RefactorGuard: Predictive Failure Analysis for Agentic Refactoring

CSCI 7000-005: Agentic AI in SWE - Spring 2026

Dima Golubenko; February 2, 2026

Problem Statement

- **Unpredictable failure modes**: Although the current state-of-the-art (SOTA) agents like MANTRA achieve impressive results (82.8% success rate for MANTRA specifically), they still fail in high-complexity scenarios, creating a reliability gap that prevents fully autonomous utilization of such systems
- **Resource Waste**: When agents fail, they waste time, cost of model inference, and developer trust. Additionally, if they fail a task, they will continue retrying, which might waste even more resources
- **Lack of Self-Correction**: the current agents attempt any task even if they are known to handle it poorly; they lack the checks to stop execution for a scenario where it will most likely fail

Motivation & Problem Significance

- **Cost efficiency**: Identifying a failing task pre-emptively could reduce the cost overhead for language models inference. When AI agents fail, they often enter expensive retrial loops consuming a lot more tokens than a successful scenario
- **Human-in-the-loop**: If failing scenarios are identified early, such tasks can be directed to human reviewers without the need to waste time and inference costs
- **Barriers to enterprise adoption**: the current success rate of around 80% for MANTRA is insufficient for enterprise production environments. Enterprise teams require high reliability and deterministic outcomes to integrate a tool with their environments

What is the current SOTA solution?

- **Reactive vs early detection**: Current tools rely on reactive mechanisms (self-correction loops, RAG) to fix errors after they happen. This “make a mistake and then fix it” approach is computationally expensive and slow.
- **Post-Hoc Analysis**: existing benchmarks like RefactorBench provide failure analysis, but only when the execution is finished. There is currently no tool that integrates these results to predict error before they happen
- **No pre-execution validation**: agents lack a task rejection mechanism. They ignore code complexity signals and attempt tasks that are statistically impossible given their context limitations

Proposed Solution: RefactorGuard

- **Framework**: Introduce an additional verification layer that analyzes code metrics and determines if an agent will succeed in the refactoring task or not
    - Current flow: User -> Agent -> Success/Failure
    - Proposed flow: User -> RefactorGuard -> [Safe?] -> Agent -> Success
- **Data approach**:
    - Will **train** on **MANTRA's (703 instances)** and **CoRenameAgent's (1,573 instances)** datasets to learn failure patterns
    - Will **validate** on **MM-Assist's (210 instances)** dataset
- **Predictive models**: Will train 2 classification models on code features (e. g. LOC, number of files, context utilization)
    - Viability prediction: Should the agent start a given task?
    - Potential issue prediction: If it fails, what will be the specific issue?

Why is the solution successful?

- **Tangible economic results**: if it works as expected, it would help reduce the number of unsuccessful agentic AI refactoring tools runs and hence reduce wasted time and inference cost
- **Improved system reliability**: refactoring agents would no longer be “hit-or-miss” and achieve real reliability
- **High predictive accuracy**: aiming to achieve F1 score of > 0.8 on the validation set

Questions?

Project Proposal – Agentic Tool for System Integration Testing

What is the problem you are trying to solve?

System integration testing is one level above unit testing and functional testing, where you see how different parts of a system interact together, or a new part of the system interacts with the existing part. The bugs found in this type of testing are hard to categorize and fix because of the different components involved.

Why is the problem important to solve?

System integration testing is hard to automate and perform at scale. Human testing (and human automation) can only cover specific interactions and not all of them. Explaining how a failure occurred is difficult without thorough technical knowledge. Finding issues on large scale customer setups (which is done in system integration testing) is critical before pushing updates – for customer retention and business trust.

What is the current state-of-the-practice or state-of-the-art in solving this problem, and what are its limitations?

Manually written tests on customer setups, automation frameworks which are limited by design, test suite pipelines, telemetry logging. AI assisted test generation but this mostly focuses on unit tests.

What is your proposed solution?

