemorySystem` built using a FAISS Douze et al. [2024] vector index. This distinguishes our approach and allows direct comparison of agent performance with/without access to learned experiences.

- **Storage:** Upon completing a task attempt, key aspects (summaries of problem, solution, rationale; tool usage; success status) are vectorized (e.g., using OpenAI's `text-embedding-3-small`, `nomic-embed-text` Nussbaum et al. [2025]) and stored.
- **Retrieval (RAG):** When initiating a new task, the agent queries memory using the current task's problem statement/hints, prioritizing experiences from the same task sequence.
- **Context Integration:** Top-k retrieved memories (including success status, relevance score) are formatted and prepended to the agent's initial prompt for the new task, subject to a configurable token limit (`max_context_tokens`).

6.4 Task Execution Workflow

For each task in a SWE-Bench-CL sequence, the agent performs:

1. **Repository Setup:** Repository is cloned and/or reset to the task's `base_commit`.
2. **Agent Interaction:** Using ACI tools and memory, agent iteratively generates `DISCUSSION`/`COMMAND` pairs. Commands trigger tools, update `AgentState`, and return results. Error handling manages failures.
3. **Termination:** Loop concludes on `submit` command, reaching a turn limit, or excessive errors.

7 Proposed Evaluation Metrics

To assess continual learning capabilities on SWE-Bench-CL, we define metrics for: (i) *solving new issues*, (ii) *retaining prior knowledge*, (iii) *transferring knowledge*, and (iv) *operating efficiently*. We record performance by testing on all previously seen tasks after each new issue, yielding a performance matrix $a_{i,j}$.

Notation Let N be total tasks, $a_{i,j}$ success rate on task j after training on task i, $\bar{a}_{0,j}$ zero-shot success on task j.

Metric	Formula	Definition
$\mathrm{SR}_{i,j}$	$a_{i,j} = \frac{p_j^{\text{pass}}}{p_j^{\text{total}}}$	Per-task success rate: immediate proficiency on task j after training on i.
ACC	$\frac{1}{N}\sum_{j=1}^{N} a_{N,j}$	Average accuracy after learning all N tasks.
F	$\frac{1}{N-1}\sum_{j=1}^{N-1}\left(\max_{1\le k\le j} a_{k,j} - a_{N,j}\right)$	Average forgetting: performance degradation on earlier tasks.
FT	$\frac{1}{N-1}\sum_{i=1}^{N-1}\left(a_{i,i+1} - \bar{a}_{0,i+1}\right)$	Forward transfer: benefit of prior tasks on new ones (vs. zero-shot).
BWT	$\frac{1}{N-1}\sum_{i=1}^{N-1}\left(a_{N,i} - a_{i,i}\right)$	Backward transfer: impact of learning new tasks on past tasks.
AULC	$\frac{1}{N}\sum_{i=1}^{N}\left(\frac{1}{i}\sum_{k=1}^{i} a_{k,k}\right)$	Area under learning curve: integrated performance over training steps.
TUE	$\frac{\text{median(time} \mid \text{success)}}{\text{median(time} \mid \text{all)}}$	Tool-use efficiency: ratio of median execution times for successful vs. all runs.

Composite CL-Score: Combines metrics into a single score with tunable weights λ:

$$\text{CL-Score} = \mathrm{ACC} - \lambda_F\, F + \lambda_{FT}\,\mathrm{FT} + \lambda_{BWT}\,\mathrm{BWT} + \lambda_{AULC}\,\mathrm{AULC} + \mathrm{CL{-}F}_\beta, \quad \lambda \ge 0$$

Continual-Learning-F_β score ($\mathrm{CL{-}F}_\beta$) is defined in Section 7.1.

