Decomposing Complexity: An LLM-Based Approach to Supporting Software Engineering Tasks

Zhijie Xu

CMU-CS-25-132

August 2025

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Carolyn Rosé, Chair
Michael Hilton

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science.

Keywords: Task Decomposition, Coding Agents, Software Engineering Automation, GitHub Issue Management, Open-Source Onboarding

To everyone who stood by me, offered guidance, and helped me grow.

Abstract

Task decomposition in software engineering enables the division of complex engineering tasks into manageable components, facilitating modularization and collaborative development. However, supporting newcomer onboarding in open source projects remains challenging, as complex issues often assume substantial domain knowledge that prevents meaningful contributions. While maintainers understand the value of providing entry-level tasks, manually creating approachable entry points competes with other development demands. In this work, we investigate task decomposition as a foundation for human-augmentation, creating and analyzing a dataset of decomposition patterns across ten Apache projects that reveals how experienced developers naturally break down complex tasks into 3 different patterns.

Building on these insights, we integrate a decomposition component into SWE-agent to validate that structured task breakdown creates genuine problem-solving value. Our system achieves a 24% performance improvement over the non-decomposed baseline on SWE-Bench verified dataset. While evaluation focused on AI agents rather than human contributors, this technical validation provides necessary evidence that decomposition creates structural value. This research reframes newcomer onboarding from "finding newcomer-oriented tasks" to "creating navigable pathways into meaningful work", establishing the foundation for validating decomposition benefits through human studies with real newcomers in live open source projects.

Acknowledgments

I would like to express my heartfelt gratitude to my advisor, Professor Carolyn Rose, for her exceptional guidance, support, and encouragement throughout this research. Her insights and mentorship have been invaluable to the completion of this thesis. I am grateful to Professor Michael Hilton for serving on my thesis committee and for his thoughtful feedback that strengthened this work. I want to thank Yiqing Xie and Atharva Naik for their helpful suggestions and engaging discussions that significantly shaped the direction of this research. My sincere appreciation goes to Yike Tan for actively engaging in technical discussions and generously providing resources that enabled my experiments, and to Yunze Xiao for their encouragement throughout this process. Finally, I am deeply grateful to my parents for their unwavering financial and emotional support. Their confidence in me has been a source of strength throughout my academic journey.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Open Source Software (OSS) projects form the backbone of modern software development, powering everything from web frameworks to operating systems. The collaborative nature of these projects depends critically on maintaining a steady influx of contributors who can understand, modify, and extend complex codebases [33]. However, the long-term success of OSS ecosystems depends not only on attracting new contributors, but also on effectively helping them become active and productive members. This process, known as onboarding, is a core challenge that influences both the sustainability and innovation of a project [22].

The transition from newcomer to active contributor in OSS projects involves overcoming numerous challenges. Among these challenges, the difficulty of finding suitable initial tasks represents a particularly critical bottleneck. Complex issues often assume domain knowledge and familiarity with project architecture, creating a substantial cognitive load that prevents newcomers from making meaningful contributions. Although maintainers understand the value of providing beginner-friendly tasks, the time and effort needed to define and organize these opportunities often compete with other development demands, limiting newcomers' chances to get involved.

Current approaches to supporting newcomer onboarding focus primarily on issue classification[36]. These approaches attempt to identify existing issues that might be suitable for newcomers. However, classification-based approaches are fundamentally limited by their reliance on existing task granularity. Accurate labeling does not make complex tasks easier for newcomers . They treat task suitability as fixed. This overlooks the possibility that task suitability can be actively shaped to better support human contributors.

The core challenge is not identifying suitable tasks, but transforming complex issues into structured, learnable pathways that support understanding and skill development. Task decomposition addresses this by breaking down complex problems into hierarchies of manageable components. This creates entry points aligned with varying skill levels while preserving connection to meaningful project goals, enabling newcomers to contribute meaningfully as they build expertise.

This thesis addresses the newcomer onboarding challenge by developing a human-augmentation framework centered on systematic task decomposition. The complete solution pathway involves three steps: (1) understanding how experienced developers naturally decompose complex tasks, (2) validating that automated decomposition can produce meaningful task breakdowns, and (3) demonstrating that such decompositions effectively support human learning and contribution in real-world settings.

