CMSC848N - Midterm Report
A.C.E - Agentic Coding Engine

Hemanth Nandakumar
hemanthn@umd.edu

Saketh Akella
sakella@umd.edu

Quan Nguyen
nguyqu03@umd.edu

1 Introduction

1.1 Motivation

The automation of the software development lifecycle represents a key frontier in artificial intelligence research. Large Language Models (LLMs) have shown strong capabilities in code generation, debugging, and synthesis, translating natural language specifications into executable programs and correcting them through reasoning. Proprietary systems such as OpenAI's Codex and Anthropic's Claude Code exemplify this potential, yet their closed-source nature and high computational requirements limit accessibility and reproducibility in academic and lightweight research contexts. This growing dependence on opaque, large-scale models has created a gap between industrial performance and academic innovation.

1.2 Background and Context

In parallel, the field of agentic AI, where models act as autonomous decision-making entities capable of planning, reasoning, and interacting with tools, has gained momentum. Agentic coding frameworks such as AutoCodeRover [15], AgentCoder [6], and CodeCoR [13] demonstrate how LLMs can orchestrate cycles of code generation, testing, debugging, and verification, evolving toward self-improving software engineering pipelines. Despite their promise, these systems often rely on large-scale models (30B+ parameters) and extensive computational resources, making them impractical for constrained research environments or smaller deployments.

1.3 Problem Statement

Traditional code generation systems typically operate in a single-shot mode, producing code once without iterative refinement. In contrast, agentic frameworks introduce reasoning-action cycles, an approach inspired by the ReAct paradigm [6], where the model alternates between reasoning steps and tool invocations to analyze results and self-correct. While this iterative approach enables more consistent logical reasoning and better alignment with human-like debugging behavior, its efficiency and practicality under limited computational resources remain open questions.

The central problem addressed in this project is the design and evaluation of a resource-efficient agentic coding framework capable of autonomously solving programming tasks with transparent reasoning and minimal computational overhead. While large LLM-based agents achieve strong results in code generation and debugging, their closed-source nature and resource demands hinder scientific progress.

1.4 Research Objectives

To bridge this gap, our project proposes A.C.E. (Agentic Coding Engine), a lightweight, reproducible, and open agentic coding framework that operationalizes the capabilities of mid-sized LLMs for automated programming tasks. A.C.E. is designed to investigate three tightly coupled research questions:

1. **Model Efficiency and Viability:** Can a quantized, small-to-mid–sized model (3–4B parameters) such as StarCoder-3B, when deployed with 4-bit quantization, perform robustly as the central reasoning engine of a multi-tool coding agent?

2. **Contribution of Agentic Components:** To what extent do agentic design elements, including iterative self-debugging loops, dynamic test generation, and retrieval-augmented generation (RAG), contribute to measurable performance gains on standard

code generation benchmarks such as HumanEval and SWE-bench-lite?

3. **Adaptability via Fine-Tuning:** How effectively can parameter-efficient fine-tuning (QLoRA) specialize a general-purpose code model toward specific software engineering domains without requiring full model retraining?

By systematically addressing these challenges, A.C.E. seeks to advance the open research agenda in agentic programming, providing an accessible, fine-grained framework for understanding how compact language models can autonomously reason, code, and self-improve, paving the way for reproducible, low-cost, and explainable coding assistants.

2 Literature Review

We reviewed the dominant methodologies in recent work on agentic code generation, model efficiency, and workflow automation, highlighting empirical trends, evaluation practices, and existing gaps that motivate A.C.E.

2.1 Overview of Agentic Code Generation

Recent work on automated code generation has moved beyond single-shot completion to engineer agentic pipelines that integrate planning, tool use, execution, verification, and iterative repair. Two parallel trends dominate the literature: (1) designing agent orchestration and workflow automation that structure multi-step code tasks, and (2) improving robustness via iterative test-and-repair loops and context augmentation (RAG). A third, enabling trend is the development of compute-efficient methods (quantization and PEFT) that make these pipelines feasible in constrained environments.

Across multiple works, agentic systems are realized as composable modules—planner, generator, executor, verifier, and repairer—that are connected either by fixed templates or by meta-composition routines. The recent works show three common methodological families.

2.1.1 Template-Based Pipelines

Most works structure agentic behavior as a chain of modular components. At the simplest level, template pipelines implement a fixed sequence — plan–generate–execute–verify–repair — encoded either as prompt templates or as an executor graph. Li et al. in AgentCoder [6] and Yang et al. in CodeCoR [13] operationalize this by defining explicit role boundaries (programmer, tester, repairer, critic) and strict message schemas (JSON-like exchange formats) so that the output of one module becomes the structured input for the next. These systems emphasize reproducibility and traceability: every generation, test, and repair is logged as an atomic event.