Role of the λ Weights: All λ coefficients may be adjusted to reflect relative priorities:

Factor	Name	Description and Use Case
λ_{F}	**Forgetting Penalty**	Increase for high memory retention (e.g., safety-critical code).
λ_{FT}	**Forward Transfer Reward**	Increase for rapid adaptation to new functions (e.g., rapid-prototyping).
λ_{BWT}	**Backward Transfer Reward**	Increase if later tasks should improve earlier ones (e.g., modular libraries).
λ_{AULC}	**Learning Speed Factor**	Modulates initial learning speed vs. eventual proficiency. Increase for fast competence.
λ_{TUE}	**Tool-Use Efficiency**	Increase to prioritize resource efficiency (fewer tool calls, less time) for practical deployment.

The specific values for λ should be determined based on research questions or deployment goals.

7.1 Continual-Learning F1 (CL-F1)

The core challenge in CL is the *stability-plasticity dilemma* [Grossberg and Grossberg, 1982, Mermillod et al., 2013]: an agent must be plastic enough to learn new information effectively, yet stable enough to prevent new learning from catastrophically disrupting previously acquired knowledge. We propose the Continual Learning F1-Score (CL-F1) to explicitly quantify this trade-off.

We define two components:

- **CL-Plasticity (CL-P): Immediate Proficiency.** Measures the agent's ability to learn and correctly solve new tasks. Defined as the average success rate on each task i immediately after it is processed:

$$\mathrm{CL\text{-}P} = \frac{1}{N}\sum_{i=1}^{N} a_{i,i}$$

- **CL-Stability (CL-S): Knowledge Retention.** Measures the agent's ability to retain performance on previously learned tasks. Defined as one minus Average Forgetting (F):

$$\mathrm{CL\text{-}S} = 1 - \mathrm{F} = 1 - \frac{1}{N-1}\sum_{j=1}^{N-1}\left(\max_{1\le k\le j} a_{k,j} - a_{N,j}\right)$$

The **CL-F1 score** is the harmonic mean of CL-Plasticity and CL-Stability:

$$\boxed{\text{CL-F1} = 2 \cdot \frac{\text{CL-P} \times \text{CL-S}}{\text{CL-P} + \text{CL-S}}} \qquad (0 \leq \text{CL-F1} \leq 1)$$

Interpretation and Rationale. A high CL-F1 score is achieved when the agent demonstrates both strong immediate learning (high CL-P) and robust retention (high CL-S). The harmonic mean penalizes imbalance. This metric addresses the stability-plasticity dilemma by rewarding a balance, adapting F-scores to sequential learning [cf. Chaudhry et al., 2018, Lopez-Paz and Ranzato, 2022].

Generalized CL-F$_\beta$ Score. We introduce the CL-F$_\beta$ score:

$$\boxed{\text{CL-F}_\beta = (1 + \beta^2) \frac{\text{CL-P} \times \text{CL-S}}{\beta^2 \, \text{CL-P} + \text{CL-S}}} \qquad (\beta > 0)$$

The parameter β controls weighting: $\beta = 1$ (balanced CL-F1), $0 < \beta < 1$ (emphasizes plasticity), $\beta > 1$ (emphasizes stability). The choice of β depends on task requirements; $\beta = 1$ is standard for general benchmarking.

8 Proposed Experiments and Hypotheses

The CL metrics (Section 7) provide the quantitative foundation for experiments crucial for validating SWE-Bench-CL and our proposed metrics. While full empirical evaluation using our agentic framework is ongoing, key experiments will:

Compare memory-enabled versus memory-disabled agents using various LLMs (e.g., GPT-4o, Claude 3.7 Sonnet, open models like DeepSeek-V3 DeepSeek-AI et al. [2025] or Gemma3-27B Gemma-Team et al. [2025]) across multiple SWE-Bench-CL sequences. Key analyses include:

- **Impact of Semantic Memory:** Quantify performance gains (ACC, SR$_{i,j}$), forward transfer (FT), and tool-use efficiency (TUE) due to the semantic memory system. It is hypothesized that memory will improve overall accuracy, yield positive forward transfer, and enhance tool-use efficiency.
- **Stability-Plasticity Trade-off:** Assess average forgetting (F), CL-Plasticity (CL-P), CL-Stability (CL-S), and the composite CL-F$_\beta$ score. It is hypothesized that memory-augmented agents will exhibit higher CL-Stability (lower forgetting) and achieve a better CL-F$_\beta$ score, demonstrating an improved balance between learning new tasks and retaining old knowledge.

