

MemGovern: Enhancing Code Agents through Learning from Governed Human Experiences

Qihao Wang^{1*}, Ziming Cheng^{2*}, Shuo Zhang^{10*}, Fan Liu^{3*},
Rui Xu⁴, Heng Lian^{5,10}, Kunyi Wang^{6,10}, Xiaoming Yu⁷, Jianghao Yin³, Sen Hu^{8,10}, Yue Hu¹,
Shaolei Zhang^{9†}, Yanbing Liu^{1†}, Ronghao Chen^{10,11†}, Huacan Wang^{1,10†}

¹University of Chinese Academy of Sciences ²National University of Singapore

³East China Normal University ⁴Fudan University ⁵Xidian University

⁶The University of British Columbia

⁷The Hong Kong University of Science and Technology (Guangzhou)

⁸Peking University ⁹Renmin University of China ¹⁰QuantaAlpha

¹¹International Academy of Phronesis Medicine (Guangdong)

*Equal Contribution †Corresponding Authors

Abstract

While autonomous software engineering (SWE) agents are reshaping programming paradigms, they do not yet systematically exploit the historical human repair experience recorded on platforms such as GitHub. Web search can expose external information, and pretrained models may encode parts of this experience, but real-world issue-tracking data remain unstructured, fragmented, and noisy. In this paper, we introduce MemGovern, a framework designed to govern and transform raw GitHub data into actionable experiential memory for agents. MemGovern employs experience governance to convert human experience into agent-friendly experience cards and introduces an agentic experience search strategy that enables logic-driven retrieval of human expertise. By producing 135K governed experience cards, MemGovern improves resolution rates on SWE-bench Verified by 5.1% on average across eight LLM backbones. As a plug-in module, MemGovern requires no model-specific adaptation.

1 Introduction

With the remarkable progress of large language models (LLMs) in code understanding and generation (Chen et al., 2021; Li et al., 2023; Rozière et al., 2024; OpenAI et al., 2024), autonomous software engineering agents (a.k.a. code agents) are reshaping the paradigm of programming (Hong et al., 2024; Qian et al., 2024; Wu et al., 2023; Wang et al., 2025). Among such tasks, autonomously fixing bugs in GitHub repositories has emerged as a key benchmark for evaluating code-agent capabilities and has become a central research focus (Jimenez et al., 2024; Yang et al., 2024; Xia et al., 2025).

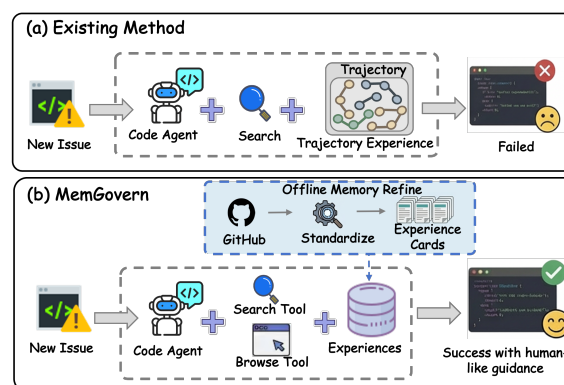


Figure 1: Comparison of MemGovern with existing methods. MemGovern learns from human experience by governing raw data into agent-friendly memories.

In real-world software maintenance, developers spend substantial effort searching and navigating existing artifacts rather than reasoning from a blank slate (Ko et al., 2006; Sadowski et al., 2015). For complex issues, prior issue and pull-request discussions can expose related failures and fixes across projects (Tan et al., 2024; Rahman and Roy, 2014). The associated commits and patches also encode recurring repair patterns that can be learned and reused (Pan et al., 2009; Kim et al., 2013). Although modern agents may use web search and pretrained models already encode some public development history, these channels do not organize noisy Issue–PR–Patch records into directly reusable repair knowledge. An advanced code agent should exploit such open-world experience through persistent and structured memory that deepens its reasoning on difficult bugs (Shinn et al., 2023; Packer et al., 2024; Gao et al., 2026; Zhang et al., 2025).

Although GitHub holds a vast repository of human experience, converting it into agent-friendly knowledge remains challenging. First, raw is-

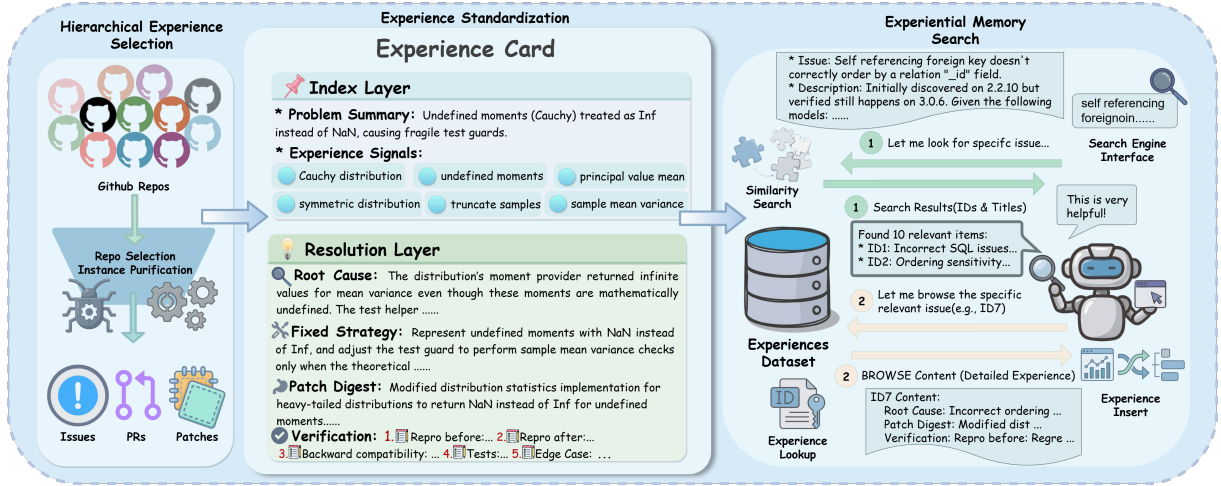


Figure 2: Architecture of MemGovern. MemGovern selects raw human experiences from GitHub and standardizes them into experience cards, enabling agents to utilize them through agentic experience search.

sue and pull-request discussions contain large amounts of unstructured and fragmented information, such as social exchanges and procedural communication, which can obscure technical insights (Kalliamvakou et al., 2014). Second, variations in terminology, module organization, and coding styles across projects hinder the standardized transfer of repair knowledge (Allamanis et al., 2019). Consequently, most retrieval-based code systems focus on context from the target repository rather than curated human repair experience across repositories (Zhang et al., 2023, 2024).

To address these challenges, we propose MemGovern, a governance framework that transforms raw GitHub data into agent-usable experiential memory. Inspired by work on persistent agent memory and structured software-knowledge retrieval (Packer et al., 2024; Yang et al., 2026; Liu et al., 2023), MemGovern adopts a key design principle: retrieval semantics should be decoupled from reasoning logic. We instantiate this principle through a dual-layer experience card, where the Index Layer captures symptom-level signals for cross-repository retrieval and the Resolution Layer preserves transferable repair strategies for agent reasoning (Deng et al., 2026).

Specifically, MemGovern comprises two components: (1) *experience governance*, a pipeline that selects, standardizes, and quality-controls raw GitHub data into governed experience cards; and (2) *agentic experience search* (Li et al., 2026), a progressive search-then-browse mechanism that interleaves reasoning with iterative information-seeking actions. Notably, we find that *how* experience is utilized matters more than *whether* it

is available—naïve RAG injection can even hurt performance, while our agentic search yields consistent improvements across models.

Experimental results on SWE-bench Verified (Jimenez et al., 2024) demonstrate that MemGovern improves code agents’ resolution rates by 5.1% on average across eight LLM backbones, using 135K governed experience cards. As a plug-in module, MemGovern requires no model-specific adaptation.

2 Related Work

Memory Construction for Agents. Building effective memory systems requires ensuring data quality and standardization. Early approaches mined fix patterns from commits or patches (Pan et al., 2009; Kim et al., 2013) but captured only syntactic transformations. Recent agent memory systems have explored structured experience organization. ExpeRepair (Mu et al., 2026) maintains dual memory banks separating episodic demonstrations from semantic insights, while SWE-Exp (Chen et al., 2026) extracts multi-level experiences from agent trajectories. More broadly, reflection and persistent-memory systems reuse prior interactions to guide later decisions (Shinn et al., 2023; Packer et al., 2024), while recent work studies structured and self-evolving agent memories (Gao et al., 2026; Zhang et al., 2025; Deng et al., 2026). ReMe manages procedural memory through experience distillation, context-adaptive reuse, and utility-based refinement (Cao et al., 2026); other studies show that regulating memory addition and deletion can mitigate error propagation and misaligned experience replay (Xiong et al., 2026), and

that memory’s effect can depend on the agent architecture, trading procedural reliability against search diversity (Zhao et al., 2026). In software domains, value-driven retrieval and cross-task experience sharing have also been explored for NPU kernel synthesis (Zheng et al., 2026).

Code Agents. Recent advances in LLMs have driven autonomous code agents for automated software engineering. SWE-agent (Yang et al., 2024) pioneered Agent-Computer Interfaces with specialized tools for LLM-based code navigation, while AutoCodeRover (Zhang et al., 2024) enhanced fault localization through syntax-tree representations and spectrum-based techniques. Agentless (Xia et al., 2025) demonstrated that simpler pipeline approaches can match agent performance with lower costs, and OpenHands (Wang et al., 2025) provides a general platform for software-development agents. Training-based methods have also emerged: SWE-Fixer (Xie et al., 2025) compiled 110K instances for fine-tuning open-source repair models, and BugPilot (Sonwane et al., 2025) generated synthetic bugs for efficient skill learning. Recent evaluation has also extended software agents to complex, large-scale systems such as Linux-kernel fault localization (Zhou et al., 2026). Existing approaches largely operate within single repositories or rely on self-generated experiences, overlooking the vast corpus of cross-repository human debugging expertise in GitHub’s collaborative ecosystem.