2.1.2 Automated Workflow Composition

A second methodological family aims to automate pipeline construction. Chen et al. in AFlow [3] formalizes a mapping from high-level goals to executable workflow graphs: it decomposes intent into subgoals via an LLM planner, semantically matches subgoals to tool metadata, and assembles a validated graph (nodes = tools/agents, edges = dataflow). Li et al. in SEW [8] extends this by making workflow structure adaptive: it records per-subroutine ROI (success rate, token cost) and uses mutation/pruning policies to evolve the orchestration over time. Practically, AFlow and SEW require a curated tool library and the capacity to evaluate many candidate workflows; they thus trade reproducibility for flexibility and potential performance gains.

2.1.3 Self-Organizing Multi-Agent Frameworks

Self-organizing multi-agent frameworks, such as Zhang et al. [14], further decentralize orchestration: agents spawn subagents, delegate subtasks, and recombine partial solutions. This methodological choice—task splitting and recombination—scales well for very large problems but presumes parallel compute and complex conflict resolution policies.

The literature shows a continuum: fixed templates (high reproducibility), auto-composition (higher autonomy), and self-organization (highest flexibility and cost). For a low-resource project, A.C.E. will adopt a principled hybrid: a reproducible template (for controlled experiments) augmented with light, deterministic composition heuristics (from Chen et al. in AFlow [3]) and conservative pruning rules inspired by Li et al. in SEW [8] (drop subroutines with persistently low ROI). This preserves experimental control while allowing modest adaptation.

2.2 Iterative Repair Methodologies

Iterative repair is the most consistently validated methodology for improving correctness. Across Li et al. in AgentCoder [6], Zhang et al. in AutoCodeRover [15], Yang et al. in CodeCoR [13], and related works, the repair pipeline follows three detailed steps: (1) generate targeted unit tests (often pytest style) via prompt templates or lightweight fuzzers; (2) execute generated code in sandboxed REPLs and parse structured failure traces (error type, line numbers, failing input, stack values); (3) craft repair prompts that include the failing trace and contextual code, then generate and rank patched candidates.

Li et al. in AgentCoder [6] emphasizes role separation: a test designer agent creates discriminative tests while a repair agent focuses on targeted edits—allowing parallel candidate evaluation and majority-voting selection. Zhang et al. in AutoCodeRover [15] extends test coverage by applying controlled fuzzing and input space expansion to expose hidden semantic bugs; its localization heuristics (stack frame mapping, variable snapshotting) enable concise repair prompts that focus the LLM's attention. Yang et al. in CodeCoR [13] introduces a critic stage: beyond pass/fail, a reflection step evaluates candidate robustness (edge cases, API misuse) and prunes brittle fixes.

Common practical decisions across these methodologies are important: tests must be compact yet discriminative; failure traces must be compressed (JSON schemas or structured bullet points) to avoid prompt bloat; and repair budgets must be capped because marginal gains decay. Empirically, iterative loops produce large improvements early (first 1–3 repairs) but diminishing returns thereafter. The literature often underreports operational cost (token counts, wall-clock time), which complicates cost–benefit assessment of prolonged repair.

A.C.E. adopts the proven three-stage repair methodology but with explicit operational controls: (a) generate a constrained, high-value pytest suite per problem (no unbounded fuzzing unless budgeted), (b) encode failures in compact structured form for prompt injection, and (c) cap repair iterations and implement lightweight candidate reflection (single-prompt critic) to improve selection without excessive inference.

2.3 Retrieval-Augmented Generation (RAG)

Several works integrate retrieval to provide grounded exemplars and API documentation, using either static pre-indexing or dynamic per-iteration retrieval. Chen et al. in AFlow [3] and Li et al. in SEW [8] use retrieval to select candidate subroutines or examples during workflow synthesis; Li et al. in AgentCoder [6] and Yang et al. in CodeCoR [13] use per-iteration retrieval so that the repair prompt can include closely related code snippets or documented idioms. Methodologically, retrieval pipelines involve: (1) corpus curation (solutions, docs, API snippets), (2) embedding model selection and indexing (FAISS/Chroma), (3) similarity search and top-k selection, and (4) context injection with truncation heuristics to fit prompt limits.

RAG improves success especially when API semantics or coding idioms are central; however, it increases token and compute overhead. The literature suggests diagnostic logging—precision@k of retrieval, qualitative trace checks linking retrieved snippets to final patches—to attribute gains to RAG rather than to random generation. A.C.E. will measure retrieval precision@k and correlate retrieval hits with successful repairs to determine RAG's net efficiency under quantized model constraints.

2.4 Efficient Model Adaptation and Evaluation Methodology

Enabling agentic pipelines on constrained hardware is achieved by combining 4-bit quantized inference with parameter-efficient adapters. Dettmers et al. in QLoRA [4] prescribes inserting LoRA adapters and training them while loading base weights in NF4/bfloat16 formats—this enables adapter training and subsequent low-bit inference within small-GPU environments. The literature prescribes matching numeric regimes between training and inference to avoid deployment mismatches. We will follow QLoRA's recipe to produce specialist adapters for StarCoder-3B and ensure inference runs in the same quantized regime. This methodological consistency lets us evaluate agentic behavior under realistic, reproducible resource constraints.

The canonical evaluation method is HumanEval with pass@k computation and sandboxed execution [12]. Agentic works augment this with repair iterations, token counts, and sometimes human quality judgments. The literature increasingly calls for standardized operational reporting: total tokens consumed, inference calls, wall-clock time, and VRAM usage [11, 14]. Reproducibility further requires publishing prompt templates, quantization recipes, and adapter weights. Our experimental design will follow the literature's best practices: (1) compute pass@1 and pass@3 using standardized sampling; (2) log operational metrics (tokens, time, VRAM); (3) include ablations isolating RAG and repair components.

2.5 Summary and Positioning of our A.C.E.

Collectively, the reviewed literature prescribes a modular, test-driven agentic methodology with optional enhancements from retrieval and workflow automation. A.C.E.'s methodological contribution is to transplant these practices into a tightly controlled, low-resource experimental regime and to provide rigorous, cost-aware measurements. Practically, A.C.E. implements a reproducible ReAct pipeline (from [6, 13, 12]) with constrained fuzzing and repair heuristics (from [15]), dynamic retrieval (from [8, 3, 13]), and QLoRA adapters (from [4]), all instrumented with detailed operational logging as advocated across the surveys [11, 12]. This positions A.C.E. to answer open questions about which agentic components yield net gains under strict token and hardware budgets, thereby filling a methodological gap in the current literature.

3 Our Methodology

This section presents the proposed technical approach and methodology for the A.C.E. framework. It is organized into three subsections describing the (i) model and deployment setup, (ii) agent architecture and supporting tool suite, and (iii) fine-tuning and evaluation methodology.

3.1 Model and Deployment Setup

We will use StarCoder-3B (Hugging Face: bigcode/starcoderbase-3b) as the core reasoning engine for A.C.E. due to its strong capabilities in code generation and reasoning. To enable deployment on constrained hardware, we will apply 4-bit quantization (NF4 + bfloat16) using bitsandbytes, reducing the model's memory usage to under 5GB. This will allow inference within a Google Colab environment (T4 GPU, 16GB RAM). The quantized model will be integrated into LangGraph, which will orchestrate multi-step reasoning loops and manage interactions with external tools in the agentic workflow.

3.2 Agent Architecture and Tool Suite

We will construct the agent following the ReAct paradigm, alternating between reasoning steps ("Thought") and action steps ("Action"). LangGraph will manage this orchestration, enabling the agent to plan, generate, execute, and iteratively debug code. We will design the agent to call external tools, retrieve relevant context, and update its reasoning based on execution feedback. This setup will ensure the agent operates in a modular, transparent, and reproducible manner, while allowing it to handle multi-step code tasks.

Tool Suite. We will provide the agent with a suite of tools to support autonomous code execution and iterative improvement. A sandboxed Python REPL will allow safe execution of code along with basic file I/O. Dynamic test generation and verification will be implemented using pytest, enabling the agent to detect errors and assess correctness systematically. The agent will perform iterative self-debugging by capturing errors during execution and feeding them back into the reasoning loop to refine code. Additionally, we will implement a retrieval-augmented generation (RAG) module using FAISS or ChromaDB to index relevant codebases. This will allow the agent to retrieve and incorporate contextually relevant code examples when addressing tasks such as those in SWE-bench-lite.

Figure 1 illustrates the overall architecture of the A.C.E. framework. The system integrates a quantized reasoning core (StarCoder-3B) with an agentic orchestration layer managed by LangGraph and a supporting tool suite for execution, retrieval, and evaluation. Together, these components enable iterative reasoning, self-debugging, and context-aware code generation under resource-constrained settings.

3.3 Fine-Tuning and Evaluation Methodology

We will apply QLoRA-based parameter-efficient fine-tuning to adapt StarCoder-3B to domain-specific programming tasks. Adapter layers will be trained on curated programming datasets such as DeepMind Code Contests while keeping the base model weights frozen. This approach will allow the model to specialize for coding tasks with-

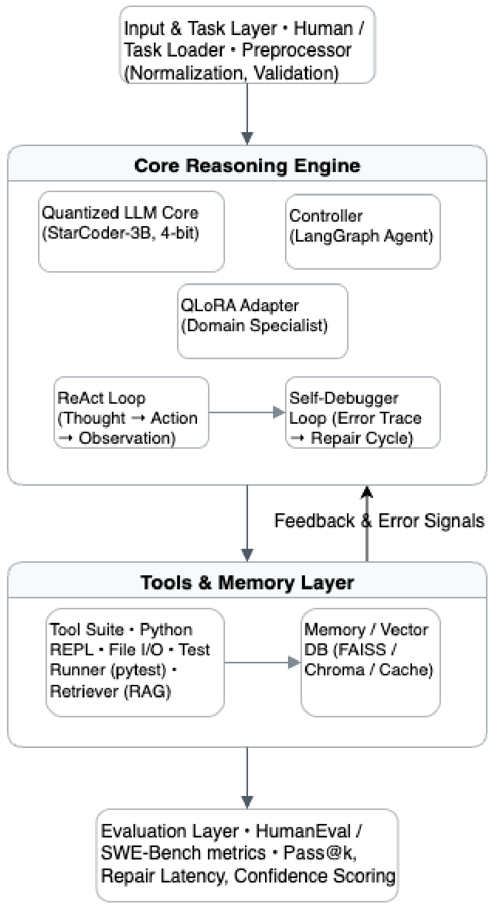
Input & Task Layer • Human / Task Loader • Preprocessor (Normalization, Validation)
Core Reasoning Engine
Quantized LLM Core (StarCoder-3B, 4-bit)
Controller (LangGraph Agent)
QLoRA Adapter (Domain Specialist)
ReAct Loop (Thought → Action → Observation)
Self-Debugger Loop (Error Trace → Repair Cycle)
Feedback & Error Signals
Tools & Memory Layer
Tool Suite • Python REPL • File I/O • Test Runner (pytest) • Retriever (RAG)
Memory / Vector DB (FAISS / Chroma / Cache)
Evaluation Layer • HumanEval / SWE-Bench metrics • Pass@k, Repair Latency, Confidence Scoring

Figure 1: High-level architecture of the proposed A.C.E. framework

out the need for full model retraining. The trained adapters will be published on Hugging Face to support reproducibility and accessibility.

We will evaluate A.C.E. on HumanEval and SWE-bench-lite benchmarks, which provide standardized measures of code correctness and problem-solving ability. We will compare multiple agent variants: the full A.C.E. agent (all modules enabled), versions without RAG, versions without the iterative debug loop, a generalist pretrained LLM, and a specialist QLoRA fine-tuned LLM. Performance will be assessed using Pass@k (k = 1, 3), repair latency measured by the number of agent steps, and token usage to evaluate efficiency. These metrics will allow us to perform a detailed ablation study and understand the contribution of each agentic component to overall performance and resource usage.

4 Progress

This section outlines the progress made so far in developing the A.C.E. framework, focusing on model deployment, agentic loop implementation, and preliminary functional outcomes.

4.1 Model Deployment and Integration

We have successfully deployed the quantized StarCoder-3B model using 4-bit NF4 quantization via bitsandbytes within a Google Colab T4 environment. This configuration enables inference with sub-8GB VRAM usage, allowing reproducible and low-cost experimentation. The model has been integrated into a LangGraph-based agentic framework, establishing the foundational reasoning and execution pipeline. The prototype system is capable of executing Python code in a sandboxed REPL and interacting with file I/O tools for controlled program evaluation.

4.2 Agentic Loop Implementation

The core agent loop has been implemented following the ReAct paradigm, alternating between “Thought” and “Action” steps, coordinated through LangGraph. We have also developed key supporting modules to enable iterative reasoning and self-correction:

- **Dynamic Test Generation and Verifier:** Automatically generating and executing pytest-style unit tests to validate agent-produced code.
- **Iterative Self-Debugging Loop:** Capturing error traces during execution and feeding them back into the LLM to autonomously correct code.
- **Structured Logging:** Recording agent reasoning steps, repair iterations, latency, and model outputs for analysis and reproducibility.