9 Limitations

While SWE-Bench-CL advances continual-learning benchmarks for coding agents, it has limitations:

- **Repository and Language Scope.** Evaluates only 8 open-source Python repositories, each with at most 50 issues per sequence. This scope may not generalize to other languages, larger projects, or development workflows with branching, parallel issue resolution, and pull-request reviews.
- **Coarse Difficulty and Dependency Signals.** Task difficulty is estimated by human fix-time categories, which are subjective. Dependency awareness relies on overlapping file paths, potentially missing deeper semantic or API-level dependencies.
- **Static Sequence Assumptions.** Issues are presented strictly in chronological order with a built-in curriculum. Real codebases often interleave unrelated issues, hotfixes, and refactorings; our simplified sequence may not reflect true software evolution dynamics.

10 Conclusion

The long-term vision is to cultivate coding agents that not only address immediate software engineering challenges but also continuously augment their expertise and reliability through sustained interaction with evolving codebases and task streams, truly embodying the principle of "agents that continuously learn."

References

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models, 2021. URL https://arxiv.org/abs/2108.07732.

Yoshua Bengio, Jerome Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *Proceedings of the 26th International Conference on Machine Learning (ICML)*, 2009.

Arslan Chaudhry, Puneet K. Dokania, Thalaiyasingam Ajanthan, and Philip H. S. Torr. *Riemannian Walk for Incremental Learning: Understanding Forgetting and Intransigence*, page 556–572. Springer International Publishing, 2018. ISBN 9783030012526. doi: 10.1007/978-3-030-01252-6_33. URL http://dx.doi.org/10.1007/978-3-030-01252-6_33.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021. URL https://arxiv.org/abs/2107.03374.

DeepSeek-AI et al. Deepseek-v3 technical report, 2025. URL https://arxiv.org/abs/2412.19437.

Matthias Delange, Rahaf Aljundi, Marc Masana, Sarah Parisot, Xu Jia, Ales Leonardis, Greg Slabaugh, and Tinne Tuytelaars. A continual learning survey: Defying forgetting in classification tasks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, page 1–1, 2021. ISSN 1939-3539. doi: 10.1109/tpami.2021.3057446. URL http://dx.doi.org/10.1109/TPAMI.2021.3057446.

Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. The faiss library. 2024.

Gemma-Team et al. Gemma 3 technical report, 2025. URL https://arxiv.org/abs/2503.19786.

Stephen Grossberg and Stephen Grossberg. How does a brain build a cognitive code? *Studies of mind and brain: Neural principles of learning, perception, development, cognition, and motor control*, pages 1–52, 1982.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. Codesearchnet challenge: Evaluating the state of semantic code search, 2020. URL https://arxiv.org/abs/1909.09436.

Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. Mistral 7b, 2023. URL https://arxiv.org/abs/2310.06825.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues?, 2024. URL https://arxiv.org/abs/2310.06770.

Sina Khajehabdollahi, Roxana Zeraati, Emmanouil Giannakakis, Tim Jakob Schäfer, Georg Martius, and Anna Levina. Emergent mechanisms for long timescales depend on training curriculum and affect performance in memory tasks, 2024. URL https://arxiv.org/abs/2309.12927.

Langchain. Langchain-ai/langgraph: Build resilient language agents as graphs., 2024. URL https://github.com/langchain-ai/langgraph.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive nlp tasks, 2021. URL https://arxiv.org/abs/2005.11401.

Robert Long, Eric Gonzalez, and Harrison Fuller. Generalization of medical large language models through cross-domain weak supervision, 2025. URL https://arxiv.org/abs/2502.00832.

David Lopez-Paz and Marc'Aurelio Ranzato. Gradient episodic memory for continual learning, 2022. URL https://arxiv.org/abs/1706.08840.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. Codexglue: A machine learning benchmark dataset for code understanding and generation, 2021. URL https://arxiv.org/abs/2102.04664.