In contrast, MemGovern addresses these limitations through comprehensive governance that curates cross-repository human experiences from GitHub at scale. Our approach introduces multi-stage curation to filter noisy discussions, content purification to standardize heterogeneous issue reports, and a dual-layer protocol that separates retrieval signals from actionable repair logic, enabling effective learning from the collective debugging knowledge embedded in real-world collaborative development.

3 MemGovern

We propose MemGovern, a unified framework that governs raw human experiences into agent-usable experiential memory. The core design principle is that effective agent memory requires a principled separation of concerns: retrieval signals must be decoupled from reasoning logic to enable cross-repository generalization. As illustrated in Figure 2, MemGovern comprises two components—

experience governance (Section 3.1) that constructs governed experience cards, and *agentic experience search* (Section 3.2) that enables agents to progressively retrieve and apply relevant experience.

3.1 Experience Governance

The efficacy of experience-driven learning hinges on the quality of the underlying data. While GitHub offers a vast repository of debugging expertise, this open-world knowledge is inherently unstructured and noisy. Historical records are often laden with social chatter, ambiguous process descriptions, and non-standardized terminology, creating a "semantic gap" that hinders direct agentic retrieval. To bridge this gap, MemGovern involves *Experience Governance*, a systematic pipeline that distills chaotic raw data into structured, high-fidelity *experience cards*. This process prioritizes information density and experience reliability through three stages: (1) *Experience Selection*, which filters low-signal noise at both the repository and instance levels; (2) *Standardization*, which reconstructs raw data through a Unified repair-experience protocol; and (3) *Quality Control*, a checklist-based mechanism to ensure memory fidelity.

3.1.1 Hierarchical Experience Selection

The primary challenge in building a reliable experiential memory is the high quality variance, ranging from unmaintained hobby projects to knowledge-dense issues. Integrating such low-signal data risks memory pollution. To address this, MemGovern uses a two-tiered selection strategy that ensures both source authority and instance completeness.

Repository Selection. To ensure the agent learns from active and high-quality software engineering practices, MemGovern filters for repositories exhibiting sustained maintenance. MemGovern selects top- M repositories based on a score that balances popularity (Stars S_r) with maintenance intensity (Issues I_r and Pull Requests P_r):

$$\text{Score}(r) = \lambda_s \log(1 + S_r) + \lambda_i \log(1 + I_r) + \lambda_p \log(1 + P_r), \quad (1)$$

where $\lambda_s, \lambda_i, \lambda_p$ represent weights for stars, issues, and PRs respectively. This ensures the source data reflects robust, active development flows.

Instance Purification. Within selected repositories, MemGovern rigorously filters (*Issue*, *PR*, *Patch*) triplets to retain only "closed-loop" repair records. A valid instance must

provide a complete chain of evidence: explicit linkage between the issue and merged code, a parsable diff, and diagnostic anchors (e.g., stack traces). Furthermore, to strip away social noise, MemGovern analyzes the comment threads and discard instances where the technical-content ratio falls below a threshold $\tau = 0.2$, preserving only threads rich in debugging logic.

3.1.2 Experience Standardization

Raw GitHub discussion threads are inherently verbose, often interleaving failure symptoms with implementation details, social exchanges, and repository-specific context. Such entanglement obscures the essential signals required for effective retrieval. To render these records actionable, MemGovern defines a *unified repair experience protocol* and standardizes selected instances accordingly.

The core philosophy of this protocol is to explicitly decouple retrieval semantics from reasoning logic. Concretely, MemGovern begins with *content purification*, where an LLM compresses the original comment stream by removing non-technical interactions (e.g., greetings, merge notifications) and redundant execution logs. The purified content is then reorganized into two functionally distinct layers within each experience card: the *Index Layer* and the *Resolution Layer*.

Index Layer. This layer captures the information available at an agent’s initial observation stage and serves as the primary retrieval interface. It includes a normalized Problem Summary and a set of generalizable Diagnostic Signals (e.g., exception types, error signatures, component-level tags). Repository-specific identifiers and incidental implementation details are intentionally removed to maximize semantic matchability across heterogeneous repositories.

Resolution Layer. This layer encapsulates the transferable repair knowledge distilled from human debugging processes. It contains the Root Cause analysis, an abstract Fix Strategy, and a concise Patch Digest. By isolating causal reasoning and procedural patterns from concrete code artifacts, this layer enables agents to reuse human repair logic beyond the original context. Formally, each standardized experience card E_i is represented as:

$$E_i = \langle \text{Index} = I_i, \text{Resolution} = R_i \rangle. \quad (2)$$

where I_i denotes the index layer with retrievable failure symptoms, and R_i denotes the resolution

layer with reusable repair logic. This structural decoupling allows agents to retrieve experiences based on symptom-level similarity encoded in the Index Layer, while executing repairs using abstract reasoning strategies encoded in the Resolution Layer, thereby enabling effective cross-repository generalization.

3.1.3 Checklist-Based Quality Control

Automated extraction pipelines can occasionally hallucinate details or miss key nuances. To guarantee the reliability of the memory bank, we introduce a checklist-based quality control mechanism that acts as a final gatekeeper before ingestion.

We employ an LLM as a structured evaluator to score each generated experience card against critical dimensions. Unlike simple filtering, this process incorporates a *Refine Loop*: if a card’s aggregate score falls below a threshold γ , the evaluator provides targeted feedback on the deficient dimensions. The extraction pipeline then regenerates only the problematic sections based on this feedback. This cycle repeats for a maximum of three iterations, ensuring that only experiences passing this rigorous quality gate are indexed. Consequently, MemGovern operates on a foundation of verifiable expert knowledge rather than noisy raw data.

3.2 Experiential Memory Search

Constructing high-quality experiential memory is only half the challenge; the agent must also retrieve and apply it effectively. Unlike standard RAG, which relies on single-shot context injection, real-world debugging is a dynamic process of hypothesis formulation and validation. MemGovern addresses this through an agentic experience search mechanism with a dual-primitive interface.

3.2.1 Dual-Primitive Interface

To balance the need for broad semantic discovery with the cost of detailed context processing, MemGovern exposes the experiential memory to agent through two primitives: *Searching* and *Browsing*.

Searching. The Searching tool serves as a high-throughput filter that enables the agent to scan the memory index using a query. Given a task-specific query q , the tool traverses the index to identify relevant experience cards. Retrieval is based on the Index Layer of each card, ensuring that matches are driven by symptom-level similarity rather than implementation details. The relevance of an experience card E_i is computed via cosine similarity in

the embedding space:

$$\text{sim}(q, I_i) = \frac{\phi(q) \cdot \phi(I_i)}{\|\phi(q)\| \|\phi(I_i)\|} \quad (3)$$

where $\phi(\cdot)$ is embedding function and I_i represents index layer of card E_i . The tool returns a ranked list of candidates $\{(E^{(k)}, \text{sim}(q, I^{(k)}))\}_{k=1}^K$, allowing the agent to assess relevance before proceeding to a deeper inspection.

Browsing. After promising candidates are identified, the Browsing tool grants access to the detailed resolution layer of a selected card. It offers high-precision retrieval without overwhelming the agent with raw details. This design ensures that once a relevant case is identified, the agent receives a clear blueprint of *what* changed and *why*, enabling direct transfer of human expertise to the agent.

3.2.2 Progressive Agentic Search

Building upon these primitives, we introduce a *progressive agentic search* mechanism that adapts dynamically to the evolving problem-solving state. Unlike conventional approaches that follow a rigid, predefined pipeline, the agent continuously reassesses its informational needs and autonomously chooses searching and browsing actions.

Query Formulation and Retrieval. The process begins with problem representation. The agent analyzes the current issue to extract diagnostic keywords, such as failure symptoms, failing test names, stack traces, and relevant module identifiers, to construct an initial query Q . It then invokes the *Searching* tool to retrieve semantically related candidates. Rather than indiscriminately consuming all results, the agent autonomously assesses the returned metadata and selects the most promising subset for deeper inspection. When the returned candidates are not sufficiently relevant, the agent can revise the query by incorporating additional observed symptoms and re-invoke *Searching*, enabling multi-round retrieval without prematurely committing to noisy details.

Selective Browsing and Analogical Transfer. For the selected candidates, the agent employs the *Browsing* tool to access their resolution layers. The core challenge here is *analogical transfer*: the agent must map the historical solution to the current repository’s context. By synthesizing the browsed evidence, the agent induces a transfer triplet: *Root Cause Pattern* $\rightarrow$ *Modification Logic* $\rightarrow$ *Validation Strategy*. For instance, if a retrieved experience

suggests "adding a boundary check for null inputs in the parser", the agent maps this abstract strategy to the specific variable names and API versions of the current codebase. This results in a concrete, actionable repair plan involving specific file locations and code edits. Overall, progressive agentic search enables the agent to autonomously and deeply acquire and exploit experiential memory.

4 Experiments

4.1 Experimental Setup

We evaluate our method on SWE-bench Verified. For a comprehensive and fair evaluation, we compare MemGovern against SWE-Agent and several strong baselines. Due to space limitations, we provide a detailed introduction to the benchmark and baselines in Appendix A.

4.2 Configuration

During the bug-fix experience governance process, we collected active, well-maintained open-source repositories from GitHub with more than 100 stars. After verification and data cleaning, we obtained approximately 150K Issue–PR–Patch triplets. These triplets were then processed by our experience governance pipeline using GPT-5.1 (medium reasoning), as described in Section 3.1. After de-duplication, the final experience card collection contains 135K items. To prevent data leakage, we perform exact matching on issue IDs to ensure that none of the 500 SWE-bench Verified issues (or their associated PRs/patches) appear in our experiential memory. The experience governance pipeline is a one-time offline process, independent of any benchmark-specific data.

Building on the SWE-Agent framework, we implemented two tools: an Experience Search tool and an Experience Browse tool. Both tools are integrated into SWE-Agent’s default toolset, together with the anthropic style code editor and patch submission module. The Experience Search tool takes a search query and a hyperparameter, top-k, which controls the number of experience-card previews returned (default to 10 to balance context length and information gain). The Experience Browse tool takes the unique index of an experience card and returns the full bug-fix experience.

4.3 Main Results

Table 1 presents the main results on SWE-bench Verified, comparing MemGovern with the SWE-

Method	LLM	Resolved Rate (%)	Δ (%)	Avg. Tokens (M)	Avg. Cost (\$)
Other baselines					
AutoCodeRover	GPT-4o	28.8	-	-	-
CodeAct	GPT-4o	30.0	-	-	-
SWESynInfer	GPT-4o	31.8	-	-	-
mini-SWE-agent	DeepSeek-V3.2-Reasoner	60.0	-	-	-
mini-SWE-agent	Claude-4-Opus	67.6	-	-	-
mini-SWE-agent	GPT-5.2	69.0	-	-	-
SWE-Agent vs. MemGovern across LLMs					
SWE-Agent	Claude-4-Sonnet	66.6	-	2.29	6.94
MemGovern		69.8	+3.2	2.35	7.27
SWE-Agent	GPT-5-Medium	65.0	-	0.78	1.14
MemGovern		67.4	+2.4	0.91	1.33
SWE-Agent	DeepSeek-V3.1T	62.8	-	0.89	0.18
MemGovern		65.8	+3.0	0.95	0.19
SWE-Agent	Qwen3-235B	47.2	-	1.17	0.09
MemGovern		55.4	+8.2	1.42	0.10
SWE-Agent	Kimi-K2-Instruct	43.8	-	0.57	0.26
MemGovern		51.8	+8.0	1.14	0.53
SWE-Agent	Qwen3-Coder-30B	48.0	-	0.92	0.07
MemGovern		51.4	+3.4	0.99	0.07
SWE-Agent	GPT-4o	23.2	-	0.73	1.38
MemGovern		32.6	+9.4	0.97	1.84
SWE-Agent	GPT-4o-Mini	14.0	-	0.99	0.14
MemGovern		17.2	+3.2	1.37	0.20

Table 1: Comparisons with prior agents. Δ is computed against SWE-Agent under the same backbone.

Agent baseline across eight LLMs. MemGovern consistently outperforms the baseline across all models, demonstrating robust and model-agnostic improvements. The gains are especially pronounced for models with weaker initial performance, while even the stronger models benefit noticeably, highlighting MemGovern’s ability to enhance task-solving capabilities across a diverse range of LLMs. Although MemGovern introduces some additional token overhead due to its memory mechanism, the average improvement of 5.1% across all backbones demonstrates a favorable effectiveness–efficiency trade-off (see Section 5.7 for detailed wall-clock analysis).

5 Analysis

5.1 Ablation of Experience Governance

To study the effectiveness of experience governance in MemGovern, we conduct ablation studies on two fundamental characteristics of governed experience: *size* and *quality*.

Effect of Experiential Memory Size. We evaluate MemGovern under 10%–100% of the full experiential memory (135K cards), as shown in

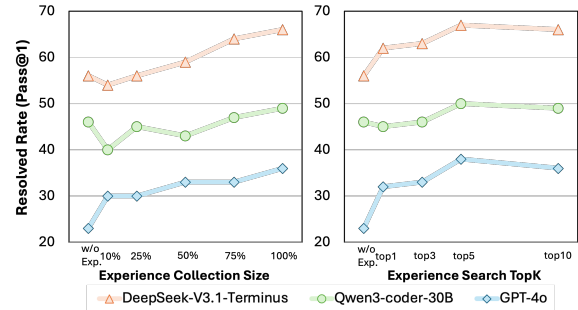


Figure 3: Results of MemGovern under various experiential memory size and MemGovern under various retrieval sizes.

Figure 3. The results show that enlarging the memory steadily improves resolution performance across LLMs, with monotonic rather than abrupt gains. This improvement can be attributed to a higher probability of retrieving relevant and transferable experience cards when the memory coverage increases. Importantly, the gains are monotonic rather than abrupt, indicating that MemGovern does not rely on a small subset of exceptional examples but instead benefits cumulatively from diverse, governed experiences. Overall, MemGovern

LLM	Strategy	Exp	Resolved Rate	Δ
DeepSeek-V3.1-T	Base Agent	×	62.8	-
	w/ RAG	✓	64.4	+1.6
	w/ Ag. RAG	✓	63.4	+0.6
	w/ Ag. Search	✓	65.8	+3.0
Qwen3-Coder-30B	Base Agent	×	48.0	-
	w/ RAG	✓	46.8	-1.2
	w/ Ag. RAG	✓	48.6	+0.6
	w/ Ag. Search	✓	51.4	+3.4
GPT-4o	Base Agent	×	23.2	-
	w/ RAG	✓	31.2	+8.0
	w/ Ag. RAG	✓	31.2	+8.0
	w/ Ag. Search	✓	32.6	+9.4

Table 2: Comparison of experience usage strategies within our agent. All memory-enabled variants retrieve from the experiential memory (“Exp”). “Ag.” denotes “Agentic,” indicating that retrieval or usage is performed dynamically during multi-step reasoning. Base Agent represents the original SWE-Agent.

effectively leverages large-scale experience aggregation, validating the necessity of constructing a sizable experiential memory through systematic governance.

Effect of Experiential Memory Quality. To assess whether the gains arise from governed experience rather than raw data, we compare MemGovern using the same memory size but different contents: (i) raw *PR+Patch* records without standardization or quality filtering, and (ii) fully governed experience cards constructed by MemGovern through the governance pipeline (see Section 3.1). As reported in Figure 4, raw records can occasionally provide partial benefits, but their effectiveness is unstable and model-dependent, likely due to noise, verbosity, and weak alignment with agent retrieval semantics. In contrast, governed experience cards consistently yield stronger and more stable improvements across models. These findings confirm that MemGovern’s improvements are driven by experience governance rather than sheer data exposure, empirically validating the effectiveness of standardization and quality control in constructing a high-fidelity experiential memory.

5.2 Superiority of Agentic Search

To verify that the gains stem from agentic search rather than naive experience injection, we compare three strategies under the same experiential memory, as shown in Table 2: (i) **RAG**, which performs a single retrieval and directly injects all retrieved experiences into the context; (ii) **Agentic RAG**,

which allows adaptive retrieval during debugging but still follows a retrieve-and-inject paradigm; and (iii) **Agentic Search** (Ours), which decouples candidate discovery from evidence usage through selective browsing.

Agentic Search consistently outperforms both RAG variants across all backbones, including DeepSeek-V3.1-Terminus, Qwen3-Coder-30B, and GPT-4o. This improvement stems from the agent’s strategic management of searching depth and breadth. By decoupling **searching breadth** (broad candidate discovery) from **browsing depth** (selective evidence grounding and logic extraction), the agent can explore a wide range of candidate experiences without being overwhelmed by irrelevant information, while simultaneously diving deeply into the most promising ones to extract actionable insights. This enables reasoning over multiple experiences in a structured and iterative manner, ensuring that only contextually relevant knowledge is integrated with the current debugging state. In contrast, both RAG variants conflate breadth with depth, making them sensitive to noise—weakly related experiences injected directly into the context often disrupt the generation process. Notably, naive RAG even hurts performance on Qwen3-Coder-30B (−1.2%), confirming that *how* experience is utilized matters more than *whether* it is available.

5.3 Effect of Various Retrieval Sizes

We study how the number of retrieved candidates (Top- K) affects performance, as reported in Figure 3. The results reveal a clear and consistent trend across different backbone models. When Top- K is small, increasing the retrieval size leads to steady performance improvements, indicating that a larger candidate pool increases the likelihood of surfacing actionable and relevant experiences. However, beyond a moderate value of K , the gains gradually diminish and eventually plateau. This behavior aligns well with the design of agentic experience search: at small K , retrieval coverage is the primary bottleneck, and expanding the candidate set helps the agent discover useful analogies; once sufficient coverage is achieved, further enlarging the candidate pool yields limited benefits, as the agent already has access to representative experience patterns and additional candidates tend to be redundant. Importantly, performance does not degrade at larger K , confirming that the selective browsing mechanism effectively filters irrelevant experiences and prevents context overload. These

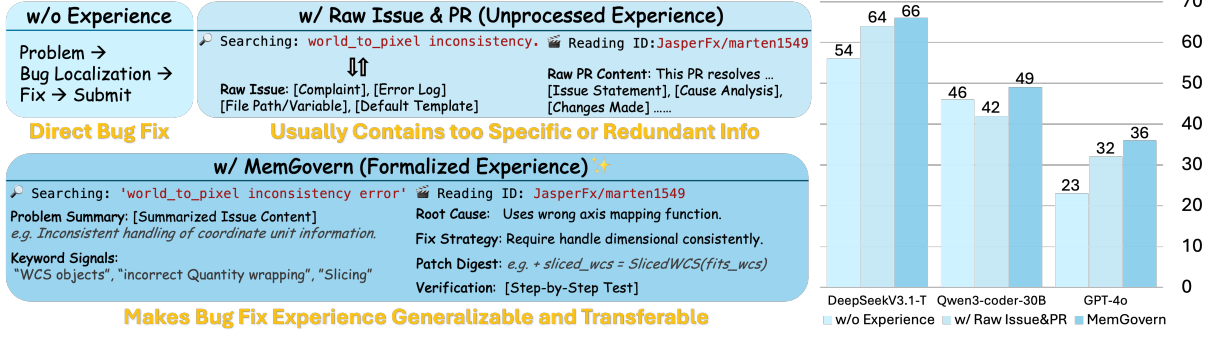


Figure 4: Comparison of using raw unprocessed experience and MemGovern’s experience.

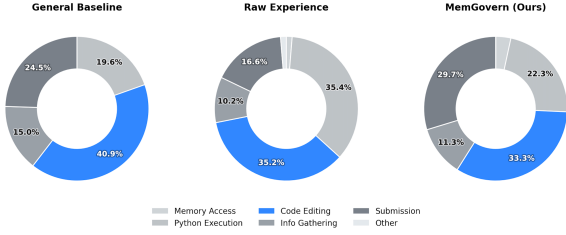


Figure 5: Action composition of the agent across different experience settings. We report the fraction of steps spent on code editing versus other actions.

findings further support the effectiveness of the decoupled retrieval-and-browsing design, showing that agentic search can safely leverage broader retrieval without sacrificing reliability.

5.4 Effect of Repository Selection

To validate our repository selection strategy, we conduct a human evaluation on 100 randomly sampled GitHub repositories with $S_r \geq 100$ and similar Issue volumes. We then construct four Top-10 subsets ranked by *Star only*, *Issue only*, *Pull Request only*, and *Our method* (Section 3.1.1). Two experienced software engineers independently assess whether issues in each subset can be distilled into transferable, reusable experiences under blinded ranking labels. As shown in Figure 6 in Appendix, our method yields the highest proportion of issues deemed reusable as experience, suggesting that combining popularity and maintenance signals selects repositories with denser, higher-quality experience candidates and strengthens the memory. Appendix D reports the sampling procedure, labeling rubric, and inter-annotator agreement.

5.5 Evidence of Cross-Repository Transfer

We parse every experience-browse call for four backbones and compare the card’s source repository with the test repository. Cross-repository cards account for 31.1%–46.7% of all browsed

cards. Across all four backbones, instances that browse at least one cross-repository card also have higher observed resolution rates than instances that browse only same-repository cards; on DeepSeek-V3.1T, the gains over the SWE-Agent baseline are +3.6 percentage points and +1.2 points, respectively. Appendix F provides the full counts and rates. Because this analysis is observational, a controlled intervention that restricts memory to same-repository-only or cross-repository-only cards remains future work.

5.6 Can Agent Learn from Experience?

We analyze agent behaviors by quantifying the proportion of each operation category (Figure 5). The results indicate that MemGovern improves both success rate and efficiency by reducing unguided exploration and encouraging effective self-testing.

Specifically, Info Gathering drops from 15.0% to 11.3% under MemGovern, indicating that experience cards offer targeted navigational guidance and reduce extensive repository exploration. This improvement aligns with a more balanced editing–execution strategy: the baseline agent often edits prematurely before adequately reproducing or localizing faults (Code Editing: 40.9%; Code Execution: 19.6%), raising the risk of faulty fixes. With MemGovern, the agent shifts focus toward self-testing (Code Execution: 22.3%) and curtails unnecessary edits (Code Editing: 33.3%) by leveraging root-cause and verification cues from experience cards, resulting in more reliable repairs.

Although Raw Experience (unprocessed GitHub PR and patches) provides useful cues, it incurs a substantially higher verification overhead (35.4% Code Execution), revealing that unrefined experience introduces noise and leads to low-quality fixes. MemGovern mitigates this issue through knowledge governance, maintaining effective infor-

Model	Base (s)	Ours (s)	Δ Time	Δ Resolve
DS-V3.1T	300.0	320.3	+6.8%	+3.0%
Qwen3-235B	180.0	218.6	+21.4%	+8.2%
Qwen3-30B	90.0	96.9	+7.7%	+3.4%

Table 3: Wall-clock time per task. Tool overhead accounts for <1% of the incremental time.

mation while compressing irrelevant contexts.

5.7 Cost and Latency Analysis

Beyond success rates, we analyze the wall-clock overhead introduced by MemGovern. Table 3 reports the end-to-end time per task for three representative backbones. The tool-level overhead (Search: 64 ms/call, Browse: 2.6 ms/call) is negligible; the dominant source of extra latency is LLM inference for processing retrieved experience. Overall, MemGovern improves resolve rates by +3.0% to +8.2% with a bounded wall-clock increase of +6.8% to +21.4%, which we consider a practical trade-off. Browse use is sparse (0.21–0.73 calls per instance); Appendix F reports tool-call counts and the browse-count distribution.

5.8 Case Study

We conducted a case study in Appendix B to examine the specific improvements introduced by MemGovern. To highlight different aspects of its performance, we designed three types of comparisons. First, we compared agents with experience to those without, exploring how experience influences agent. Second, we contrasted the traditional RAG with our agentic search, demonstrating the impact of MemGovern’s agentic experience search. Third, we compared raw experiences with MemGovern’s governed experiences, showing how agent-friendly memories enhance agent performance.

6 Conclusion

In this paper, we propose MemGovern, a framework for governing raw GitHub data into agent-usable experiential memory, along with an agentic experience search method. MemGovern constructs 135K experience cards and improves resolution rates on SWE-bench Verified by 5.1% on average across eight LLM backbones. Our analysis shows that the gains stem from experience governance and agentic search rather than mere data exposure, and we identify specific conditions under which experience cards help or distract the agent.

Limitations

MemGovern enables agents to learn from governed human experiences for improved bug-fixing performance. A limitation is the additional tokens generated when searching the memory during agent execution. Our wall-clock analysis (Section 5.7) shows that tool-level overhead is negligible and the dominant cost comes from LLM inference, with a total wall-clock increase of 6.8%–21.4% depending on the backbone. The evaluation is restricted to SWE-bench Verified and is consequently centered on its Python repositories; gains across other languages, task distributions, benchmarks, and production-style agents remain to be established. Exact issue-ID exclusion prevents direct overlap with the 500 test issues but does not constitute a complete semantic near-duplicate audit. The repository-selection study uses only two engineers, and the help-versus-distract taxonomy is a small, post-hoc error analysis. Future work will investigate stronger leakage audits, controlled same-versus cross-repository memory conditions, cross-benchmark evaluation, and more efficient memory compression.

References

- Miltiadis Allamanis, Earl T. Barr, Premkumar Devanbu, and Charles Sutton. 2019. [A Survey of Machine Learning for Big Code and Naturalness](#). *ACM Computing Surveys*, 51(4):1–37.
- Zouying Cao, Jiaji Deng, Li Yu, Weikang Zhou, Zhaoyang Liu, Bolin Ding, and Hai Zhao. 2026. [Remember Me, Refine Me: A Dynamic Procedural Memory Framework for Experience-Driven Agent Evolution](#). In *Findings of the Association for Computational Linguistics: ACL 2026*, pages 16803–16822, San Diego, California, United States. Association for Computational Linguistics.
- 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, and 39 others. 2021. [Evaluating Large Language Models Trained on Code](#). *Preprint*, arXiv:2107.03374.
- Silin Chen, Shaoxin Lin, Yuling Shi, Heng Lian, Xi-aodong Gu, Longfei Yun, Dong Chen, Lin Cao, Jiyang Liu, Nu Xia, and Qianxiang Wang. 2026. [SWE-Exp: Experience-Driven Software Issue Resolution](#). *Preprint*, arXiv:2507.23361.
- Yi-Xuan Deng, Xiaoqin Liu, Yi Zhang, Guo-Wei Yang, and Shuojin Yang. 2026. [Your Code Agent Can Grow](#)

- Alongside You with Structured Memory. *Preprint*, arXiv:2603.13258.
- Huan-ang Gao, Jiayi Geng, Wenyue Hua, Mengkang Hu, Xinzhe Juan, Hongzhang Liu, Shilong Liu, Jiahao Qiu, Xuan Qi, Yiran Wu, Hongru Wang, Han Xiao, Yuhang Zhou, Shaokun Zhang, Jiayi Zhang, Jinyu Xiang, Yixiong Fang, Qiwen Zhao, Dongrui Liu, and 8 others. 2026. [A Survey of Self-Evolving Agents: What, When, How, and Where to Evolve on the Path to Artificial Super Intelligence](#). *Preprint*, arXiv:2507.21046.
- Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiaowu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. 2024. [MetaGPT: Meta programming for a multi-agent collaborative framework](#). In *The Twelfth International Conference on Learning Representations*.
- Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. SWE-bench: Can language models resolve real-world github issues? In *International Conference on Learning Representations*, volume 2024.
- Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. German, and Daniela Damian. 2014. [The promises and perils of mining GitHub](#). In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 92–101, Hyderabad India. ACM.
- Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. [Automatic patch generation learned from human-written patches](#). In *2013 35th International Conference on Software Engineering (ICSE)*, pages 802–811, San Francisco, CA, USA. IEEE.
- Amy J. Ko, Brad A. Myers, Michael J. Coblenz, and Htet Htet Aung. 2006. [An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks](#). *IEEE Transactions on Software Engineering*, 32(12):971–987.
- Jian Li, Yizhang Jin, Dongqi Liu, Hang Ding, Jiafu Wu, Dongsheng Chen, Yunhang Shen, Yulei Qin, Ying Tai, Chengjie Wang, Xiaotong Yuan, and Yabiao Wang. 2026. [SE-Search: Self-Evolving Search Agent via Memory and Dense Reward](#). *Preprint*, arXiv:2603.03293.
- Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, and 48 others. 2023. [StarCoder: May the source be with you!](#) *Preprint*, arXiv:2305.06161.
- Mingwei Liu, Simin Yu, Xin Peng, Xueying Du, Tianyong Yang, Huanjun Xu, and Gaoyang Zhang. 2023. [Knowledge Graph based Explainable Question Retrieval for Programming Tasks](#). In *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 123–135, Bogotá, Colombia. IEEE.
- 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](#). *Preprint*, arXiv:2411.00622.
- Fangwen Mu, Junjie Wang, Lin Shi, Song Wang, Shoubin Li, and Qing Wang. 2026. [ExpeRepair: Dual-Memory Enhanced LLM-Based Repository-Level Program Repair](#). *Proceedings of the ACM on Software Engineering*, 3(FSE):3954–3976.
- OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altmenschmidt, Sam Altman, Shyamal Anadkat, Red Avila, Igor Babuschkin, Suchir Balaji, Valerie Balcom, Paul Baltescu, Haiming Bao, Mohammad Bavarian, Jeff Belgum, and 262 others. 2024. [GPT-4 Technical Report](#). *Preprint*, arXiv:2303.08774.
- Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez. 2024. [MemGPT: Towards LLMs as Operating Systems](#). *Preprint*, arXiv:2310.08560.
- Kai Pan, Sunghun Kim, and E. James Whitehead. 2009. [Toward an understanding of bug fix patterns](#). *Empirical Software Engineering*, 14(3):286–315.
- Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, Juyuan Xu, Dahai Li, Zhiyuan Liu, and Maosong Sun. 2024. [ChatDev: Communicative Agents for Software Development](#). In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 15174–15186, Bangkok, Thailand. Association for Computational Linguistics.
- Mohammad Masudur Rahman and Chanchal K. Roy. 2014. [An insight into the pull requests of GitHub](#). In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 364–367, Hyderabad India. ACM.
- Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, and 7 others. 2024. [Code Llama: Open Foundation Models for Code](#). *Preprint*, arXiv:2308.12950.
- Caitlin Sadowski, Kathryn T. Stolee, and Sebastian Elbaum. 2015. [How developers search for code: A case](#)

- study. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 191–201, Bergamo Italy. ACM.
- Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. [Reflexion: Language agents with verbal reinforcement learning](#). In *Advances in Neural Information Processing Systems 36*, pages 8634–8652, New Orleans, Louisiana, USA. Neural Information Processing Systems Foundation, Inc. (NeurIPS).
- Atharv Sonwane, Isadora White, Hyunji Lee, Matheus Pereira, Lucas Caccia, Minseon Kim, Zhengyan Shi, Chinmay Singh, Alessandro Sordoni, Marc-Alexandre Côté, and Xingdi Yuan. 2025. [BugPilot: Complex Bug Generation for Efficient Learning of SWE Skills](#). *Preprint*, arXiv:2510.19898.
- Shin Hwei Tan, Ziqiang Li, and Lu Yan. 2024. [Cross-Fix: Resolution of GitHub issues via similar bugs recommendation](#). *Journal of Software: Evolution and Process*, 36(4):e2554.
- Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. 2024. Executable code actions elicit better LLM agents. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pages 50208–50232. PMLR.
- Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangu Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Daniel Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, and 5 others. 2025. OpenHands: An open platform for AI software developers as generalist agents. In *International Conference on Learning Representations*, volume 2025.
- Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W. White, Doug Burger, and Chi Wang. 2023. [AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation](#). *Preprint*, arXiv:2308.08155.
- Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2025. [Demystifying LLM-Based Software Engineering Agents](#). *Proceedings of the ACM on Software Engineering*, 2(FSE):801–824.
- 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](#). In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 1123–1139, Vienna, Austria. Association for Computational Linguistics.
- Zidi Xiong, Yuping Lin, Wenya Xie, Pengfei He, Zirui Liu, Jiliang Tang, Himabindu Lakkaraju, and Zhen Xiang. 2026. [How Memory Management Impacts](#) LLM Agents: An Empirical Study of Experience-Following Behavior. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 623–645, San Diego, California, United States. Association for Computational Linguistics.
- Chang Yang, Chuang Zhou, Yilin Xiao, Su Dong, Luyao Zhuang, Yujing Zhang, Zhu Wang, Zijin Hong, Zheng Yuan, Zhishang Xiang, Shengyuan Chen, Huachi Zhou, Qinggang Zhang, Ninghao Liu, Jinsong Su, Xinrun Wang, Yi Chang, and Xiao Huang. 2026. [Graph-based Agent Memory: Taxonomy, Techniques, and Applications](#). *Preprint*, arXiv:2602.05665.
- John Yang, Carlos Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. [SWE-agent: Agent-computer interfaces enable automated software engineering](#). In *Advances in Neural Information Processing Systems*, volume 37, pages 50528–50652. Curran Associates, Inc.
- Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. 2023. [RepoCoder: Repository-Level Code Completion Through Iterative Retrieval and Generation](#). In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 2471–2484, Singapore. Association for Computational Linguistics.
- Guibin Zhang, Muxin Fu, and Shuicheng Yan. 2025. [MemGen: Weaving Generative Latent Memory for Self-Evolving Agents](#). *Preprint*, arXiv:2509.24704.
- 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*, pages 1592–1604, Vienna Austria. ACM.
- Xinyu Zhao, Junpeng Wang, Yuzhong Chen, Menghai Pan, Chin-Chia Michael Yeh, Jiarui Sun, Yan Zheng, Mahashweta Das, and Tianlong Chen. 2026. [Demystify the Role of Memory in Machine Learning Engineering Agents](#). In *Findings of the Association for Computational Linguistics: ACL 2026*, pages 10811–10826, San Diego, California, United States. Association for Computational Linguistics.
- Yujie Zheng, Zhuo Li, Shengtao Zhang, Hanjing Wang, Junjie Sheng, Jiaqian Wang, Junchi Yan, Weinan Zhang, Ying Wen, Bo Tang, and Muning Wen. 2026. [Towards cold-start drafting and continual refining: A value-driven memory approach with application to NPU kernel synthesis](#). *Preprint*, arXiv:2603.10846.
- Zhenhao Zhou, Zhuochen Huang, Yike He, Chong Wang, Jiajun Wang, Yijian Wu, Xin Peng, and Yiling Lou. 2026. [Taming System Complexity: Demystifying Software Engineering Agents in Diagnosing Linux Kernel Faults](#). In *Proceedings of the 64th Annual Meeting of the Association for Computational*

A Experimental Setup

Benchmark. We evaluate our method on SWE-bench Verified (Jimenez et al., 2024), a human-validated subset of SWE-bench containing 500 real-world GitHub issues and focused on functional bug fixing in a controlled, self-contained setup. For each instance, the model is given only the natural-language issue description and the corresponding repository. Correctness is assessed by running developer-written unit tests against the generated patch, providing a consistent and rigorous evaluation of automated bug-fixing performance.

Baselines. For a comprehensive and fair evaluation, we compare MemGovern against strong baselines including SWE-Agent (Yang et al., 2024), AutoCodeRover (Zhang et al., 2024), CodeAct (Wang et al., 2024), and SWESynInfer (Ma et al., 2024). The comparison covers eight LLM backbones spanning open and proprietary models. The four open-source backbones are DeepSeek-V3.1-Terminus, Qwen3-Coder-30B, Kimi-K2-Instruct, and Qwen3-235B; the four proprietary backbones are GPT-4o, GPT-4o-Mini, Claude-4-Sonnet, and GPT-5-Medium. MemGovern is implemented as a plug-and-play module that can be integrated into existing agent scaffolds with minimal changes; unless otherwise stated, we adopt SWE-Agent as the shared backbone for subsequent experiments.

B Case Study Analysis

To provide a qualitative understanding of how MemGovern enhances autonomous software engineering, we present three case studies from the SWE-bench Verified dataset. These cases analyze the impact of accessing experience, the superiority of agentic search over static retrieval, and the necessity of experience governance.

B.1 Impact of Experiential Memory

We first examine how the presence of experience enables the agent to move beyond superficial fixes to robust solutions. Figure 7 illustrates a case involving a crash in Django’s `order_by()` function when `Meta.ordering` contains query expressions.

Without Experience (Baseline). The baseline agent, lacking historical context, relies on a “Mechanical Diagnosis.” It traces the indexing error in the stack trace and attempts to fix the crash by blindly enumerating input types. This results in a “Defensive Bypass” patch: it applies a superficial type check (`if not isinstance(field, str)`)

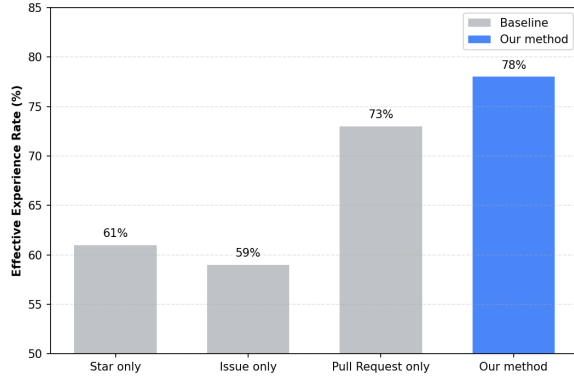


Figure 6: Human evaluation of repository selection. Proportion of issues judged reusable as transferable experience across different ranking rules.

to return the raw object. While this suppresses the immediate crash, it violates the expected return contract of the function, potentially causing silent failures downstream.

With Experience (MemGovern). MemGovern retrieves a relevant experience card and leverages its structured **Resolution Layer** to guide the repair. Specifically, the **Root Cause** field informs the agent that “OrderBy expressions were accepted... without explicit validation”, while the **Fix Strategy** provides a clear directive to “add explicit type checking” rather than bypassing the object. Leveraging these governed insights, MemGovern implements a semantic-aware fix that extracts the field name (`field.expression.name`), thereby preserving the API contract. This demonstrates how MemGovern’s structured schema—decoupling diagnostic signals from actionable resolution logic—provides the precise reasoning necessary to distinguish between a superficial symptom-fix and a correct repair.

B.2 Superiority of Agentic Experience Search

Standard RAG often suffers from semantic drift, where retrieved documents share surface keywords but differ in technical substance. Figure 8 compares MemGovern’s Agentic Search against a Standard RAG baseline on a Django issue where `number_format` renders small decimals in scientific notation.