4.3 Preliminary Results

With these modules, the baseline A.C.E. agent is capable of autonomously solving introductory HumanEval tasks, achieving consistent functional correctness on simpler coding problems. These early results confirm the viability of lightweight, quantized LLMs for controlled agentic code generation under limited computational resources.

5 Further Plan

Our next development milestones focus on enhancing the agent’s context-awareness, adaptability, and specialization. This section outlines three key areas of upcoming work: integrating retrieval-augmented generation (RAG), enabling domain specialization through QLoRA fine-tuning, and conducting a comprehensive evaluation and analysis of system performance.

5.1 RAG Integration

We are currently developing a retrieval-augmented generation (RAG) module to provide the agent with external code and documentation context. Planned steps include:

- Building a FAISS or ChromaDB-based vector index of relevant code snippets and documentation.
- Integrating a LangGraph retriever node for dynamic context injection during code generation.
- Evaluating the impact of RAG on code repair accuracy and overall task success, particularly on SWE-bench-lite problems.

5.2 Domain Specialization via QLoRA

Once the RAG pipeline is stable, we plan to perform parameter-efficient fine-tuning (QLoRA) on StarCoder-3B using curated programming datasets such as DeepMind Code Contests. The resulting adapter weights will be published on Hugging Face to support reproducibility. This step will produce a "specialist" variant of A.C.E. tailored for autonomous coding tasks.

5.3 Evaluation and Analysis

In the final phase, we will conduct a full experimental study comparing multiple agent variants:

- Full A.C.E. versus ablated versions (without RAG or without the debugging loop).
- Generalist pre-trained LLM versus QLoRA fine-tuned specialist LLM.

We will evaluate using metrics including Pass@k (k = 1, 3), Repair Latency (number of agent steps per correction), and Token Cost (model usage). The final deliverables will include a reproducible GitHub repository, a detailed research report, and structured evaluation logs documenting agent performance across all experimental conditions.

References

[1] Ashok K. Chandra, Dexter C. Kozen, and Larry J. Stockmeyer. Alternation. *Journal of the Association for Computing Machinery*, 28(1):114–133, 1981.

[2] Ming Chen and Hao Zhou. Retrieval-augmented generation for large language models: A survey. *arXiv preprint arXiv:2402.19473*, 2024.

[3] Rui Chen, Lingfeng Zhao, and Ming Gao. Aflow: Automating agentic workflow generation. *arXiv preprint arXiv:2410.10762*, 2024.

[4] Tim Dettmers, Mike Lewis, Sam Shleifer, and Luke Zettlemoyer. Qlora: Efficient finetuning of quantized llms. *arXiv preprint arXiv:2305.14314*, 2023.

[5] Edward J Hu, Yelong Shen, et al. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*, 2021.

[6] Dong Li, Rui Peng, et al. Agentcoder: Multi-agent-based code generation with iterative testing and optimisation. *arXiv preprint arXiv:2312.13010*, 2023.

[7] Raymond Li, Yacine Belkada, et al. Starcoder: May the source be with you! *arXiv preprint arXiv:2305.06161*, 2023.

[8] Tianyu Li, Han Xu, et al. Sew: Self-evolving agentic workflows for automated code generation. *arXiv preprint arXiv:2505.18646*, 2025.

[9] Hugo Touvron, Thibaut Lavril, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

[10] Jiawei Wang, Rong Chen, et al. Self-organizing multi-agent systems for code generation and task decomposition. *arXiv preprint arXiv:2409.08765*, 2024.

[11] Yuxin Wang, Zhihan Liu, Wenxuan Huang, et al. Ai agentic programming: A survey of techniques, challenges, and opportunities. *arXiv preprint arXiv:2508.11126*, 2025.

[12] Kun Xu, Junjie He, and Rui Wang. A survey on large language models for code generation. *arXiv preprint arXiv:2406.00515*, 2024.

[13] Ming Yang, Yue Zhao, et al. Codecor: An llm-based self-reflective multi-agent framework for code generation. *arXiv preprint arXiv:2501.07811*, 2025.

[14] Hao Zhang, Yue Lin, et al. A survey on code generation with llm-based agents. *arXiv preprint arXiv:2508.00083*, 2025.

[15] Wei Zhang, Chen Luo, et al. Autocoderover: Autonomous program improvement. *arXiv preprint arXiv:2404.05427*, 2024.