To build an agentic system integration testing tool to approach this problem with reasoning rather than as a static test suite. Basic flow could be - understand the system and find potential interactions, have human in the loop to review test plan, generate and execute the tests, monitor the results (maybe have human in the loop for this too), reason about the failures and modify the tests accordingly. System integration tests are very unique to each solution and are hard to generalize. Having an agentic system could save hundreds of engineer hours per feature.

How would you convince the professor, classmates, and the research community that your solution is successful?

Compare metrics like code coverage, bugs found, running time (how long the system is occupied running the test) to traditional system integration tests or suites.

Compare the cost of training the models and building this system to the conventional test suites. See how feasible it is to implement and how quickly.

Agentic Network Log Analysis

- Networking, compute, and memory are the core components for building critical infrastructures across industries

- Identifying subtle packet, handshake anomalies, or physical layer issues among millions of packets requires deep protocol expertise and hours of manual filtering.

- Current practices include hard coded logic and some statistical analysis, but rely on industry expertise. Some demonstrations from Wireshark using Gemini 3 are available, but widespread use is limited due to context windowing issues.

- The proposal is an investigation is enabling LLMs to look into 20-40Gb client/server logs
    - Retrieve logs through indexing
    - Statistical analysis
    - Layered approaches
    - Multi agent solution integrating networking standards

- A successful project will be able to identify a variety of network errors, or point to application errors while keeping a limited context window

Coordination Under Constraint: Global Workspace–Style Architectures for Agentic Refactoring

Designing and evaluating a Global Workspace–style agent architecture to study coordination, attention, and error propagation in multi-module refactoring.

Telephone +1 – 720 691 3747

Email Manodnya.kadambiniharsha@colorado.edu

Website https://www.linkedin.com/in/manodyna-kh/

Problem

- Agentic systems are increasingly used for software refactoring tasks.
- Multi-module refactoring requires coordination across interacting components.
- Current agentic refactoring approaches make decisions largely locally.
- Local decisions often conflict, causing error propagation and regressions.
- There is no principled coordination mechanism for global refactoring decisions.

Why Now?

- Refactoring errors scale with system size and coupling.
- Small coordination failures can introduce subtle, high-impact bugs.
- Human developers mitigate this via global reasoning and attention.
- Agentic systems lack comparable coordination safeguards.
- Improving coordination is critical for reliable agentic software tools.

The Current Landscape

- Single-agent LLM-based refactoring tools.
- Multi-agent pipelines with role specialization.
- Tool-augmented agents with planning and reflection loops.
- Coordination is implicit or heuristic, not explicit.
- No shared workspace for global arbitration of refactoring decisions.

What Next?

- Design a Global Workspace–style coordination architecture for agentic refactoring.
- Specialized agents propose refactoring actions independently.
- A shared workspace aggregates and compares competing proposals.
- A coordination mechanism selects actions based on global coherence.
- Architecture is inspired by functional coordination theories, not cognition claims.

Eval?

- Compare baseline agentic refactoring with workspace-coordinated refactoring.
- Evaluate refactoring correctness and regression introduction.
- Measure error propagation across modules.
- Assess consistency and stability across repeated runs.
- Demonstrate improved coordination under global constraints.

Coordination Under Constraint: Global Workspace–Style Architectures for Agentic Refactoring

Designing and evaluating a Global Workspace–style agent architecture to study coordination, attention, and error propagation in multi-module refactoring.

Telephone +1 – 720 691 3747

Email Manodnya.kadambiniharsha@colorado.edu

Website https://www.linkedin.com/in/manodyna-kh/

Agentic Cross-Repository PR Reviewer

CSCI 7000

RAYAAN LODHI

Problem

PR Reviews Lack Cross-Repository Context

- ❏ Modern systems = many repos / microservices
- ❏ PR reviewers see local diffs only
- ❏ Domain knowledge lives *across* repos

Example:

- ❏ A frontend changes an API call shape, but the backend repo isn't referenced in the PR
  - ❏ Reviewers can't catch any possible issues without strong domain knowledge

Importance

Uncaught errors in PRs often times cause more delay (induced failure in integration testing or QA). Agentic tools *can* reduce it.