Standard RAG. As defined in our comparisons, the RAG baseline performs a single retrieval before the repair process and directly injects all retrieved experiences into the context. In this case, the retrieval is driven by surface similarity with the

query “django numberformat decimal_pos,” fetching irrelevant experiences related to frontend CSS alignment (e.g., “text-align: right”) or locale-based separators. Influenced by this noise, the agent attempts a fix that aggressively mutates the number sign (`abs(number)`) early in the function, introducing a “Defensive Bypass” that breaks sign handling for other numeric types.

Agentic Search (MemGovern). In contrast, MemGovern employs a *Search-then-Browse* workflow. The agent first retrieves candidates and then uses the *Browsing* tool to inspect the logic. It successfully identifies a relevant case regarding “scientific notation cutoff” and rejects irrelevant frontend issues. By transferring the specific logic found in the experience—adding a threshold check, MemGovern produces a precise fix that resolves the issue without side effects. This highlights that the ability to iteratively browse and verify experience is crucial for filtering retrieval noise.

B.3 Necessity of Experience Governance

Finally, we analyze the importance of governing raw data into structured memory. Figure 9 depicts a complex issue involving `HttpResponse` handling `memoryview` objects from PostgreSQL.

Raw PR+Patch Records. When provided with raw PR+Patch records without standardization or quality filtering (Left), the agent is overwhelmed by noise. The retrieved raw data contains extraneous information about `BaseHandler` and `make_bytes` that is not central to the specific `memoryview` edge case. Confused by the verbose diffs and social commentary, the agent misclassifies the issue as a generic iterable problem. It produces a patch that intercepts `memoryview` but fails to handle `bytearray` correctly, causing the code to fall through to a generic loop that crashes when iterating over bytes.

Governed Experience Cards (MemGovern). MemGovern (Right) utilizes fully governed experience cards where noise has been stripped via *Content Purification*. The *Resolution Layer* clearly highlights the specific fix strategy: “Handle `memoryview` and `bytearray` objects.” Freed from the distraction of unrelated code changes in the original PR, the agent focuses on the core logic. It implements a focused patch that explicitly converts both `memoryview` and `bytearray` to raw bytes, ensuring correct serialization. This confirms that experience governance is essential to convert noisy open-source data into actionable, agent-friendly

memory.

C Experience Card Schema

A detailed explanation of MemGovern’s Experience Card Schema is described in Fig 10.

D Reproducibility Details

This section provides implementation details to support reproducibility.

Technical-Content Ratio. We use a locally deployed LLM (Qwen3-235B) to classify each comment in an issue thread as technical or non-technical (binary 0/1). The classification prompt is as follows:

```
Analyze this GitHub issue
comment. Determine if it
contains SUBSTANTIVE TECHNICAL
CONTENT directly related to
debugging, code analysis, error
diagnosis, or repair strategies
(excluding greetings, thanks,
"+1", merge notifications, or
procedural meta-discussions).
Output ONLY: 1 if technical,
0 if non-technical. Comment:
[COMMENT_TEXT]
```

The ratio is computed as the fraction of technical comments over total comments. The threshold $\tau = 0.2$ was heuristically selected through pilot studies; downstream Standardization and Quality Control (Sections 3.1.2–3.1.3) provide further refinement.

Quality Control Scoring Dimensions. Each experience card is evaluated along four dimensions, each scored 1–5:

- *Completeness*: Whether all required fields (Problem Summary, Signals, Root Cause, Fix Strategy, Patch Digest) are present without critical omissions.
- *Consistency*: Whether extracted content is factually faithful to the original Issue/PR discussion.
- *Usability*: Whether Index Layer signals and Resolution Layer strategies are logically aligned (e.g., “race condition” in Signals corresponds to concurrency handling in Fix Strategy).

Model	nDCG	Lat.	Price	Dim.
Qwen3-Emb-0.6B	0.751	22	\$0.010	1024
Qwen3-Emb-4B	0.802	27	\$0.020	2560
Jina Emb v3	0.766	85	\$0.045	1024
BAAI/bge-m3	0.753	29	\$0.010	1024

Table 4: Embedding model comparison. The columns report nDCG@10, latency in milliseconds, price per one million tokens, and embedding dimensionality.

- *Hallucination*: Whether the card contains fabricated details neither grounded in the source data nor attributable to common programming knowledge.

If the overall score falls below 80% (total < 16 out of 20) or any dimension scores below 3, a Refine Loop is triggered: the LLM evaluator generates targeted feedback on deficient dimensions, and only problematic sections are regenerated for up to three iterations.

Embedding Model. We use Qwen3-Embedding-0.6B for all encoding processes, selected based on the MTEB leaderboard by balancing retrieval quality, latency, and cost:

Deduplication. Deduplication operates at two levels:

- *Experience-level*: Each card is indexed by a unique identifier (repo + issue_id), preventing duplicate entries. We also perform exact matching on issue IDs against the 500 SWE-bench Verified instances to prevent data leakage.
- *Field-level*: The Signals field undergoes set-based exact matching after generation to remove duplicate keywords.

Repository-Selection Annotation Protocol.

The candidate pool contains 100 repositories with at least 100 stars and comparable issue volumes. Each of the four ranking rules—Star only, Issue only, Pull Request only, and our composite score—selects a Top-10 repository subset. We sample 100 closed Issue–PR–Patch records from each Top-10 subset, giving 400 method-conditioned record evaluations in total. Two software engineers independently label every record. The ranking method and ranking score are hidden, and records are shuffled before annotation.

A record is labeled *reusable* only if it (i) forms a complete Issue–PR–Patch chain, (ii) contains

enough technical evidence to identify a failure symptom and the corresponding repair, (iii) supports a repair strategy that can be stated beyond repository-specific identifiers, and (iv) includes a checkable validation signal. Records with missing links, purely procedural discussion, or fixes that cannot be interpreted beyond local identifiers are labeled *not reusable*. Cohen’s κ before adjudication is 0.71. After independent annotation, disagreements are resolved by discussion; Figure 6 reports the adjudicated labels.

Differential-Case Coding Protocol. The help-versus-distract study in Appendix E is selected mechanically from all cases with different outcomes between MemGovern and the SWE-Agent baseline; no outcome cases are subsampled. The same two engineers independently inspect the randomized traces and assign one of the predefined help or distract categories using the category definitions in Appendix E. System identifiers are hidden during the initial trace inspection, while actions, browsed cards, patches, and test outcomes remain visible because they are required for causal coding. Disagreements are adjudicated after the independent pass. We treat this taxonomy as exploratory error analysis rather than a controlled validation of the memory design.

Governing Model. The experience governance pipeline uses GPT-5.1 (medium reasoning). We conducted a pilot comparison on 100 sampled instances and found that GPT-5.1 produced experience cards with higher quality scores under our checklist-based evaluation, particularly in extracting accurate root cause analysis and abstract fix strategies. Cheaper models frequently required more refinement iterations. Since governance is a one-time offline process—the 135K cards are constructed once and reused across all downstream experiments—the governing model cost is a fixed upfront investment rather than a recurring runtime overhead.

Release Plan. Upon publication, we will release the experience-governance pipeline, the agentic-search tools, and the 135K governed experience cards, subject to the licenses and redistribution conditions of the underlying repositories. The release will include the prompts, configuration, and documentation required to reconstruct the memory and integrate the Search and Browse tools into an SWE-Agent-style scaffold.

D.1 Repository Data Validity Annotation Prompt

Table 5 presents the complete prompt used to assess whether Issue–Pull Request pairs provide transferable experience for training code agents.

E Error Analysis: When Does Experience Help vs. Distract?

To better understand *when* experience cards improve agent performance and *when* they mislead, we mechanically select all 63 differential cases: 39 where only MemGovern resolved the issue and 24 where only the baseline resolved it. MemGovern browsed card content in 28 of these cases: 16 MemGovern-only successes and 12 baseline-only successes. We identify four help patterns and four distract patterns under the double-coding protocol in Appendix D. Because the categories are post-hoc and the counts are small, we present them as exploratory error analysis rather than controlled validation.

E.1 When Experience Helps (16 browsed → resolved)

- **Guided Localization** (4 cases): The card specifies the exact fix site, enabling the agent to skip broad exploration. For example, in sphinx-8475, the card stated “extend except clause for TooManyRedirects”; the agent navigated directly and implemented a 3-step fix, while the baseline spent 15 extra steps exploring wrong modules.
- **Conceptual Reframing** (7 cases): The agent extracts an architectural principle from the card and independently derives a different fix. For instance, in django-11848, the card described a static 50/50 year cutoff; the agent recognized this was not directly applicable and instead derived a dynamic per-RFC7231 approach, using the card as a contrastive reference.
- **Bug Pattern Recognition** (4 cases): The agent recognizes a common bug pattern described in the card and maps it to the current code. In requests-2317, the card described a bytes/str confusion in another project; the agent identified the identical pattern and applied the same fix logic.
- **Multi-card Synthesis** (1 case): The agent combines complementary cards to form a

Prompt: Repository Data Validity Verification for Code-Agent Experience Construction

1. Task Background. We are constructing a large-scale dataset that teaches code agents to implement Pull Request (PR) changes from Issue descriptions. Two Top-20 repository sets selected by different strategies are provided. Determine whether each repository contains Issue–PR pairs with *experience-transfer value*, i.e., examples from which an agent can learn general programming logic or reusable problem-solving patterns.

2. Sampling Procedure. For each repository:

1. Open the Pull Requests page and filter by **Merged** status.
2. Randomly sample 10 merged PRs across different months or years; do not select only the 10 most recent PRs.
3. Prefer PRs explicitly associated with an Issue through Fixes #..., Closes #..., or a direct Issue link.
4. If a PR has no associated Issue and its description is too brief to establish the problem, skip it and resample until 10 qualifying candidates are collected.
5. If fewer than 10 qualifying PRs exist, report the actual number; do not fabricate, duplicate, or infer missing samples.