Martial Mermillod, Aurélia Bugaiska, and Patrick Bonin. The stability-plasticity dilemma: Investigating the continuum from catastrophic forgetting to age-limited learning effects, 2013.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. Codegen: An open large language model for code with multi-turn program synthesis, 2023. URL https://arxiv.org/abs/2203.13474.

Zach Nussbaum, John X. Morris, Brandon Duderstadt, and Andriy Mulyar. Nomic embed: Training a reproducible long context text embedder, 2025. URL https://arxiv.org/abs/2402.01613.

OpenAI and Princeton-NLP. Introducing swe-bench verified, 2024. URL https://openai.com/index/introducing-swe-bench-verified.

German I. Parisi, Ronald Kemker, Jose L. Part, Christopher Kanan, and Stefan Wermter. Continual lifelong learning with neural networks: A review. *Neural Networks*, 113:54–71, 2019. ISSN 0893-6080. doi: https://doi.org/10.1016/j.neunet.2019.01.012. URL https://www.sciencedirect.com/science/article/pii/S0893608019300231.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

Taiwei Shi, Yiyang Wu, Linxin Song, Tianyi Zhou, and Jieyu Zhao. Efficient reinforcement finetuning via adaptive curriculum learning, 2025. URL https://arxiv.org/abs/2504.05520.

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

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models, 2023. URL https://arxiv.org/abs/2210.03629.

A Appendix: Prompt Template for Patch Generation (SWE-Bench Harness Experiments)

Adapted from Jimenez et al. [2024], the following prompt template was used as the basis for generating patches in the experiments described in Section 5. Placeholders like `problem_statement`, `retrieved_context`, etc., were filled dynamically based on the task and experimental condition (e.g., `retrieved_context` would contain information from past tasks for the memory-enabled condition). The `patch_example_content` was a static example of a diff.

```
You will be provided with a partial code base and an issue statement explaining
a problem to resolve.
The goal is to generate a code patch in the **unified diff format** that
resolves the issue.

<issue>
{problem_statement}
</issue>

Relevant context from the repository ({repo} at commit {base_commit}):
<code>
{retrieved_context}

**Hints (if any from the original issue):**
{hints_text}

**Files to consider (based on gold solution, try to identify which files to modify):
{text_files}
</code>

Here is an example of a patch file. It consists of changes to the codebase.
It specifies the file names, the line numbers of each change, and the removed
and added lines. A single patch file can contain changes to multiple files.

<patch>
--- a/file.py
+++ b/file.py
@@ -1,27 +1,35 @@
 def euclidean(a, b):
-    while b:
-        a, b = b, a % b
-    return a
+    if b == 0:
+        return a
+    return euclidean(b, a % b)


 def bresenham(x0, y0, x1, y1):
     points = []
     dx = abs(x1 - x0)
     dy = abs(y1 - y0)
-    sx = 1 if x0 < x1 else -1
-    sy = 1 if y0 < y1 else -1
-    err = dx - dy
+    x, y = x0, y0
+    sx = -1 if x0 > x1 else 1
+    sy = -1 if y0 > y1 else 1

-    while True:
-        points.append((x0, y0))
```

```
-        if x0 == x1 and y0 == y1:
-            break
-        e2 = 2 * err
-        if e2 > -dy:
+    if dx > dy:
+        err = dx / 2.0
+        while x != x1:
+            points.append((x, y))
             err -= dy
-            x0 += sx
-        if e2 < dx:
-            err += dx
-            y0 += sy
+            if err < 0:
+                y += sy
+                err += dx
+            x += sx
+    else:
+        err = dy / 2.0
+        while y != y1:
+            points.append((x, y))
+            err -= dx
+            if err < 0:
+                x += sx
+                err += dy
+            y += sy

+    points.append((x, y))
     return points
</patch>
```

I need you to solve the provded issue by generating a single patch file that I can apply directly to this repository using git apply. Please respond with a single patch file in the format shown above.

Respond below: