Adaptive Memory System for Coding Agents

CS224V Project Proposal | Jeremy Gu

Motivation

Following our Oct 22 discussion on agent memory systems that (1) learn across sessions, (2) improve from feedback, and (3) decontextualize experiences into transferable patterns. Your key insight: *"abstraction level needs to be adjusted on the fly—nobody is doing that."*

Three Research Questions

Current coding agents (Claude Code, Cursor, GitHub Copilot) lack three critical capabilities:

Q1: How can agents maintain goal alignment during multi-step tasks?

Problem: Agents drift from original objectives—research on SWE-agent shows 75% of failures relate to scope control breakdown (52% incorrect/overly specific implementations, 23.4% cascading errors), SWE-agent（NeurIPS 2024).

Our preliminary analysis of 10 real coding sessions confirms agents frequently modify files outside intended scope or skip verification steps.

Solution: Multi-dimensional alignment checking

- **Scope**: File modifications within intended scope?
- **Plan**: Tool usage matches current phase?
- **Test**: Verification steps executed?
- **Evidence**: Changes have supporting rationale?

Mechanism: Track alignment violations through weighted combination of four guards, WARN at drift ≥0.5 and issue a rollback advisory (block submission) at ≥0.8, following shadow→advisory rollout.

Evaluation on SWE-bench Verified:

- Primary metric: Drift Rate (% actions violating any of four guards)
- Secondary metrics: Scope alignment, False positive rate
- Baseline: Unmonitored agent (Q1 disabled)
- Target: Drift rate <15% (exact target TBD after baseline, Week 1)

Q2: How can agents extract and reuse patterns across sessions?

Problem: Each task solved from scratch—no knowledge accumulation across sessions.

Solution: Pattern extraction and retrieval

Workflow:

1. **Extract**: LLM analyzes successful task → Identifies reusable pattern
2. **Decontextualize**: Transform specific solution into general approach
3. **Store at multiple abstraction levels**:
    - Level 1: High-level approach (e.g., "validate inputs before processing")
    - Level 2: Conceptual explanation with key steps
    - Level 3: Concrete code template
4. **Retrieve**: Semantic search when new task matches problem signature
5. **Apply**: Inject relevant pattern into agent context

Evaluation on SWE-bench Verified:

- **Pattern Reuse Rate**: % test tasks where pattern was retrieved (similarity ≥ 0.7)
- **Success with Pattern**: Resolve rate when pattern applied vs not applied
- **Coverage**: % test tasks with available relevant pattern from training
- Target: Demonstrate positive correlation between pattern availability and success

---

Q3: How should abstraction levels adapt dynamically?

Problem: Static responses don't match varying needs—some situations need detailed guidance, others need concise hints.

Solution: Context-based abstraction selection

Context Signals (all observable from SWE-bench):

- **Task indicators**: Problem statement length, number of files mentioned
- **Pattern factors**: First time seeing this pattern type vs repeated
- **Execution state**: Number of prior attempts, recent action success

Selection Approach:

- Initial strategy: Rule-based heuristics
- If time permits (Week 4): Learn from outcomes (which level worked best in which contexts)

Evaluation on SWE-bench Verified:

- **Primary**: Compare resolve rate of dynamic selection vs fixed-level baselines
- **Secondary**: Analyze which levels work best for which task types
- Target: Dynamic ≥ Fixed (any improvement validates approach)

Novel Contribution: This addresses your insight—"nobody is doing dynamic abstraction." Current systems (AutoCodeRover, SWE-agent) use fixed retrieval depth.

Data & Evaluation

Dataset: SWE-bench Verified (500 high-quality tasks)

What SWE-bench Provides:

- Real-world bug reports from popular Python projects

- Ground truth solutions (for evaluation only)
- Test suites for objective pass/fail measurement
- Human-annotated difficulty labels (e.g., "<15 min fix"), enabling stratified analysis

Note: A chronological variant (e.g., SWE-bench-CL) can be used as an auxiliary dataset to stress-test cross-session learning (Q2), but the primary benchmark remains Verified for comparability.

