ChatGPT

A Prompt-Based Framework for Automated Software Development Using Large Language Models: Design, Implementation, and Evaluation

Introduction

AI-assisted software engineering has rapidly evolved with the advent of large language models (LLMs) capable of generating code, documentation, tests, and more from natural language descriptions. Tools like OpenAI Codex (the model behind GitHub Copilot) demonstrated that LLMs can translate prompts into functional code, solving about 28% of programming problems on first attempt (and up to 70% with multiple tries) [1]. Early applications of LLMs in software development have focused on specific tasks – for example, code completion, bug detection, code translation, and documentation generation [2] [3]. These tools have shown promise in assisting developers. *However, they typically operate as isolated components rather than an integrated workflow across the software lifecycle, leading to fragmentation that limits their overall effectiveness* [2]. In practice, systems like GitHub Copilot act as "AI pair programmers" that suggest code, but developers still must integrate and verify these suggestions manually [4]. This indicates the need for more cohesive frameworks that can leverage AI *beyond* just single-step code snippets.

Prompt engineering has emerged as a key technique to harness LLMs for such broader tasks. Instead of retraining models, engineers craft input prompts to elicit desired behaviors from pre-trained LLMs [5]. By providing structured instructions and context, prompt engineering allows us to **extend LLM capabilities without modifying their internal parameters** [5]. This approach has enabled LLMs to perform complex tasks across many domains, relying solely on well-designed prompts to guide the model. The effectiveness of prompt engineering underscores that *how* we ask the model is crucial to what results it produces – a principle that will be central in our framework. Using off-the-shelf models (like GPT-4) with carefully orchestrated prompts can thus integrate AI into software development workflows *without* any new model training.

Concurrently, there is a **rise of multi-agent coding models** in which multiple LLMs (or multiple prompt instances of an LLM) collaborate, each with specialized roles. Recent research has shown that a collection of specialized agents can work together to achieve better results than a single monolithic model [6]. For instance, one agent can analyze requirements and plan a solution, which then guides another agent focused on coding, followed by an agent that tests and debugs – mirroring a human software team [7] [8]. Such multi-agent LLM systems have demonstrated remarkable improvements in complex tasks. Google's Chain-of-Agents framework, for example, enabled multiple LLM-based agents to handle different parts of a long task and yielded significant performance gains (over 10% improvement on long-context benchmarks) compared to single-agent approaches [9] [10]. Similarly in coding, Islam et al. (2024) showed that a four-agent collaboration (for retrieval, planning, coding, and debugging) achieved state-of-the-art code generation accuracy on difficult programming challenges [11]. These developments highlight a paradigm shift: *instead of one AI trying to do everything, we can have a team of AIs – each prompt-engineered for a specific role – working in concert*. In summary, the convergence of prompt engineering techniques with multi-agent

LLM architectures offers a new path to automate software development end-to-end. This thesis proposes to design and evaluate such a prompt-based, multi-agent framework that can autonomously progress from initial requirements to deployable code.

Problem Statement

Despite advances in AI coding assistants, **manual software development remains slow and labor-intensive**. Traditional development workflows require human effort at every stage – from gathering requirements and designing architecture to writing code, testing, and debugging [12]. These handoffs between phases introduce delays and opportunities for error. The idea of accelerating this workflow with AI is compelling, but current tools fall short of full automation. Modern LLM-based aides like Copilot and ChatGPT typically address only *parts* of the process (most often coding), and even then under constant human guidance. In fact, many AI coding applications focus narrowly on the implementation phase [13], which studies suggest accounts for only 15–35% of the total effort in the software lifecycle [14]. *This leaves the majority of development tasks – from initial requirements engineering to testing and maintenance – largely untouched by automation.* The result is that developers still spend considerable time interpreting vague requirements, managing project structure, writing boilerplate, and fixing errors that AI code generators might introduce.

Moreover, current AI-assisted tools have **significant limitations that necessitate heavy human intervention**. AI-generated code can be syntactically correct yet semantically wrong or insecure – for example, it may produce plausible-looking functions that contain subtle bugs or vulnerabilities. Researchers have observed that LLMs often "hallucinate" incorrect outputs that *appear* valid [15], meaning a developer must meticulously review and debug AI-written code. A recent empirical study of GitHub Copilot usage found that while developers appreciate the useful code suggestions, a major challenge is integrating those suggestions into a working project and ensuring they are correct [4]. In practice, using Copilot is a "double-edged sword" that still requires developers to validate and adjust its outputs carefully [16]. This indicates *current AI coding assistants cannot handle an entire development workflow on their own* – they lack built-in mechanisms for verification, context management across stages, and alignment with the project's evolving requirements.

Crucially, there is **no structured, prompt-based framework that orchestrates multiple LLM agents through the full software lifecycle**. Prior works have provided either isolated capabilities or uncoordinated collections of tools. For example, one system might generate code from a given spec, and another might generate tests, but they are not integrated in a cohesive loop. Surveys of LLM-based software engineering note many point solutions and research prototypes, yet they "fall short of offering integrated solutions" that cover all phases of development [17]. To date, no unified approach exists that can take an initial requirement, *use AI to plan the software, generate the code, test it, and iteratively refine it with minimal human input*. The absence of such end-to-end automation means developers still must manually glue together the outputs of various AI tools (if they use them at all) and handle the transitions between stages. This gap is what our research targets. We posit that a well-defined prompt-driven multi-agent framework can fill this void by having specialized AI agents collaboratively handle each part of the software development lifecycle. By doing so, we aim to overcome the limitations of current tools (fragmentation, lack of verification, need for constant oversight) and address the core problem: **the lack of a fully automated, AI-driven software development workflow that can reliably produce working software from a high-level prompt**.

Objectives

The primary goal of this thesis is to develop and evaluate a **multi-agent, prompt-based framework for automated software development using LLMs**. The research objectives are as follows:

- **Design a Prompt-Orchestrated Multi-Agent Framework:** Define an architecture where multiple specialized LLM-based agents (e.g. Requirements Analyst, Architect, Developer, Tester) collaborate to carry out the software development process from start to finish. Each agent will be assigned clear responsibilities and will communicate via natural-language prompts and responses. The framework should outline how tasks are broken down and passed between agents, and how the overall workflow is controlled.

- **Implement a Prototype System (Targeting Node.js, React.js, and Flutter):** Build a working prototype of the proposed framework to demonstrate its feasibility. The implementation will focus on developing full-stack applications – for instance, a Node.js backend with a React web frontend or a Flutter mobile app – entirely through the AI agents' interactions. Off-the-shelf LLMs (such as GPT-4 or similar) will be used as the intelligence behind each agent. This prototype will serve as a proof-of-concept, with example projects (e.g. a simple e-commerce app or a task management tool) generated by the AI from scratch.

- **Evaluate Performance Against Traditional Development Approaches:** Rigorously assess the framework's outcomes in comparison to (a) manual development and (b) standard AI-assisted coding without a multi-agent structure. Key evaluation criteria will include **development efficiency** (time and effort to produce a working application), **code quality and correctness** (bugs, test pass rates, adherence to requirements), and **usability** (how easily a user – even a non-programmer – can utilize the framework to obtain software). Quantitative metrics (like number of prompt iterations, time to completion, automated test success rates) and qualitative feedback (developer experience, maintainability of generated code) will be collected. The objective is to verify that the proposed approach yields improvements in development speed and output quality while remaining usable to its intended users.

By achieving these objectives, the thesis will demonstrate a novel approach to automate software engineering tasks using large language models, and provide insights into its benefits and remaining challenges. In essence, we seek to show that an orchestrated prompt-based multi-agent system can produce functional software with minimal human input, and to measure how well it performs this task.

Scope and Limitations

This research will focus on **general-purpose application development** using widely-used technologies, within the context of what current LLMs can realistically handle. The scope is defined as follows:

- **Application Domain:** We target typical backend and frontend development scenarios, specifically web and mobile applications. The prototype will concentrate on Node.js for backend logic (e.g., creating REST APIs or server-side functions), React.js for web frontend interfaces, and Flutter for cross-platform mobile app UIs. These choices cover a broad range of modern development stacks, demonstrating generality – the framework is not tied to a single domain or niche problem. However, extremely domain-specific software (such as real-time embedded systems or highly specialized scientific computing) is outside our scope.

- **Automation Focus:** The framework aims to automate the *software development lifecycle stages* of requirements interpretation, design/architecture, coding, and testing within the above domain. DevOps tasks like deployment, performance tuning, or long-term maintenance are considered out of scope for this prototype (though in future the approach could extend to them). We assume a relatively **self-contained project** where one can define requirements and proceed to a deliverable application without needing extensive external systems integration beyond standard libraries and APIs.

- **Use of Pre-Trained LLMs (No Model Training):** A key limitation is that we will **not train or fine-tune any new language models** as part of this work. The framework will rely on existing state-of-the-art LLMs (such as GPT-4 or similar) accessed through their APIs or platforms. All improvements in capability will come from better prompt engineering and workflow design, not from modifying the underlying models. This decision is partly pragmatic – training custom LLMs is resource-intensive – and partly philosophical, to show what is possible with *only* clever orchestration of current models. It also reflects real-world usage where practitioners often leverage off-the-shelf models. The implication is that our system's performance is bounded by the strengths and weaknesses of the base LLMs. For example, if the LLM has knowledge cut off in 2021, it might not know the latest library versions, and the framework must work within such constraints.

- **Known Limitations and Assumptions:** We acknowledge that **LLMs can produce errors or require careful prompting**, and our framework will not magically eliminate all failure modes. There may be cases where the AI agents get stuck or produce suboptimal designs, requiring a human to intervene (e.g., rephrasing a requirement or providing a hint). Complex projects that are very large (beyond the context window of the models even with summarization) or ill-specified might be beyond the framework's current capability. We also assume the availability of an automated testing environment to execute and validate generated code – the scope includes writing tests but not manual UI testing or extensive security auditing. These are constraints we accept in this exploratory project.

In summary, the scope is deliberately chosen to demonstrate the framework in a **feasible yet meaningful setting** – full-stack apps that a small team might build – while avoiding areas that would require capabilities beyond today's LLMs. By using prompt engineering to integrate pre-trained models into a development workflow, we align with the strength of those models (flexible natural language understanding and generation) and mitigate the need for any training data or model fine-tuning [5]. The limitations outlined will be discussed in our evaluation, and we will clarify which challenges were addressed and which remain for future work.

Proposed Framework

Our proposed solution is a **multi-agent framework** in which each agent is an LLM prompt tailored to a specific role in the software development process. The agents work together, passing artifacts (textual plans, code, test results) to one another in a pipeline that transforms an initial idea into a finished application. The primary agents and their roles are:

- **Requirements Analyst Agent:** This agent takes the initial user input – a high-level idea or feature request – and translates it into a structured software requirements specification. It acts like a business analyst, clarifying ambiguities and asking for any needed details. For example, if the user says "I need an app to manage personal tasks with user login," the Requirements agent will produce refined requirements or user stories (e.g. "User can create an account with email/password," "User can add, edit, delete tasks," etc.). It may also list acceptance criteria for each feature. The output of this stage is a clear, expanded problem definition that will guide the rest of the agents. *By ensuring the requirements are well-defined upfront, we reduce the likelihood of misunderstandings propagating through development.* (This corresponds to roles identified in prior research, where an LLM agent handles requirement analysis in natural language [7] .)

- **Architect Agent:** Using the requirements specification, the Architect agent designs a high-level solution. This includes choosing the system architecture, outlining the components or modules, selecting tech stack details (if not already specified), and formulating an overall plan for implementation. It might produce diagrams or structured text describing the database schema, API endpoints, class structures, or UI component hierarchy. Essentially, this agent plays the role of a software architect or lead developer – it decides *how* to implement the requirements. For instance, it could decide to implement a REST API in Node.js with Express, a React frontend with certain state management, and a Flutter mobile app interfacing with the same API. The Architect agent's output will serve as a blueprint for the Developer agent. Having an explicit architectural plan in natural language (or pseudocode) ensures the Developer agent has a strong guiding context, much like a human developer following a design document. *(Notably, giving the LLM a "planning" role has been shown to improve subsequent code generation – e.g., a planning agent's output can guide a coding agent to be more effective [8] .)*

- **Developer Agent:** This agent is responsible for writing the actual code (in multiple languages as needed) to implement the features according to the design and requirements. There may in fact be multiple Developer sub-agents – e.g., one specialized in backend Node.js code, another in front-end React code, another in Dart/Flutter – or a single agent that handles all coding sequentially. For clarity, we refer to it collectively as the Developer agent(s). The Developer agent takes input from the Architect (and the refined requirements) and generates code files for each component. It will produce backend code (server logic, database calls), frontend code (UI components, state management), and any other necessary source files (configurations, etc.). This is done through prompt engineering that instructs the LLM to output code in the required language and format. The Developer agent works iteratively: it might create an initial version of each module and then refine them if needed (especially after testing). By dividing coding tasks (front-end vs back-end) or using specialized prompts for each, the framework can leverage the LLM's ability to handle one focused task at a time, which helps manage complexity and context length. *This mirrors how in human teams, different developers or sub-teams work on different parts of the codebase concurrently.* The Developer agent's output is a set of code artifacts that, in principle, should fulfill the outlined design and requirements.

- **Tester Agent:** Once code is generated, the Tester agent steps in to verify its correctness and quality. It generates test cases and test code (e.g., unit tests or integration tests) based on the requirements and acceptance criteria given by the Requirements agent. It then executes these tests (in an automated test environment) to see if the Developer agent's code passes them. The Tester agent role is akin to a QA engineer who ensures the product meets the specifications. If tests pass, this agent confirms the solution is correct. If some tests fail, the Tester agent communicates the results (and possibly diagnostic feedback) back to the Developer agent for debugging. This triggers an iteration where the Developer agent fixes the bugs and resubmits code, and testing runs again – an automated test-and-debug cycle. *Such feedback loops are crucial for autonomy: the system can catch and correct mistakes without a human in the loop* [18]. The presence of a dedicated testing agent also addresses the known issue of LLMs hallucinating plausible but incorrect outputs [15] – by checking against test oracles, the framework doesn't trust code until it's validated.

These agents collaborate through a **prompt orchestration system** that we design. The orchestration ensures that each agent's output is fed as context into the next agent's prompt. For example, after the Requirements agent produces the specification, the system will construct a prompt for the Architect agent along the lines of: *"Given the following requirements, design a software architecture…"* including the spec text. Similarly, the Developer agent's prompt will include the architecture plan and relevant requirements: *"Generate the code for the following component as per the design…"*. The Tester agent receives the requirements and perhaps the code (or at least function signatures) to know what to test. In essence, the prompts are the messages the agents exchange, and this forms an **inter-agent communication protocol in natural language**. Each agent "speaks" in the form of its written output, and the others "listen" by having that output included in their input context. This approach leverages LLMs' strength in language understanding to enable them to interface with each other. As prior work highlights, assigning distinct roles with modular goals to different agents and letting them converse can greatly enhance overall performance [19] [6]. We will implement a coordinator (a simple program) that manages this flow: triggering agents in sequence or loop and collating their outputs.

Throughout the process, **context and memory management** is important. The framework will maintain a shared workspace of artifacts: the refined requirements, the architecture design, the evolving codebase, and test results. Agents may query this shared context. For instance, if a later agent needs to recall a piece of the requirements, it will be provided. Techniques like summary prompts or retrieving relevant portions of context will be used to cope with LLM context window limits (drawing inspiration from solutions like ALMAS's use of code summaries [20], though detailed implementation of that is secondary in our current scope). The collaborative design ensures that each agent focuses on its specialty but remains **aware of the broader project context**, as communicated through the prompts. This aligns with the "three Cs" principles (Context-aware, Collaborative, Cost-effective) noted by Tawosi et al. (2025) – specialized agents must seamlessly communicate with one another (and with humans if needed) to reduce cognitive load and improve productivity [21].

To illustrate the end-to-end flow: suppose a user provides a high-level goal to the system. The Requirements Analyst agent refines it into, say, a set of 10 user stories. The Architect agent then creates a system design, perhaps deciding on 5 modules and an API contract between front-end and back-end. The Developer agent then writes code for each module (maybe sequentially). Once code is ready, the Tester agent generates test cases for each user story and runs them – finding, for example, that 2 tests failed. The framework then prompts the Developer agent with the **failure information** (e.g., "Test X failed, expected output Y but got Z") and asks it to fix the code. The Developer agent debugs (like a human programmer would upon a failing unit test) and produces a patch. The tests run again and now all pass. Finally, the system outputs the complete codebase (and possibly the test suite) to the user. At this point, we have an application that meets the requirements as far as our tests can determine. Throughout this process, minimal human input was needed after providing the initial prompt – the agents negotiated the solution among themselves. This kind of autonomous, coordinated workflow is what sets our framework apart from existing AI coding tools.

It's worth noting that while we define four main agents here, the framework is extensible. One could add a **Documentation Agent** to generate user documentation or a **Refactor Agent** to improve code style, etc., as needed. The principle is the same: each new agent would consume the appropriate context and produce a domain-specific output. Our design aligns with agile team roles for modularity [19] but remains flexible in agent composition. By mirroring real software team dynamics (analyst, architect, developer, tester), we aim to ensure that the multi-agent system covers the critical skills needed for development [22] [19]. This human-team analogy also aids in prompt design – we instruct each agent in terms of the responsibilities analogous to a job role, which is intuitively understood by LLMs and keeps their outputs focused.

In summary, the proposed framework is a **pipeline of LLM-driven agents** that transforms requirements into architecture, architecture into code, and code into a tested product, all via natural language interactions. The innovation lies in structuring these prompts and agents such that the *LLMs collaborate effectively*, with the output of one becoming the input for another, much like a relay team. This division of labor and iterative refinement is expected to lead to more accurate and coherent outcomes than a single LLM attempting everything in one go [6] [17]. The next section will detail how we plan to implement and evaluate this system to validate these claims.

Methodology

To investigate the effectiveness of the proposed framework, we will employ a combination of **qualitative analysis**, **quantitative experiments**, and **comparative evaluations**. Our methodology is structured as follows:

1. Prototype Development: First, we will implement the multi-agent framework as a research prototype. This involves writing the orchestration code that manages prompts and agents, and integrating it with a large language model API. We will likely use a model such as GPT-4 (or an equivalent advanced LLM available) for all agents, modifying the prompt for each role. The prompt templates for each agent will be carefully crafted and tuned through small-scale tests. For example, we might perform pilot runs on a simple "to-do list app" requirement to refine how the Requirements agent asks clarifying questions or how the Developer agent formats its code output. During this stage, we will document qualitatively how the agents behave, identify any failure modes (e.g., the Developer agent ignoring part of the design), and improve the prompt strategy. Techniques like few-shot prompting or step-by-step reasoning may be employed where beneficial (e.g., instructing the Developer agent to first outline its plan in comments before coding). The outcome of this phase is a working system that can autonomously attempt to build non-trivial (but small-scale) applications.

2. Experimental Design: With the prototype in hand, we will conduct experiments on a set of **realistic project prompts**. These prompts will describe target applications in natural language, akin to what a client or product manager might give. We will assemble a diverse set of, say, 5–10 project scenarios within our scope (e.g., *"A library management system with a web UI for searching and a backend for tracking books and users"*, *"A personal finance tracker mobile app with user authentication and charts of expenses"*, etc.). For each scenario, we will run our framework end-to-end to generate the software. We will then evaluate the results on multiple criteria. Importantly, we will also have baselines to compare against: - **Manual**

Implementation Baseline: For a subset of the projects, we (or hired developers) will implement the solution manually (or use an existing reference implementation if available). This provides a ground truth for code correctness and an estimate of the effort required without AI. - **Traditional AI Assistance Baseline:** We will use a single-agent approach (e.g., prompting ChatGPT or Copilot directly with the same project description) to see how far a *one-shot* or single-agent solution goes. In practice, this might involve asking ChatGPT to output the full code for the project in one prompt, or using Copilot continuously to build the project. This baseline represents the current typical use of AI in coding without our structured framework.

3. Metrics Collection: We will evaluate each generated project on several **quantitative metrics**: - **Development Time**: How long does the framework take from receiving the prompt to producing a working codebase? This will be measured in wall-clock time and possibly broken down by agent cycles. We will compare this to the manual baseline (e.g., person-hours saved) and to the single-agent AI baseline. - **Prompt Iterations**: How many back-and-forth cycles occurred, especially between Developer and Tester agents? Fewer iterations might indicate the framework converged quickly, while many iterations might indicate difficulty. We log each agent's invocation count. - **Code Correctness and Functionality**: Primarily measured by the **test success rate**. Since the Tester agent is generating tests from requirements, we use those as an objective measure – e.g., "8 out of 8 critical user stories passed their tests" indicates full functional correctness with respect to stated requirements. We may supplement this with running additional predefined tests or doing manual testing to catch anything the AI tests missed. - **Code Quality**: We will analyze the generated code for readability, maintainability, and efficiency. This can be done via automated static analysis tools or metrics (like cyclomatic complexity, code duplication percentage, etc.) and via human expert review. For example, we might have independent developers rate the code on a scale for clarity and style. We will also check for the presence of obvious bad practices or vulnerabilities. Prior work noted that AI-generated code can sometimes include security flaws [23], so we will inspect whether the framework's iterative process mitigates that (perhaps by the Tester agent catching functional issues, though security might be beyond its scope). - **Error Rate/Bug Count**: If any bugs are discovered (through tests or later inspection), we count them. We compare the number of post-generation fixes needed in our framework vs. the baselines. - **Usability and Effort**: Although our framework is autonomous, we will measure the *human effort* required to use it. This includes the complexity of the initial prompt authoring and any interventions needed. If possible, we will conduct a small **user study**: give a few non-programmers and a few programmers a chance to try the system (with some training) and gather their feedback via a questionnaire. We'll ask them about perceived ease-of-use, whether they trust the generated code, and how it compares to either coding themselves or using Copilot. This qualitative data will gauge the framework's practicality for end-users.

4. Comparative Analysis: We will compare our framework's performance with the baselines on all the above metrics. For instance, if our framework reduces development time by 50% compared to manual coding, and produces code passing 95% of tests on the first try, that's a strong result. If the single-agent approach (ChatGPT one-shot) fails many tests or produces incoherent structure while our multi-agent approach succeeds, that demonstrates the value of the structured prompts. We will use appropriate statistical analysis if sample size allows – for example, if we have multiple runs or multiple tasks, we can compute average metrics and see if differences are significant.

We will also examine **failure cases** in detail. If the framework struggled on a particular requirement (say it misinterpreted a specification or an agent got stuck in a loop), we perform a case study analysis of that scenario. We'll trace the conversation between agents to identify where the breakdown occurred (e.g., the

Requirements agent failed to clarify an edge case, leading to a design flaw). This qualitative error analysis will inform potential improvements and also highlight the framework's limits. It's expected that complex logic or very creative tasks might be hard for the AI agents to handle fully – documenting these cases will be part of the evaluation.

5. Qualitative Evaluation: Beyond metrics, we will evaluate the **quality of the development experience** and the final product qualitatively. For the experience, we consider how understandable the agent outputs are (since eventually a human maintainer might read the generated design or code). We'll check if the architecture decisions made by the AI align with reasonable human practices, or if they are convoluted. For the final application, we consider if it meets not just the letter of the requirements but the intent – for example, was the user experience sensible in the React app? These aspects might require human judgment to evaluate, so we plan to involve a small panel of software engineers to do a blind review of projects (without knowing if a project was AI-generated or human-coded, if feasible) to rate their quality. This can help address subtle aspects of "usability" of the code and product that pure metrics might not capture.

During evaluation, we will also track *which aspects of the framework contributed most to success*. For example, we might observe that the planning by the Architect agent greatly helped the Developer agent avoid mistakes, or conversely that the framework needed multiple rounds because the initial requirements were ambiguous. These observations will be crucial for the Discussion section of the thesis, where we reflect on the approach.

In summary, our methodology is to **build** the system, **test it on representative scenarios**, and **measure** its performance from multiple angles. We will compare it with alternative approaches to substantiate claims of improvement. This mix of quantitative rigor and qualitative insight will enable us to answer the core research questions: - Can a prompt-based multi-agent LLM system reliably produce working software? - How efficient and correct is it compared to traditional development (with or without AI assistance)? - What limitations or failure patterns emerge, and how might they be addressed?

By following this plan, we aim to provide a thorough evaluation of the framework's viability. All experimental results, including example agent dialogues and generated code, will be documented to illustrate how the framework operates in practice. This methodology will validate the framework's contributions and clarify its potential impact on software engineering workflows.

Expected Outcomes

We anticipate several key outcomes from this research, which, if realized, would demonstrate the value of the proposed framework:

- **Significant Reduction in Development Time:** The multi-agent LLM framework is expected to dramatically accelerate the software development process. By automating requirement analysis, coding, and testing, the end-to-end time from an idea to a working application could be reduced from days or weeks (in manual development) to potentially hours. For example, instead of a developer writing code and tests for each feature sequentially, our agents work in parallel (the Requirements and Architect agents set the stage while the Developer agent then produces code much faster than a human typing). We expect to measure notable time savings in our experiments – possibly an order-of-magnitude speedup for small-to-medium projects. This outcome aligns with prior observations that AI co-developers can streamline workflows and accelerate project timelines

[18]. A successful result would be that our framework consistently delivers functional software in a short, predictable timeframe, freeing humans from lengthy coding sessions.

- **High Accuracy and Code Quality (Functional, Testable Outputs):** We aim for the framework to produce **fully functional codebases with minimal human corrections needed**. In practice, this means the code generated should pass the automatically generated test suite and meet the specified requirements right away, or after only a few internal fix iterations. The inclusion of the Tester agent is designed to ensure this. We expect that, in our evaluations, most if not all of the critical user stories will be satisfied by the AI-generated application. If the framework works as intended, the final output should be *not just boilerplate code*, but a correctly working system (e.g., one can run the Node.js server and the React app and see that all described features operate). Achieving a high test pass rate is a primary success criterion. Additionally, we anticipate that code quality in terms of structure and readability will be reasonable. The Architect agent's guidance should lead to modular, well-organized code, and the iterative debugging should eliminate many logical errors. Past research has shown that multi-step AI approaches can substantially improve correctness – for instance, multi-agent code generation methods have achieved very high problem-solving success rates [11]. Our expected outcome is that by orchestrating planning, generation, and testing, we will similarly attain a level of code quality that rivals human-written code for the tasks at hand.

- **Improved Efficiency vs. Traditional AI-Assisted Approaches:** We expect our framework to outperform a single-agent prompting strategy (like one-shot ChatGPT coding) on multiple fronts. The multi-agent design should handle complex or long requirements better because each agent keeps the task focused and clear (mitigating issues like context length limits or prompt confusion). We predict fewer mistakes and rewrites compared to a monolithic approach, and our evaluation will likely show a higher success rate for projects completed by the multi-agent system. In essence, this framework should demonstrate how prompt-based role specialization yields *more reliable and scalable* AI-driven development. We also anticipate that the runtime debugging cycle through the Tester agent will catch errors that a single-pass generation would miss, resulting in more reliable outputs [6] [24].

- **Usability and Accessibility of Software Creation:** A broader expected outcome is that this framework can make software development more accessible to those with little or no coding experience. By allowing a user to simply describe what they want and letting the AI agents handle the technical implementation, we lower the barrier to realizing software ideas. We expect to show through our user trials or anecdotal examples that even non-developers can obtain a working app by using the framework, essentially performing as a no-code or very-low-code platform powered by AI. This could be transformational – for example, a small business owner could "write" their custom app by literally writing a natural language prompt. While professional oversight is still valuable (especially for edge cases or fine-tuning the product), the heavy lifting of programming is done by the AI. We anticipate feedback indicating that users find the framework's interface (the prompts and agent outputs) understandable and the process intuitive, after some initial learning. If successful, our system will serve as a prototype of how future software engineering might allow domain experts to create software *without deep programming knowledge*, by leveraging conversational AI.

- **Challenges and Learnings:** We also expect to document the limitations discovered. For instance, the framework might struggle with very complex logic or have difficulties if requirements are incomplete. We might observe issues like inter-agent miscommunication or prompt misalignment that require careful handling (e.g., an agent misunderstanding another's output). These findings are an outcome as well, as they will highlight areas for future improvement. A likely observation is the sensitivity of the process to prompt phrasing – small changes in how instructions are given can affect output quality. We will report such insights, contributing to the body of knowledge on prompt engineering and multi-agent coordination. Additionally, we expect to discuss the computational cost of the approach (using multiple large-model calls) and whether the speed-ups in development time are offset by increased usage of computing resources, addressing the "Cost-effective" aspect mentioned in some studies [21].

In quantitative terms, a successful outcome might be: *For a given set of benchmark projects, the framework is able to autonomously deliver complete solutions with over 90% of predefined tests passing on the first try, and with development time less than 20% of the manual coding time.* We also expect subjective outcomes like positive user feedback and maintainers finding the generated code reasonable to work with. Ultimately, if our hypothesis is correct, the evaluation will conclude that **prompt-based multi-agent LLM systems can feasibly automate the bulk of software development tasks** for certain classes of projects, with only light human supervision needed. This would be a noteworthy step beyond the current state of AI coding tools, moving closer to the vision of an AI system that *collaboratively builds software* much like a human team, and delivering tangible productivity and accessibility benefits.

We will validate these outcomes with evidence in the thesis: timing measurements, test results, code excerpts, and user quotes. For example, we might include a table comparing results of manual vs. our framework vs. single-agent for each project, or a case study walkthrough showing how our agents fixed a bug that a single-pass approach left in. We expect to see results echoing the literature – Ashraf and Talavera (2025) found multi-agent LLMs reduced human errors and accelerated timelines [18], and our work will likely reinforce those findings in our context. If some outcomes are not fully met (e.g., if human intervention was needed more than expected), that too will be reported along with analysis, serving as valuable feedback for future research in this direction.

In summary, the project hopes to demonstrate a **working prototype of automated software development** that achieves faster development, correct and testable code, and a user-friendly interface for specifying software tasks. Achieving these expected outcomes would underscore the thesis that orchestrated LLM agents, guided by structured prompts, can transform how software is engineered, pointing toward a future where AI systems take on much of the programming workload in collaboration with humans.

References

- Ashraf, B., & Talavera, G. (2025). *Autonomous Agents in Software Engineering: A Multi-Agent LLM Approach*. [Preprint]. This work explores the use of multiple LLM-powered agents (for requirements, coding, testing, etc.) in automating software development, finding efficiency gains and discussing challenges of coordination [25] [24].

- Ashrafi, N., Bouktif, S., & Mediani, M. (2025). *Enhancing LLM Code Generation: A Systematic Evaluation of Multi-Agent Collaboration and Runtime Debugging for Improved Accuracy, Reliability, and Latency*. arXiv: 2505.02133. This study combines a multi-agent coding strategy with an automated debugging loop, showing that a planner agent guiding a coder agent, followed by execution feedback, improves code accuracy and reliability [6] [26].

- Chen, M., Tworek, J., Jun, H., Yuan, Q., et al. (2021). *Evaluating Large Language Models Trained on Code*. arXiv:2107.03374. OpenAI's paper introducing Codex (the model behind GitHub Copilot), which demonstrates the capability of LLMs to generate code from natural language and highlights the model's performance (solving ~28.8% of problems on first attempt) and limitations [1].

- Fan, A., Gokkaya, B., Harman, M., Lyubarskiy, M., Sengupta, S., Yoo, S., & Zhang, J. M. (2023). *Large Language Models for Software Engineering: Survey and Open Problems*. [Meta Research Technical Report]. This survey provides an overview of LLM applications in software engineering and discusses challenges such as hallucination and verification in AI-generated code [15].

- Islam, M. A., Ali, M. E., & Parvez, M. R. (2024). *MapCoder: Multi-Agent Code Generation for Competitive Problem Solving*. In *Proceedings of the 62nd Annual Meeting of the ACL (Volume 1: Long Papers)* (pp. 4912–4944). ACL. This paper introduces a framework with four LLM agents (retrieval, planning, coding, debugging) that collaborate to solve programming tasks, achieving state-of-the-art results on several code generation benchmarks [11].

- Sahoo, P., Singh, A. K., Saha, S., Jain, V., & Mondal, S. (2025). *A Systematic Survey of Prompt Engineering in Large Language Models: Techniques and Applications*. arXiv:2402.07927 (v2). This survey explains prompt engineering as an indispensable technique to steer LLM behavior using task-specific instructions, enabling models to perform a wide range of tasks without additional training [5].

- Tawosi, V., Ramani, K., Alamir, S., & Liu, X. (2025). *ALMAS: An Autonomous LLM-based Multi-Agent Software Engineering Framework*. In *MAS-GAIN Workshop at ASE 2025*. (arXiv:2510.03463). This work proposes a vision of an agile-aligned multi-agent LLM framework covering multiple SDLC stages. It highlights the need for integrated ecosystems (not isolated tools) and aligns agents with roles like product manager, developer, tester [19], demonstrating a prototype that adds a new feature to an application autonomously [27].

- Zhang, B., Liang, P., Zhou, X., Ahmad, A., & Waseem, M. (2023). *Practices and Challenges of Using GitHub Copilot: An Empirical Study*. In *Proc. of the 35th Intl. Conf. on Software Engineering and Knowledge Engineering (SEKE 2023)*. This empirical study analyzes discussions from developers on Copilot's usage. It finds that while Copilot can increase productivity, its main limitations include integration difficulties and the need for careful review of AI-suggested code [28] [16].

- Zhang, Y., & Sun, R., et al. (2024). *Chain-of-Agents: Large Language Models Collaborating on Long-Context Tasks*. In *NeurIPS 2024*. (OpenReview Preprint). This paper (and corresponding Google AI blog, 2025) presents a multi-LLM collaborative framework that divides long tasks among several agents and a manager agent. It demonstrates improved performance (up to 10% gain) over single-model baselines on tasks like long document QA and code completion, by enabling agents to communicate and handle segments of the input [9] [10].

---

[1] [2107.03374] Evaluating Large Language Models Trained on Code
https://arxiv.org/abs/2107.03374

[2] [3] [13] [14] [17] [19] [20] [21] [27] [2510.03463] ALMAS: an Autonomous LLM-based Multi-Agent Software Engineering Framework
https://ar5iv.labs.arxiv.org/html/2510.03463

[4] [16] [28] arxiv.org
https://arxiv.org/pdf/2303.08733

[5] [2402.07927] A Systematic Survey of Prompt Engineering in Large Language Models: Techniques and Applications
https://arxiv.org/abs/2402.07927

[6] [8] [26] Enhancing LLM Code Generation: A Systematic Evaluation of Multi-Agent Collaboration and Runtime Debugging for Improved Accuracy, Reliability, and Latency
https://arxiv.org/html/2505.02133v1

[7] [12] [18] [22] [24] [25] (PDF) Autonomous Agents in Software Engineering: A Multi-Agent LLM Approach
https://www.researchgate.net/publication/388834987_Autonomous_Agents_in_Software_Engineering_A_Multi-Agent_LLM_Approach

[9] Chain of Agents: Large language models collaborating on long-context tasks
https://research.google/blog/chain-of-agents-large-language-models-collaborating-on-long-context-tasks/

[10] openreview.net
https://openreview.net/pdf?id=LuCLf4BJsr

[11] aclanthology.org
https://aclanthology.org/2024.acl-long.269.pdf

[15] [23] coinse.github.io
https://coinse.github.io/publications/pdfs/Fan2023yu.pdf