We focus on the first two steps. Through an empirical analysis of decomposition practices

across Apache projects, we uncover patterns that characterize how experienced developers structure complex tasks. These insights inform the design of LLM-based decomposition capabilities, which we integrate into SWE-agent [40] — a state-of-the-art software engineering agent. The system interprets natural language requirements, reasons about code structure, and transforms GitHub issues into structured subtasks. Evaluated on the SWE-Bench Verified dataset, this approach yields a 24% improvement over the baseline without decomposition.

While our ultimate goal is to support human contributors, evaluating decomposition through automated agents serves as a necessary precursor — if structured decomposition fails to benefit AI systems with vast computational resources, it is unlikely to aid human newcomers with more limited capabilities. The demonstrated performance gains suggest that systematic task breakdown creates meaningful structure for complex problem-solving, establishing a foundation for future work that directly supports human contributors in navigating and contributing to large-scale software projects.

Chapter 2

Background and Related Work

2.1 A History of Task Decomposition in Software Practice

Task decomposition in software engineering has evolved from hierarchical methodologies to more systematic and hybrid approaches across project planning, requirements engineering, and development management. Foundational work such as The Mythical Man-Month [11] and Work Breakdown Structures [1] established early principles for organizing and dividing software tasks. The Structured Analysis and Design Technique (SADT) [16] introduced functional decomposition through diagrammatic methods, while requirements engineering advanced decomposition through hierarchical refinement of stakeholder needs into testable requirements [14].

Modern practices integrate these foundations with agile methods. User story decomposition — via CRUD splitting, business rule extraction, or workflow-based methods — has become central in agile development [25]. Empirical studies suggest Scrum with Kanban and XP support more effective decomposition than Scrum alone [37]. Issue tracking systems reinforce this through Epic $\rightarrow$ Story $\rightarrow$ Task hierarchies, with collaborative decomposition seen as a problem-solving process [8]. Requirements traceability remains key for connecting tasks to implementation and verification [2].

Despite methodological advances, challenges remain in balancing granularity, managing dependencies, and adapting decomposition techniques to evolving contexts. Literature consistently highlights the importance of hierarchical structures, collaboration, and traceability in managing complexity [31][35]. Current trends reflect a hybrid of traditional and agile methods. Future work should empirically assess decomposition effectiveness, develop selection guidelines, and explore integration with emerging tools [17].

2.2 Modern Approaches: LLM for Task Planning and Decomposition

The emergence of large language models has enabled advances in task planning for software agents, allowing them to generate execution strategies for complex programming tasks by sequencing subtasks and deciding when to invoke external tools [4]. These planning capabilities differ fundamentally from the goal of this work: structural task decomposition, which focuses on

breaking down a complex problem into logically coherent, self-contained subproblems that reflect the underlying problem structure.

Unlike traditional approaches based on predefined workflows, LLM-based planning leverages natural language understanding and code comprehension to generate context-aware execution plans [42]. Current systems follow diverse architectural paradigms. Single-agent frameworks like SWE-Agent manage internal state transitions while sequencing actions [40]. Multi-agent systems such as MASAI employ hierarchical sub-agents to coordinate tool usage and decision-making, achieving 28.33% resolution on SWE-bench [4]. Tool-augmented frameworks like CodeAgent demonstrate significant efficiency gains by incorporating retrieval and navigation tools, improving pass rates by 2.0–15.8 percentage points while reducing token usage (56.9K vs. 138.2K) [42].

These systems typically rely on planning-oriented reasoning strategies such as Chain-of-Thought (CoT) reasoning, which improves performance in multi-step inference tasks by guiding execution flow [39], and Modularization-of-Thought (MoT), which constructs hierarchical reasoning graphs to inform planning decisions [28]. Adaptive frameworks like ADaPT dynamically adjust execution plans based on intermediate tool outcomes [19].