What SWE-bench Cannot Provide:

- Interactive user feedback (no real users in loop)
- Time measurements (automated execution)

Our Split:

- Memory-building corpus (for Q2): Use the official `princeton-nlp/SWE-bench` train split to extract and store patterns (no overlap with final test set)
- Final evaluation: Use the full SWE-bench Verified (500) as an independent test set for reportable results

For Q2/Q3 auxiliary analysis, we may additionally construct time-ordered subsets (e.g., repository-wise sequences or SWE-bench-CL) to assess temporal generalization, while keeping Verified as the primary report.

Dataset Justification: We report main results on SWE-bench Verified (comparability, reproducibility). We additionally evaluate on time-ordered subsets/CL to stress-test cross-session effects; these are supplementary and do not replace Verified as the primary benchmark.

Metrics

Primary Metric: Resolve Rate

- Definition: % tasks where all required tests pass
- Current landscape (Verified): top closed-source systems report >70% (e.g., Claude Sonnet 4.5 ~77%, OpenAI o3 ~71% at the time of writing); research baselines vary widely by agent scaffold (framework/pipeline)
- Our Target (Verified context): set relative to published research baselines under similar constraints (e.g., ≥30% with our guard/pattern/dynamic-abstraction pipeline), finalized after baseline runs

Q1 Metrics:

- Drift Rate (primary): % actions violating alignment guards (weighted combination of Scope/Plan/Test/Evidence)
- Scope alignment (secondary): report as Precision/Recall —
  - Precision: % of edited files within gold patch scope
  - Recall: % of gold patch files that were edited
- Baseline: Will establish in Week 1 by running unmonitored agent
- Target: Drift rate <15% (improvement over baseline, exact target TBD)

Q2 Metrics:

- Pattern reuse rate: % tasks where pattern retrieved
- Success with pattern: Resolve rate (pattern used) vs (pattern not used)
- Target: Demonstrate statistical significance using proportion/paired tests (Fisher/χ^2 or McNemar) or logistic regression; report 95% CIs

Pattern usage levels (operational definitions):

- retrieved_only: pattern retrieved and shown; no evidence of application.
- used_like: partial application (step/template overlap ≥ τ, or guard improvement Δdrift > 0).
- used_strict: clear adherence to steps/templates and corresponding guard improvements.

Logging chain: log retrieve → present → adopt (level) with failure_reason (e.g., mismatch, conflict, low confidence).

Q3 Metrics:

- Dynamic vs Fixed: Resolve rate comparison
- Level distribution: Which levels used most often
- Target: Dynamic ≥ best fixed level

Statistical Requirements:

- Prefer proportion tests (Fisher/χ^2), paired McNemar for within-task comparisons, or logistic regression with fixed effects; report 95% CIs and effect sizes
- Ablation study: Q1 only, Q2 only, Q1+Q2, Full

Implementation

Tech Stack:

- Agent scaffold (fixed for comparability): tools/control/validation/budget configuration is locked per experiment
- LLM API: GPT-4o for critical reasoning tasks (goal parsing, pattern extraction)
- Local LLM: Qwen-32B for high-volume tasks (code generation)
- Vector DB: ChromaDB for pattern retrieval
- Budget: per-task cap (compute-matched) + total budget note (hybrid strategy)

Agent Scaffold (Fixed, versioned): To ensure fair comparisons and reproducibility, all experiments run under a fixed agent scaffold — a versioned configuration that explicitly locks the agent's tools, control logic, validation rules, and resource budgets. We log a `scaffold_id` (plus key knobs like τ/top-k/cap) per run so any observed gains are attributable to our mechanisms, not incidental parameter changes.

Run Policy & Reproducibility:

- Q1: start in shadow mode (warn-only), enable rollback advisory after threshold calibration
- Q2: retrieval threshold τ tuned on validation; top-k ≤ 3 with dedup; log retrieve → present → adopt(level)
- Repro: single entry command via runner.sh; log seed, scaffold config, per-task cost, tokens/calls/actions

Timeline (6 weeks):

- **Week 1-2**: Q1+Q2 implementation, establish baselines
- **Week 3**: **Checkpoint evaluation on 50 validation tasks**
  - Go/No-Go decision: If pattern reuse ≥20% AND resolve rate improvement observed → proceed with Q3
  - Otherwise: Focus on strengthening Q1+Q2
- **Week 4**: Q3 implementation (conditional on Week 3 results)
- **Week 5**: Main evaluation on SWE-bench Verified (500 tasks)
- **Week 6**: Paper + Demo

Expected Contributions

1. **Mechanism**: Multi-dimensional alignment monitoring for coding agents
2. **Framework**: Pattern decontextualization with multi-level storage
3. **Exploration**: Context-based abstraction selection (addresses "nobody is doing this")
4. **Evaluation**: Rigorous comparison with SOTA on standard benchmark

---

Open Questions / Limitations

Known from start:

- Optimal alignment weights: Will tune during training (Week 1-2)
- Pattern quality: Depends on LLM extraction accuracy (will validate manually)
- Abstraction selection: Starting with rules, learning if time permits
- Time savings: Cannot measure actual time on automated benchmark (will use proxy metrics like action count, token usage)

To be determined Week 3:

- Whether Q3 is feasible given Q2 results
- Exact target numbers after baseline establishment

Future work:

- User study to validate original "user expertise" adaptation hypothesis
- Multi-language support (currently Python-only via SWE-bench)

Comparison with Research Systems

While commercial tools (Claude Code, Cursor, GitHub Copilot) exhibit the problems motivating this work, we evaluate against research systems on the standard SWE-bench benchmark:

System	Drift Monitoring	Cross-Session Memory	Dynamic Abstraction
AutoCodeRover	❌	Static retrieval	❌
SWE-agent	❌	❌	❌

System	Drift Monitoring	Cross-Session Memory	Dynamic Abstraction
Ours (proposed)	✅ Multi-dimensional	✅ Decontextualized patterns	✅ Context-aware

Differentiation: Only research system proposing all three capabilities with dynamic abstraction adjustment.

References (placeholders; add split/time/DOI/links on update)

- AutoCodeRover — ISSTA 2024 (Lite 19–22%; Verified ~46%).
- SWE-agent — NeurIPS 2024 (Full ~12.5%).
- Claude Sonnet 4.5 — Verified ~77% (2025-09).
- OpenAI o3 — Verified ~71% (2024-12, high compute).

Appendix: Key Definitions and Assumptions

Notation

Throughout this proposal:

- **Bold**: Key terms being defined
- *Italics*: Emphasis or quoted concepts
- `Code`: Variable names, file paths, function calls
- Percentages: Always out of 100 (e.g., 20% = 20/100)

Core Concepts

Memory

- **Definition**: A system's ability to store, retrieve, and apply knowledge from previous task executions to improve future performance.
- **In our context**: Cross-session pattern storage where successful solutions are extracted, decontextualized, and made available for future similar tasks.
- **Contrast**: Current agents have no memory—each task starts from zero knowledge.

Goal Alignment

- **Definition**: The degree to which an agent's actions match the intended task scope and plan.
- **Measurement**: Tracked through four weighted dimensions with combined drift score:
  - Scope Guard (weight 0.4): File modifications within intended boundaries?
  - Plan Guard (weight 0.3): Tool usage matches current task phase?
  - Test Guard (weight 0.2): Verification steps executed?
  - Evidence Guard (weight 0.1): Changes have supporting rationale?
- **Drift Score Calculation**: `drift_score = 0.4×scope + 0.3×plan + 0.2×test + 0.1×evidence`
- **Action Decision**:
  - `drift_score < 0.5` → OK (proceed)
  - `0.5 ≤ drift_score < 0.8` → WARN (log but allow)
  - `drift_score ≥ 0.8` → rollback advisory (suggest rollback/block submission)

- **Example of drift**:
  - Task: "Fix login validation to accept emails with '+' character"
  - Expected scope: `login_validator.py` (1 file, ~5 line change)
  - Agent did: Modified 12 files, refactored authentication system, changed database schema, skipped testing
  - Result: Goal drift detected, rollback advisory triggered (submission blocked)

Multi-step Task

- **Definition**: A task requiring multiple sequential actions across different phases (e.g., understand → reproduce → implement → verify → test).
- **In SWE-bench**: Each GitHub issue requires an estimated 5-15 agent actions on average (will validate on training set), spanning file reading, editing, test execution, and debugging.
- **Why it matters**: Single-step tasks don't exhibit goal drift; multi-step tasks are where agents lose track of original objectives.

Scope (Task Scope)

- **Definition**: The set of files, functions, and test cases that are relevant to a specific task.
- **In SWE-bench**: Inferred from problem statement and (for evaluation) validated against ground truth patch.
- **Components**:
  - `allowed_paths`: Files mentioned in problem or likely related
  - `forbidden_paths`: System files, configs, unrelated modules
  - `required_tests`: Test cases that must pass for success
- **Example**:
  - Task: "Fix null pointer in payment validation"
  - Scope: `[payment_processor.py, payment_validator.py, test_payment.py]`
  - Out of scope: `user_auth.py`, `database_config.py`
- **Measurement**: % actions that modify files within scope

Phase (Task Execution Phase)

- **Definition**: The current stage of problem-solving in a multi-step task.
- **Standard phases** (adapted from software debugging methodology):
  1. **Understand**: Read files, search code, analyze problem
     - Allowed tools: `read_file`, `grep`, `search`
     - Forbidden tools: `edit_file`, `submit`
  2. **Reproduce**: Run tests, observe failures
     - Allowed tools: `run_test`, `bash`
     - Forbidden tools: `edit_file` (don't fix before confirming bug)
  3. **Implement**: Modify code, make changes
     - Allowed tools: `edit_file`, `insert`, `replace`
     - Required before: Must have run failing test
  4. **Verify**: Test changes, check for regressions
     - Allowed tools: `run_test`
     - Required before: Must have edited at least one file

- **Transition logic**: Phases can overlap but must follow partial order (can't implement before understanding)
- **In practice**: Inferred from action history, not explicitly declared

  Mapping note (terminology alignment with implementation): Understand/Reproduce/Implement/Verify ↔ reproduce/modify/test/regress

Action

- **Definition**: A single atomic operation that an agent performs during task execution.
- **Types** (from SWE-bench agent toolkit):
  - Read: `read_file(path)`, `search(pattern)`, `grep(query)`
  - Write: `edit_file(path, old, new)`, `insert(path, line, content)`
  - Execute: `run_test(test_name)`, `bash(command)`
  - Control: `submit(solution)`, `rollback()`, `checkpoint()`
- **Metadata** (logged for each action):
  - Timestamp, current phase (inferred), files affected, success/failure outcome
- **Constraints**: Max 100 actions per task (SWE-bench standard), 5 min timeout per action
- **Why it matters**: Unit of analysis for drift detection (Q1) and efficiency measurement (Q2)

Pattern

- **Definition**: A reusable problem-solution pair extracted from a successfully completed task.
- **Components**:
  - Problem signature: Type of bug/issue (e.g., "null pointer exception")
  - Solution approach: General strategy (e.g., "add validation before object access")
  - Code template: Concrete implementation (optional, for Level 3)
- **Example**: "When encountering AttributeError on None, check for None before accessing attributes"

Decontextualize

- **Definition**: The process of transforming a context-specific solution into a general, transferable pattern.
- **Process**: Strip away specific variable names, file paths, and project details → Retain core logic and approach
- **Example transformation**:
  - Context-specific: "In `payment_processor.py:45`, added `if customer.account is None: return` before `customer.account.debit(amount)`"
  - Decontextualized: "Before accessing object attributes, validate that object is not None to prevent AttributeError"

Extract

- **Definition**: The automated process of identifying key patterns from successful task executions using LLM analysis.
- **Input**: Complete task trajectory (problem description, actions taken, final solution, test results)
- **Output**: Structured pattern with problem signature, solution approach, and abstraction levels
- **Trigger**: Only executed when task succeeds (all tests pass)

Abstraction Level

- **Definition**: The degree of detail in presenting a pattern, ranging from high-level hint to concrete implementation.
- **Three levels**:
  - Level 1 (Hint): One-sentence guidance (e.g., "Check for None before accessing attributes")
  - Level 2 (Explanation): Conceptual approach with key steps (e.g., "When object might be None: (1) add if-check, (2) handle None case, (3) proceed with safe access")
  - Level 3 (Code): Full implementation template with example
- **Selection criteria**: Based on task complexity, pattern familiarity, and agent's recent performance (Q3 research question)

Context (for Abstraction Selection)

- **Definition**: Observable signals about the current task and agent state that inform which abstraction level is most appropriate.
- **Three categories**:
  1. **Task indicators**: Problem statement length, # files mentioned, repository size
  2. **Pattern factors**: Times this pattern type has been seen (0 = first, 5+ = familiar)
  3. **Execution state**: # prior attempts on this task, recent action success rate
- **Key distinction**: All signals are observable from SWE-bench execution (no user profile needed)
- **Why it matters**: Enables dynamic selection without interactive user feedback
- **Example**:
  - High context (complex task + unfamiliar pattern + low success rate) → Level 3
  - Low context (simple task + familiar pattern + high success rate) → Level 1

Data Structures

SWE-bench Data Format

- **Definition**: A benchmark dataset of 2,294 real-world GitHub issues from 12 popular Python repositories.
- **Key fields per task**:
  - `instance_id`: Unique identifier (e.g., "django__django-12453")
  - `problem_statement`: GitHub issue description (user-reported bug)
  - `repo`: Repository name and URL
  - `base_commit`: Git commit hash before the fix
  - `patch`: Ground truth solution (used only for evaluation, not visible to agent)
  - `test_patch`: Test file demonstrating the bug
  - `PASS_TO_PASS` / `FAIL_TO_PASS`: Test names that define success criteria
- **What it provides**: Real-world complexity, objective test-based evaluation, ground truth for comparison
- **What it lacks**: No interactive users; no time measurements
  - Note: Difficulty labels are available in the Verified subset and used for stratified analysis
- **Why SWE-bench for this research**:
  1. Objective evaluation: Test pass/fail eliminates subjective judgment
  2. Reproducibility: Dockerized environment ensures consistent results

3. Real-world relevance: Issues from production codebases
4. Community benchmarks: Enables comparison with published systems
5. Scale: 500 tasks (SWE-bench Verified) provide statistical power for final evaluation

Our Data Split (Final)

- **Memory-building corpus (Q2)**: Use the official SWE-bench train split to extract/store patterns; strictly disjoint from final eval
- **Final evaluation (Q1–Q3)**: Full SWE-bench Verified (500) as the independent test set for reportable results
- **Supplementary**: Time-ordered subsets/CL used only to stress-test cross-session effects; main results remain on Verified
- **Why this split**: Ensures comparability/reproducibility (Verified) while enabling temporal analyses as add-ons

Task Execution (in SWE-bench)

- **Definition**: A complete automated attempt to solve one SWE-bench task from start to terminal state.
- **Lifecycle**:
  - Begins with: Task loading and initial problem analysis
  - Contains: All agent actions (reads, edits, tests, searches)
  - Ends with: Test suite execution and pass/fail result
  - Duration: TBD (depends on action limit, typically 50-100 actions)
- **Characteristics**: Deterministic, automated, single-shot execution with no human intervention

Chat Session (in Cursor/AI Assistants)

- **Definition**: An interactive conversation between a human developer and an AI coding assistant for solving a coding problem.
- **Structure** (typical format):
  - User message: Task request or clarification
  - Assistant message: Analysis, code, or questions
  - Tool calls: File reads, edits, bash commands
  - Results: Output from tool executions
- **Lifecycle**:
  - Begins with: User's initial request
  - Contains: All user-AI exchanges and tool operations
  - Ends with: User satisfaction or abandonment
  - Duration: Variable (5 minutes to several hours)
- **Characteristics**: Interactive, non-deterministic, human-guided throughout
- **Key difference from Task Execution**:
  - Task Execution: Single initial prompt → automated agent → test evaluation
  - Chat Session: Back-and-forth dialogue → human guidance → subjective success
- **Our usage**: Supplementary data for validating chat2events extraction (not for quantitative evaluation)

Baseline (Comparison Systems)

- **Definition**: Reference systems used to evaluate relative performance of our approach.
- **External baselines** (three levels):

    1. **Weak baseline**: Vanilla GPT-4
        - Single-shot patch generation from problem statement
        - No tool use, no iteration
    2. **Medium baseline**: Static RAG
        - Retrieve similar past solutions (no decontextualization)
        - Present to agent without abstraction selection
    3. **Strong baseline**: AutoCodeRover (research baseline)
        - Published system; we report under the Verified split with split/time/scaffold noted (compute-matched)
        - Static retrieval depth, no goal tracking
        - Reference: Zhang et al. 2024
- **Internal baselines** (for ablation):
    - **Unmonitored agent**: Our system WITHOUT drift detection (Q1 disabled)
        - Purpose: Measure Q1 effectiveness by comparing drift rate before/after monitoring
        - Expected: Higher drift rate than monitored version
    - Q1 only: Goal tracking without pattern memory
    - Q2 only: Pattern learning without goal tracking
    - Fixed-level: Pattern with Level 1/2/3 always (for Q3 comparison)
- **Measurement**: All baselines run on same test set for fair comparison

Note on recent SOTA systems: Recent proprietary systems (Claude 3.7: 70.3%, o3: 71.7%) achieve higher scores through massive test-time compute scaling ($1,600+ per task for o3). We compare against research systems with comparable compute budgets and documented methods (AutoCodeRover, SWE-agent) to evaluate our novel contributions—drift monitoring, pattern learning, and dynamic abstraction—rather than competing solely on compute resources. Our target (Verified, compute-matched) is ≥30%, finalized after baseline runs.

Assumptions

1. **Pattern transferability**: Patterns extracted from training tasks will be relevant to test tasks (validated through semantic similarity threshold ≥ 0.7 in retrieval).

2. **LLM extraction quality**: GPT-4o can reliably identify reusable patterns from successful solutions (will validate manually on sample of 20 patterns in Week 2).

3. **Guard weight tunability**: Optimal weights for Four-Guard system (currently 0.4/0.3/0.2/0.1) can be learned from training data (Week 1-2 tuning phase).

4. **Context signals sufficiency**: Task complexity can be estimated from observable features (problem statement length, number of files, repository size). For SWE-bench Verified subset, difficulty labels ("<15min", "15min-1h", "1-4h", ">4h") are available and can be used directly. For other datasets, we fall back to heuristic estimation.

5. **Action limit**: Agent can complete most tasks within 50-100 actions (standard SWE-bench practice).

6. **Test suite reliability**: SWE-bench test suites accurately reflect task success (validated by benchmark creators).

7. **Proxy metrics validity**: Action count and token usage serve as reasonable proxies for time savings when direct time measurement isn't available.

8. **Pattern diversity**: A subset of the official train split will yield diverse pattern types covering common bug categories (null checks, type mismatches, boundary conditions, concurrency). We will validate by inspecting problem statements in the memory-building corpus.

9. **Retrieval threshold**: Cosine similarity ≥ 0.7 is appropriate for semantic matching (too low = irrelevant patterns, too high = miss useful patterns). Will validate on validation set and adjust if needed.

10. **Single pattern per task**: For tasks with multiple matching patterns, agent receives top-1 pattern only (to avoid overwhelming context). Future work could explore multi-pattern presentation.

11. **Phase inference accuracy**: Current phase can be reliably inferred from action history using heuristics (e.g., no edits yet → understand phase). Will validate by manual inspection of 20 task executions in Week 1.

12. **File mention extraction**: Number of files mentioned in problem statement can be reliably extracted using NLP parsing (regex + heuristics). Will validate on sample of 20 problem statements in Week 1.

Scaffold Config Template (minimal)

Example minimal YAML for the primary setup:

```yaml
scaffold_id: v1_verified_guard
tools:
  plan_then_edit: true
  allow_shell: test_only
  edit_granularity: hunk_or_file
q1:
  shadow_mode: true
  weights: {scope: 0.4, plan: 0.3, test: 0.2, evidence: 0.1}
  thresholds: {warn: 0.5, rollback: 0.8}
q2:
  tau: 0.70
  top_k: 3
  dedup: true
  compress: true
budget:
  per_task_cap_usd: 2.0
  max_steps: 100
  max_runtime_min: 30
logging:
  log_chain: true   # retrieve → present → adopt(level)
```