3. Binary Annotation Criteria. For each *(Issue, PR code changes)* pair, assign a binary score.

Score 1 (valid and transferable): The change (i) contains substantive implementation logic, such as modifying a condition or loop, adjusting API arguments, adding input validation, or introducing exception handling; (ii) demonstrates a reusable repair or implementation pattern; and (iii) has sufficiently clear Issue context and PR code changes to connect the problem with its solution.

Score 0 (invalid or low-value): The change consists primarily of (i) documentation, spelling, comments, linting, or formatting; (ii) version or dependency updates without substantive implementation logic; (iii) lock files, generated files, images, compiled files, or other binary artifacts; or (iv) repository-specific hard-coded identifiers, URLs, dates, or magic numbers without a reusable solution pattern. For mixed PRs, judge whether the main problem-solving logic is understandable and transferable.

4. Verification Requirements. Inspect the Issue description, PR description, and code diff before assigning a label; do not judge from the PR title alone or infer unsupported Issue–PR links. For auditability, retain the repository name, PR number/URL, Issue number/URL, merge date, binary score, and a brief justification for every sample.

5. Required Repository-Level Output. Aggregate the annotations using the following format. Compute *Valid Rate* as $Valid/Sampled \times 100\%$. The Notes column may summarize common invalid samples or dominant repository characteristics.

Rank	Repository	Sampled	Valid	Valid Rate	Notes
1	owner/repository	10	0–10	0–100%	Optional summary

6. Candidate Repository Sets. Set A—Maintenance Intensity: [INSERT REPOSITORY LIST A HERE] **Set B—Second Selection Strategy:** [INSERT REPOSITORY LIST B HERE]. Complete the sampling and annotation independently for every repository in both sets.

Table 5: Prompt used for manually validating the experience-transfer value of Issue–Pull Request pairs.

complete understanding. In django-14007, Card 1 explained converter inheritance while Card 2 showed where converters should be invoked; the agent synthesized both to identify that insert operations bypass the converter pipeline.

E.2 When Experience Distracts (12 browsed → unresolved)

- **Misdirected Exploration** (4 cases): The card points the agent to the wrong code area or opposite approach. In django-13128, the card recommended using ExpressionWrapper—exactly the workaround the issue aimed to eliminate. The agent recognized the conflict but still spent 20 extra steps (57 vs. 37 baseline) recovering.
- **Stale Detail Adoption** (2 cases): The card’s code-level suggestions are outdated. In pytest-7324, the card suggested

ast.NameConstant (deprecated since Python 3.8); the agent over-complicated the fix that the baseline solved with ast.Constant alone.

- **Conflicting Card Confusion** (2 cases): Multiple cards suggest different strategies, and the agent produces an incorrect hybrid. In sympy-13615, one card classified elements as real/non-real/symbolic while another used numeric/symbolic; the agent merged both schemes, producing inconsistent logic (30 edits vs. 23 baseline).
- **Premature Confidence** (4 cases): The card provides a correct high-level principle, but the agent over-trusts a specific implementation detail. In sympy-24066, the card correctly stated “dimensionless args → dimensionless result”; the agent implemented a `dim == 1` check, but the codebase re-

quired the `is_dimensionless()` API since $\text{time}/(\text{capacitance} \times \text{impedance}) \neq 1$ symbolically.

E.3 Which Card Fields Matter Most

The **Fix Strategy** field in the Resolution Layer is associated with both the strongest positive pattern (Guided Localization, 25% of help cases) and the strongest negative pattern (Misdirected Exploration, 33% of distract cases). The **Patch Digest** field carries version-sensitivity risk (Stale Detail and Premature Confidence together account for 50% of distractions). These observations are consistent with decoupling abstract strategy from concrete implementation in experience governance, and motivate future work on version-awareness and confidence calibration.

F Additional Quantitative Analyses

F.1 Same- and Cross-Repository Experience

We parse every Browse call for four backbones and compare each card’s source repository with the repository of the SWE-bench Verified instance. Table 6 shows that 31.1%–46.7% of browsed cards are cross-repository. Same-repository cards always correspond to an issue ID different from the test issue because the latter and its associated PR and patch are excluded during memory construction.

Table 7 compares instance-level outcomes. A *same-repository reader* browses cards only from the test repository, whereas a *cross-repository reader* browses at least one card from another repository. The cross-repository group has a higher observed resolution rate for every backbone. On DeepSeek-V3.1T, comparison with the SWE-Agent outcome on the same instance subsets gives gains of +1.2 percentage points for same-repository readers and +3.6 points for cross-repository readers. These associations demonstrate that cross-repository experience is used at scale and can be useful, but they do not replace a controlled same-only versus cross-only memory intervention.

F.2 Search and Browse Frequency

Table 8 reports tool use over the 500 benchmark instances. Browsing is sparse, ranging from 0.21 to 0.73 calls per instance. For DeepSeek-V3.1T, the distribution of Browse calls per instance is: 276 instances make no Browse call, 144 make one call, 75 make two calls, and 5 make three calls. Together with the per-call measurements

in Section 5.7, these counts show that the wall-clock increase is driven mainly by LLM inference over browsed content rather than by the Search or Browse implementations themselves.

Backbone	Browsed cards	Same repo	Cross repo	Cross repo (%)
DeepSeek-V3.1T	309	213	96	31.1
Kimi-K2	184	98	86	46.7
Qwen3-Coder-30B	103	58	45	43.7
Qwen3-235B	363	197	166	45.7

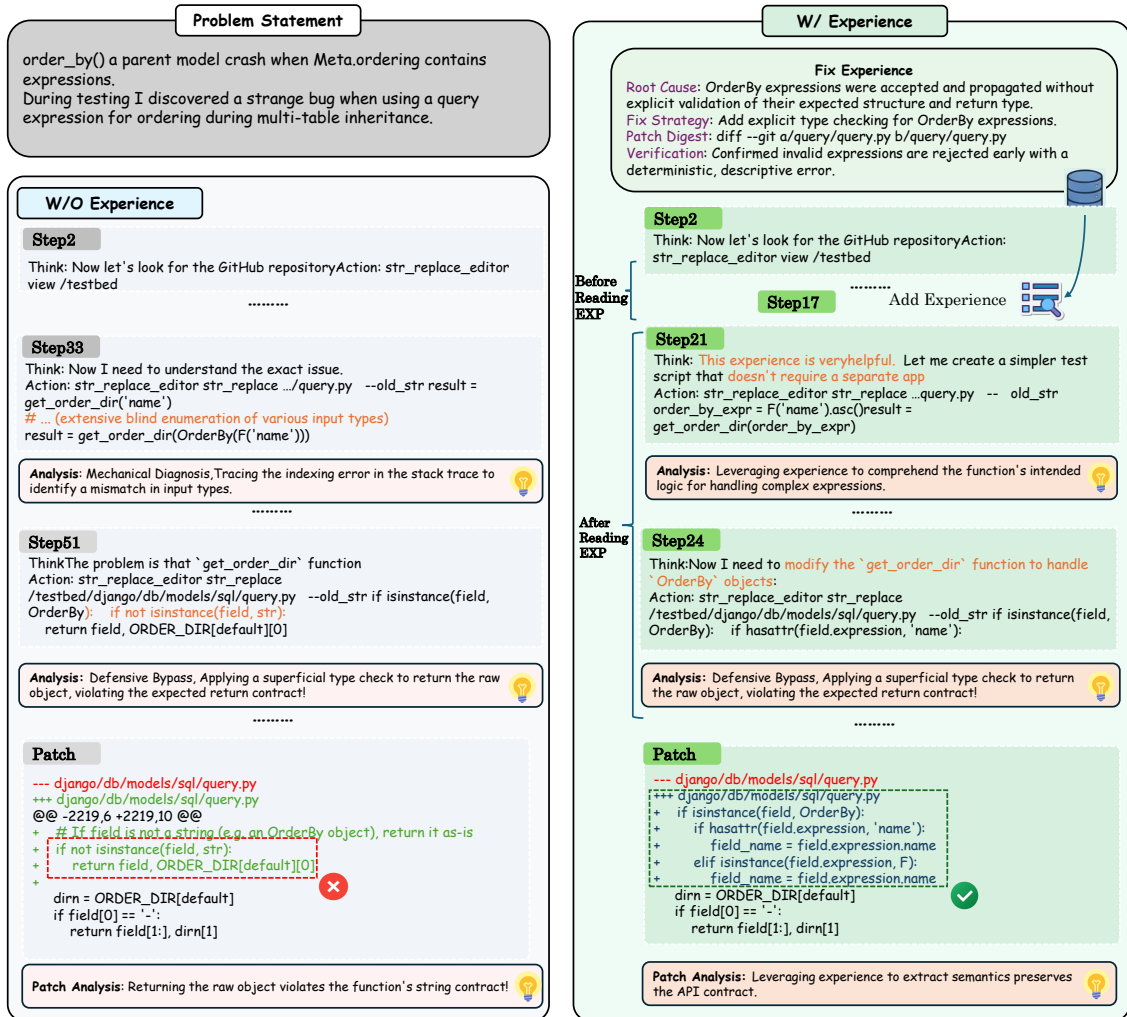
Table 6: Source-repository breakdown of all browsed experience cards.

Backbone	Same-repository readers		Cross-repository readers	
	Resolved (%)	n	Resolved (%)	n
DeepSeek-V3.1T	58.6	169	69.1	55
Qwen3-235B	46.7	107	51.1	47
Qwen3-Coder-30B	41.3	46	46.2	26
Kimi-K2	50.0	84	52.6	38

Table 7: Observed resolution rates for instances that browse only same-repository cards and those that browse at least one cross-repository card.

Backbone	Search calls	Browse calls	Browse/instance
DeepSeek-V3.1T	825	309	0.62
Qwen3-235B	501	363	0.73
Qwen3-Coder-30B	509	103	0.21
Kimi-K2	343	184	0.37

Table 8: Search and Browse tool-call counts across 500 SWE-bench Verified instances.