Especially with the onset of agentic tools, shipping with quality and speed is paramount.

Current/Related Solutions

- ❏ CodeantAi - Offers AI based code reviews, lacks cross-repository solutions
- ❏ Coderabbit - Automated, agentic PR reviewer, lacks cross-repository solution

More Generally…

- ❏ Static PR bots (linting, style coercion, tests)
- ❏ LLM PR reviewers (confined to single-repo context)

Proposed Solution

- ❏ Agentic AI PR reviewer
- ❏ Cross-repository retrieval
- ❏ Contract-aware feedback

Success Criteria

Unfortunately, there are no benchmarks for a task like this right now.

Measures of success can come from purposefully inducing cross-repo integration issues in experiments of ranging size and complexity, and detecting whether the product is able to catch and address the issue.

Questions.

From CoRename to CoMove

CSCI 7000 Project Proposal | Sudhanva Manjunath

Move Method has four phases:

1. Identify a misplaced method in its host class.
2. Find a suitable target class.
3. Check refactoring pre- and post-conditions.
4. Execute the transformation.

CoRenameAgent shows that developer intent plus an agentic, human-in-the-loop workflow can safely coordinate refactorings across a codebase.

MM-ASSIST addresses many of these steps for Move Method using LLMs and static analysis, but it does not iteratively learn from developer feedback.

Toward an Agentic Move Assistant

Instead of one-shot recommendations like MM-ASSIST, I propose an intent-driven, learning workflow that begins from a developer’s seed move in the IDE.

The system should support three roles:

- **Intent interpretation:** infer why the move happened.
- **Safe proposal:** suggest additional moves and ask for approval.
- **Project-wide propagation:** search for similar cases elsewhere.

Key addition: Episodic memory stores approvals and rejections so future suggestions improve over time.

Questions the project will Answer

- Can we infer a reliable “move scope” from one seed move?
- Does episodic memory reduce false positives compared to MM-ASSIST?
- What new or modified agents are actually needed for Move Method?

Robust & Explainable LLM-Based Refactoring Agents via Heterogeneous Models and Human-in-the-Loop Feedback for MANTRA

Problem I am attempting to solve :

- Shared hallucinations: MANTRA’s developer and reviewer agent uses different instances of same LLM for generation and review, causing the same reasoning errors to go undetected.
- Fragile architecture choices: Performance is highly sensitive to embedding models, RAG chunking, and prompts, yet these are rarely evaluated systematically.
- Limited human collaboration: The systems offer one-shot recommendations with little explainability, reducing trust and adaptability in real-world codebases.
- Resolving Clean code dependency: Try to make strong robust Agentic AI workflow that cna work with bad quality/obfuscated code. For example lot of “temp” and “t1”,”t2” in variable names.

Why Is This Problem Important?

Most current refactoring agents assume & use :

- clean code
- strong naming conventions
- homogeneous model reasoning

This project challenges those assumptions and aims to make refactoring agents:

- more reliable
- more transparent
- more aligned with how developers actually work

Current State-of-the-Art (MANTRA) & Its Limitations

I propose we extend the MANTRA architecture by systematically studying how model diversity, retrieval strategies variance, and human feedback will affect the reliability and usability of the automated refactoring agents.

Rather than introducing a new refactoring type, the project will focus on making the refactoring agent more robust, explainable, and generalizable, especially under realistic and adversarial conditions.

Proposed Solution

1. Heterogeneous Agent Design

Instead of using two instances of the same LLM for the developer and reviewer agents, we will:

- use different LLMs for generation and review
- compare various combinations of *strong generator* + *conservative reviewer*
- study whether diversity reduces shared hallucinations and logical blind spots

2. RAG Sensitivity Analysis

We will systematically evaluate:

- Effect of different embedding models
- Various dynamic chunking strategies (method-level, class-level, hybrid)
- The tradeoff between retrieval cost and refactoring accuracy/embedding quality

This directly addresses an underexplored assumption in MANTRA: that RAG configuration choices are largely benign.

3. Human-in-the-Loop Episodic Refinement

We introduce a lightweight human feedback loop, where:

- The agent will proposes a refactoring
- The human will approve the intent but can adjust implementation details, if needed
- The feedback is stored in episodic memory and reused in future decisions

This will move MANTRA refactoring from a one-shot recommendation to a collaborative process.

4. Robustness Under Poor Code Quality

Finally, we test the agent on intentionally degraded code, such as:

- generic method and variable names (temp, data, t1,t2)
- weak documentation
- low semantic signal

This will help us evaluate whether LLM-based refactoring truly understands structure, or relies on naming heuristics.

Evaluation Plan

- Benchmark: Test against RefactorBench and MANTRA’s original setup to get a clear and fair comparison.
- Controlled Variations: Systematically vary embedding models, RAG chunking strategies, and developer–reviewer LLM pairs (same vs different models) to study their effects.
- Metrics: Use CodeBLEU and AST Precision/Recall, but also track compile success, false positives, latency, and token cost so quality isn’t measured in isolation but cost and success rate is measured as well.
- Stress Testing: Evaluate on messy and obfuscated code (generic names, poor structure) and larger multi-file, compound refactorings.
- Human-in-the-Loop Study: Measure acceptance rate, trust, explainability, and time saved when developers can guide and correct the agent.
- Rigor: Report variance across runs and show statistical significance instead of cherry-picked wins.

How Will I Convince People It Works?

- Quantitative improvements over MANTRA baseline
- Comparing hallucination and false-positive rates
- Studying performance under degraded code quality
- Developer feedback in human-in-the-loop evaluation
- Clear cost-to-performance comparison

Thank You

Project Proposal: Agentic Architectural Inference from Code and Deployment

- What is the problem you are trying to solve?

  Legacy software products often lack accurate and up-to-date architectural documentation. While the source code and deployments evolve continuously, high -level system views or low-level design views are typically created manually and quickly become outdated and inconsistent with existing systems. Moreover, organizations follow their own internal conventions, terminology, and diagram styles difficult to adopt. With this architectural understanding remains fragmented and heavily dependent on tribal knowledge.

- Why is the problem important to solve?

  Reliable and consistent architecture diagrams are essential for onboarding, compliance, long-term system maintenance especially in domains such as banking. This is one of the problems that I actually initially faced after I joined my organization three years ago. A system that derives architecture from existing code and deployment configurations will improve trust, usability and make system knowledge accessible to beyond a small group of experts.

- What is the current state-of-the-practice or state-of-the-art in solving this problem, and what are its limitations?

  Current approaches rely on manual documentation, static analysis tools, and LLM-based developer assistants. LLM-based tools can summarize or answer questions about codebases, but they typically operate in a single pass, do not maintain a persistent architectural model, and may hallucinate structure. Most existing tools also ignore deployment and infrastructure information or treat it separately from code, limiting their usefulness for real architectural reasoning

- What is your proposed solution?

  I propose to build an agentic system that infers from both high-level and low-level architectural diagrams by analysing both application code as well as deployment artifacts such as infrastructure-as-code. The system uses multiple agents to explore code structure, component boundaries and align them with deployment resources (eg: services, containers, VMs). While keeping in mind, the generated diagram follows organization- specific documentation conventions.The system iteratively refines its output and explicitly goes through uncertainty where architectural diagrams are ambiguous.

- How would you convince the professor, classmates, and the research community that your solution is successful?

  Results can be compared with diagrams drawn by principal architects and existing LLM tools. We can also survey developers productivity and level of understanding after referring to these diagrams when created. Qualitative feedback and case studies can show more trustworthy architectural insight than current state-of-art methods.

Spec-Guided Code Skeleton Generation

Ishika Patel - ishika.patel@colorado.edu

Opportunity

Tooling to help realize the full agentic software engineering vision, starting with having better specification signals.

Current Solutions & State of Practice

- Spec implementation is typically an ad-hoc process
- Many tools treat spec interpretation as a single transformation because they optimize for the end-task. By stopping at skeleton, we may have the opportunity to bring more visibility to agentic reasoning.

Proposed Solution

- Build an agentic system that generates code skeleton from short specifications
  - Given: Spec/Idea in natural language
  - Agent will identify outcomes, fill gaps in spec and propose required methods
  - Eval Output: Code skeleton with Doc-style specs

Evaluation Plan

- Evaluate correctness using ClassEval or RepoClassBench
  - Class Eval gives class_name, class_description, skeleton, method_info
  - RepoClassBench gives the class_name, a detailed spec, a vague spec

CSCI-7000: SWE AGENTIC COMP | PROJECT PITCH

ViSQL Agent

The Challenge

Text2SQL chatbots and agents fail when the database schema grows too large, and when overall semantic understanding is lacking. Improvements are still possible.

Long-term Memory

Similar to previous work like *AgentSM*, provide a a long-term RAG memory for the agent to store common table relationships it encounters. Also store feedback from GenUI feedback loop.

Gen UI Feedback Loop

ViSQL named after "vis"ualization. Main contribution that is not well researched or explored. Generate previews as graphs, plots, or tables, let users correct query mistakes with UI elements.

PRIMARY HYPOTHESIS

"Closed-loop parameter feedback loops between humans and agents significantly reduce the semantic error rate compared to other zero-shot or pure memory based architectures."

$$\text{Error}_{\text{HITL}} \ll \text{Error}_{\text{Base}}$$

Framework: LangGraph / A2UI; Bench Dataset: Spider 2.0

Execution-Guided Autonomous Program Repair Using Agentic AI

By - Soham Sant

MOTIVATION

- Developers spend **significant** time debugging and fixing bugs.
- Software failures and bugs are **detected** automatically(test,CI), but **not fixed** automatically.
- **Manual debugging doesn’t scale** with growing codebase.
- No end-to-end reliable autonomous system that:
    - Identifies bugs
    - Fixes them
    - Validates code
    - Integrates safely

Limitations of Existing Tools

Current Tools : eg. GitHub Copilot , IDE Assistants

They do suggest fixes and explain errors, but :

- No Test Validations
- No safe rollbacks
- Significantly depends on Human in the loop.

State-of-the-Art Agents:

Eg. SWE-agent, AutoCodeRover, SWE-bench based tools.

They can navigate the codebase, edit files, run tests, but:

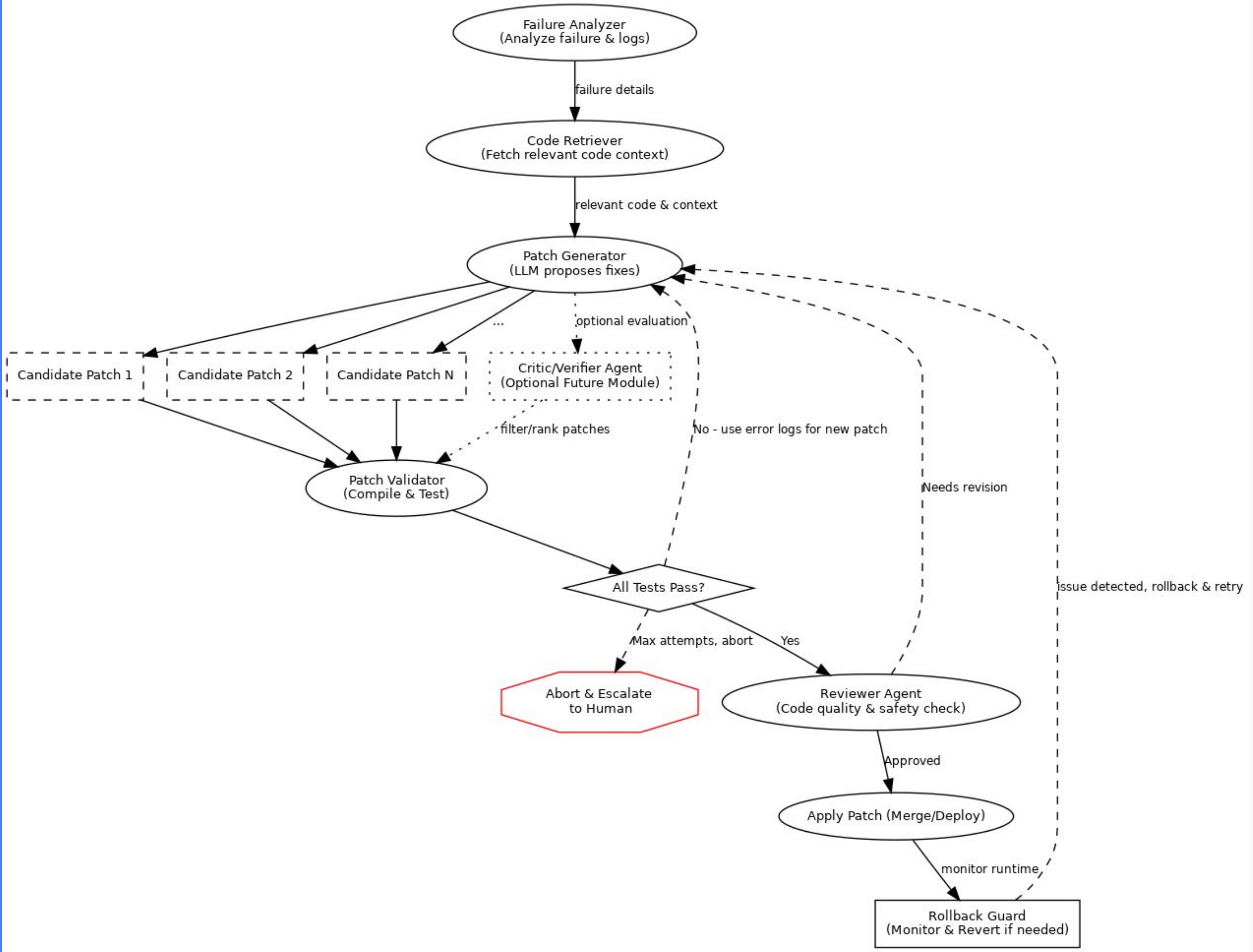
- Poor fault localization
- Overfit to pass tests
- Lack of explainability
- No safe rollbacks

Proposed Solution

An autonomous system of multi-agents that detects software failures/bugs, proposes patches, testing them rigorously and safely integrate or rollback in case of failure.

- Execution guided
- Multi-Agent
- Feedback-driven
- Safe Roll-back

Proposed Workflow



Core Components & How They Work

Component	How It Works
Failure Analyzer	Stack trace parser, log-to-NL summary
Code Retriever	Static analysis, test-to-code mapping,RAG
Patch Generator	LLM with few-shot bug-fix prompting
Validator	Unit test runner + compiler + linter
Reviewer Agent	Critic LLM + fix scoring + hallucination filters
Rollback Guard	Threshold-based abort + human-in-loop

Evaluation Plan

Quantitative Metrics:

- Fix Success Rate
- Rollback rate
- Attempts per bug
- Time to green CI

Ablation Study:

- Without critic agent
- Without memory
- Singlet-shot vs multi-attmept

User Study:

Feedback on usability and patch quality from developers.

Minimum Viable Product(MVP)

- Detects failing test cases in a Python/Java project (pytest/JUnit)
- Parses error logs
- Generates 3-5 fix candidates using a LLM
- Run tests in a separate environment (Docker)
- Pick the best fix (via simple thresholds or Reviewer Agent)
  - All Test Pas
  - Minimal Code change
  - No compiler issue
- Submit a pull request with explanation
- Rollbacks if Fails

Questions? & Feedbacks!

ReactRefactor Agent

What problem am I trying to solve?

- React apps accumulate React-specific code smells over time
- Examples:
    - Large Components doing state, effects, and rendering
    - Prop Drilling across many layers
    - Props in initial state causing stale UI
- Developers refactor manually, but it’s slow, risky, and inconsistent
- There is no end-to-end system that detects these smells and safely refactors React code with human approval

Why does this problem matter?

- React refactors are not just structural, they affect state, hooks, and UI behavior
- Small mistakes can break rendering or cause subtle bugs
- Because of this, developers:
  - delay refactoring
  - or reject automated suggestions
- The result is growing technical debt and harder-to-maintain components

Current state of practice & limitations

- Research has cataloged React-specific code smells and the refactorings developers use to address them.
- Smell detection is mostly AST-based, while refactoring generation is often LLM-based.
- There is no full pipeline that connects detection, planning, execution, and verification.
- Most tools lack strong human-in-the-loop support for reviewing and approving refactorings.
- LLM-only tools risk hallucinating unsafe changes, while AST-only tools miss intent and design context.

My proposed solution

- A MANTRA-style agentic refactoring system for React
- Flow:
  - Scout agent detects React smells (AST-based, explainable)
  - The Planner agent maps smells to refactorings and generates a step-by-step plan, then runs a quick compile/typecheck to catch obvious breakage.
  - Human reviews and approves the plan
  - Builder Agent applies the approved refactoring.
  - Reviewer agent validates the changes by running Typescript Checks, Eslint and tests.
- The system iterates until it’s correct or stops

How would I prove it works?

- We measure before-and-after smell detection to verify that refactorings actually remove the targeted smells.
- We enforce hard correctness gates, including TypeScript compilation, ESLint checks, and Jest tests.
- We conduct a human study to assess whether developers would accept the refactored code as a real pull request.
- Developers also evaluate readability and confidence in the generated refactorings.
- We compare our approach against AST-only refactoring tools and LLM-only refactoring approaches.

Project Idea Pitch: MCP Tool Selection

A Refactoring-Oriented Benchmark + Agent Study

Mohammed Raihan Ullah

What Is The Problem?

- Modern code agents can call tools, but real workflows involve many tools (IDE refactorings, search, tests, git, linters).
- In MCP-style ecosystems, tools are large/heterogeneous/changing → agents waste steps by choosing wrong tools or looping.
- Existing evaluations are generic tool-use or issue-fixing focused; they don’t isolate tool selection for refactoring.
- Refactoring is tool-heavy: navigate → choose primitive (rename/extract/move) → apply → validate → commit.
- Benchmarks measure end-to-end success but don’t directly measure:
  - tool discovery/retrieval from many MCP tools,
  - tool choice quality (best refactoring primitive vs brittle text edits),
  - effect of tool availability (IDE vs CLI-only) on success/cost.

Why Is It Important To Solve?

- More reliable refactoring agents (safer transformations, fewer regressions).
- Actionable evaluation: separate “reasoning about refactoring” vs “choosing/using tools”.
- Evidence for which MCP tools matter most for refactoring workflows.
- Tool-selection traces can become supervision data for better agents.