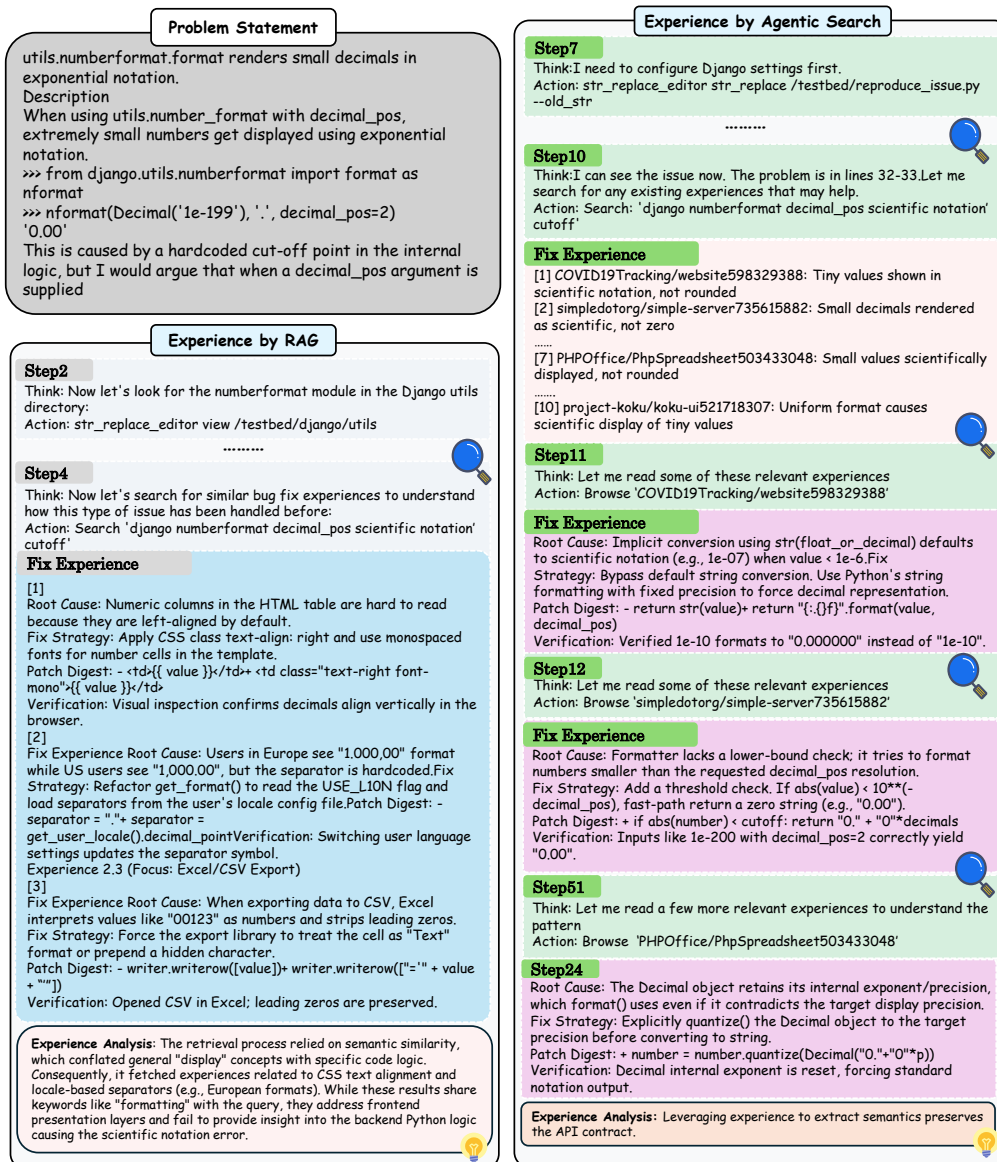


Figure 8: Comparison of Standard RAG (Top) vs. MemGovern's Agentic Search (Bottom). RAG retrieves irrelevant frontend formatting experiences due to surface similarity. MemGovern's agentic workflow allows it to browse and locate the specific backend logic for handling scientific notation thresholds.

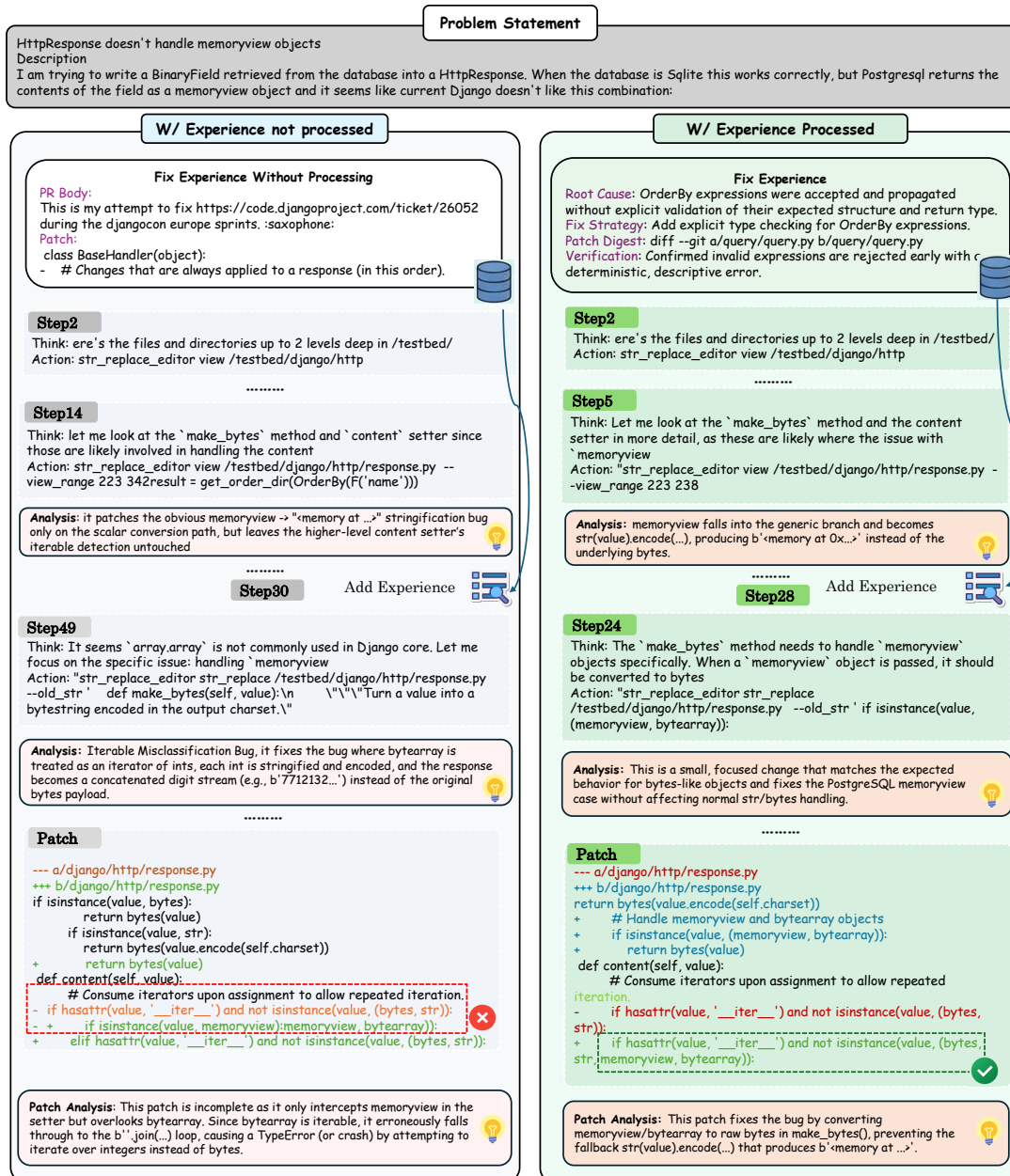


Figure 9: Comparison of Raw Experience (Left) vs. MemGovern's Processed Experience (Right). Raw experience distracts the agent with unrelated code changes, leading to an incomplete patch. Governed experience isolates the core repair logic, enabling the agent to correctly handle both memoryview and bytearray cases.

Fields

- "Problem Summary" (*String*): Concise technical summary of the bug root pattern. Must be generalizable; do NOT include repository names, commit hashes, or overly specific variable names.
- "Signals" (*List[String], 10–18 items*): High-signal matchable keywords/phrases extracted ONLY from the issue content. Each item is a short term (typically 2–4 words) covering mechanism/type, symptom/failure mode, trigger/input/environment, and affected component; include common aliases when helpful (e.g., NPE / null pointer). Avoid vague stop words; no duplicates.
- "Root Cause" (*String*): Evidence-backed causal chain from trigger to failure and the violated assumption. If uncertain, provide 1–2 candidate root causes, each with supporting evidence and a concrete observation/test to disambiguate.
- "Fix Strategy" (*String*): Design-level summary of how the bug was addressed (e.g., validation/guards, ordering/state transitions, parsing rules, synchronization, error handling/reporting). Mention constraints and compatibility notes (behavior/API, backward compatibility, performance impact), plus trade-offs/risks and mitigations. Do NOT restate the patch line-by-line.
- "Patch Digest" (*String*): Structured semantic digest grounded in the diff. Must include *Changed Areas* (generalized components/modules/files) and 3–8 *Key Chunks* describing what changed and why, linked back to the root cause. Do NOT paste raw diffs or large code blocks.
- "Verification" (*String*): Concrete, checkable verification plan and evidence: reproduce steps (before/after), tests added/updated (unit/integration/regression), and key edge cases/boundaries. If the patch has no tests, propose minimal steps with expected outcomes.

Figure 10: MemGovern's Experience Card Schema.