State-Of-The-Art Practices, and Its Limitations

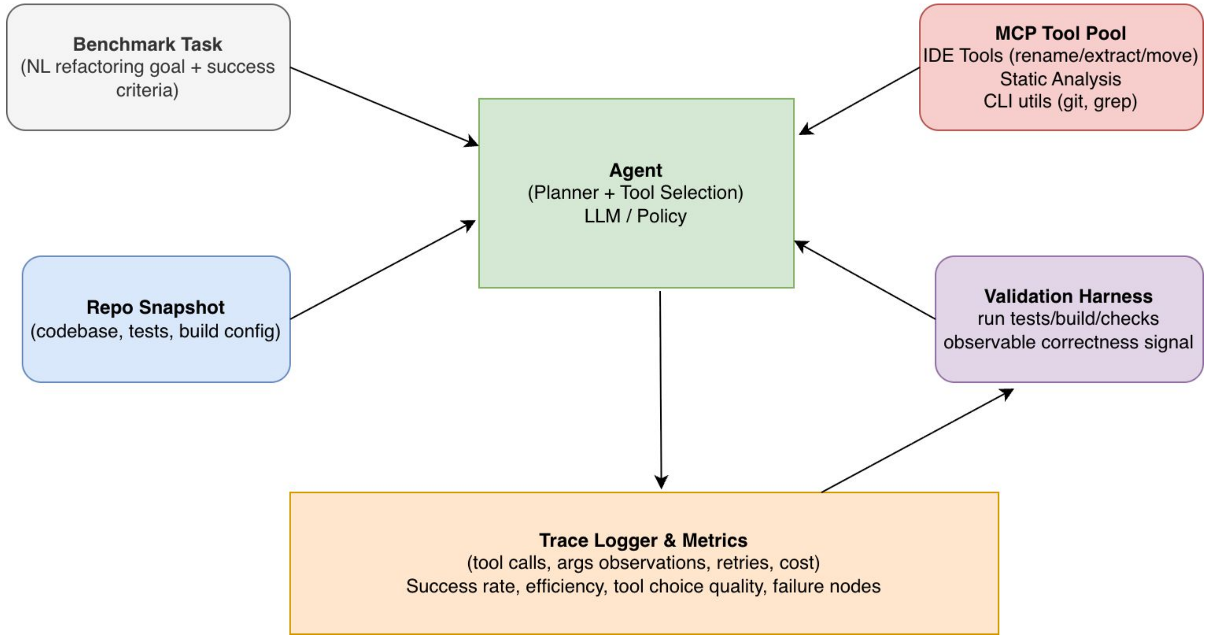
- Tool-use learning & API calling: ReAct, Toolformer, ToolLLM/ToolBench, Gorilla — mostly generic APIs, not IDE refactorings.
- Agent benchmarks: API-Bank, AgentBench — interactive tool use, but not refactoring-centric.
- SE agents: SWE-bench + SWE-agent — repo-level issues with execution feedback; refactoring tool-selection not isolated.
- Refactoring benchmarks: RefactorBench — multi-file refactors, but not explicitly MCP tool-pool/tool-retrieval focused.
- MCP-scale benchmarks: LiveMCPBench, MCP-Bench — tool-rich MCP evaluation, but broad domains; refactoring/IDE workflows underrepresented.

Key gap: no benchmark that stresses tool selection for refactoring under a realistic MCP tool pool.

Proposed Solution

- **Build a refactoring-oriented MCP benchmark** where the agent must choose among many MCP tools (IDE + CLI).
- **Measure tool selection**: retrieval, tool choice quality, argument correctness, recovery from failures.
- **Main question**: Which tools actually help refactoring agents, and when?

Proposed Solution: Architectural Overview


Benchmark Task
(NL refactoring goal + success criteria)
MCP Tool Pool
IDE Tools (rename/extract/move)
Static Analysis
CLI utils (git, grep)
Agent
(Planner + Tool Selection)
LLM / Policy
Repo Snapshot
(codebase, tests, build config)
Validation Harness
run tests/build/checks
observable correctness signal
Trace Logger & Metrics
(tool calls, args observations, retries, cost)
Success rate, efficiency, tool choice quality, failure nodes

How Would You Convince Others That Your Solution is Successful?

- Start ~30–50 tasks (rename/extract/move etc.), then scale.
- **Metrics**: success rate, tool-selection accuracy, efficiency (tool calls/time/cost), safety (structured refactoring vs text edits).
- **Baselines**: SWE-agent-style CLI, IDE-enabled agent, retrieval-only recommender, multi-agent vs single-agent.
- **Ablations**: remove IDE tools / remove tests / enlarge tool pool / add distractor tools.

Open Questions & Research Directions

- Tool discovery at scale (ranking/reranking/schema grounding with hundreds of tools).
- When to refactor vs edit (learn preference for safe primitives).
- Generalization across repos/languages/IDEs.
- Training from traces without overfitting.
- Human-in-the-loop: minimal hints/approval that most improves selection.