Despite technical gains in planning and tool integration, current LLM-based systems focus primarily on execution optimization rather than structural problem analysis—i.e., how to identify and isolate the fundamental subproblems within a complex software engineering task. While planning methods achieve impressive automation results, they operate primarily at the procedural level, lacking mechanisms for generating decompositions that reveal the conceptual architecture of the original problem.

This gap becomes particularly relevant when considering interpretability and modularity in AI-assisted development. Current approaches optimize for end-to-end task completion but provide limited insight into why certain decompositions are chosen or how the subproblems relate to each other structurally. Understanding these structural relationships could enable more effective human-AI collaboration and provide a foundation for iterative problem-solving approaches that go beyond sequential execution.

Furthermore, current limitations suggest the need for alternative approaches. State-of-the-art models solve fewer than 40% of issues on SWE-Bench [21], and planning failures often cascade without effective structural understanding of the problem space [19]. In multi-agent settings, poor coordination leads to tool conflicts or incomplete coverage [23], and these challenges motivate exploring decomposition methods that prioritize structural clarity alongside execution efficiency.

Chapter 3

Analyzing Real-World Task Decomposition: A Exploratory Dataset

3.1 Motivation and Dataset Selection Criteria

Current research in SE task decomposition lacks a comprehensive dataset that reflects real-world practices. Existing task planning datasets, such as TaskBench [34] and AsyncHow [24], primarily target general task automation or asynchronous planning in everyday scenarios. While TaskBench focuses on API tool usage across synthetic domains, and AsyncHow emphasizes reasoning over procedural dependencies in instructional tasks, neither captures the granularity, ambiguity, and domain-specific context found in real-world software development. As a result, these benchmarks fall short in modeling how developers decompose complex software issues into actionable subtasks within platforms like GitHub, leaving a gap between academic research and practical applications in software engineering.

To address the lack of structured datasets for software engineering task decomposition, we conducted a review of over 30 popular open-source repositories on GitHub, including projects such as PyTorch [38] and uv [5]. While many of these repositories employ issue checklists to track development progress, we found that the majority of checklists are informal and consist of scattered notes or reminders, lacking the systematic decomposition of complex tasks into well-defined subtasks. GitHub's recently introduced sub-issue feature, offers more structured task hierarchies but remains in limited beta and has seen minimal adoption across large-scale projects.

Consequently, we selected Apache Software Foundation projects as our primary data source due to their structural consistency, data quality, and domain diversity. Unlike the fragmented issue tracking systems across GitHub repositories, Apache projects uniformly use JIRA [6], which provides built-in support for hierarchical task relationships with clear parent-child linkages — an essential feature absent in most GitHub issues.

Moreover, Apache projects are maintained by experienced contributors operating under well-defined development practices. Each task is typically accompanied by corresponding commits and code changes, offering concrete examples of how human practitioners decompose and execute tasks in real-world software development contexts, providing training data for learning decomposition patterns from established practices.

Spanning domains such as web servers, databases, and machine learning frameworks, the 10

selected Apache projects[1] ensure our dataset reflects a wide range of software engineering contexts across both project types and historical periods from July 24, 2005, to November 12, 2024. This combination of structural rigor, maturity, temporal coverage, and domain diversity provides a solid foundation for studying real-world task decomposition in software engineering.

3.2 Data Collection and Preprocessing Pipeline

Our data collection methodology employed a systematic four-stage pipeline to extract and validate task decomposition patterns from Apache Foundation projects.

Stage 1: Project Analysis and Selection We first analyzed all Apache Foundation projects to identify those with significant task decomposition practices. Using Jira's REST API, we extracted structured metadata from 364 Apache Foundation projects and stored in MongoDB. The data schema can be refered to table 3.1. To ensure good quality and richness of task decomposition types, we set a threshold of 150 parent tasks, resulting in the selection of only 10 projects with sufficient hierarchical task structures for meaningful decomposition analysis. The remaining projects have insufficient subtask utilization for meaningful decomposition analysis, leading us to focus exclusively on these high-utilization projects.

Field	Description
`id`	Internal unique identifier for the parent issue
`issue`	Title of the parent task
`issue_description`	Description of the parent task
`key`	Jira issue key (e.g., HIVE-24369)
`project`	Project name (e.g., HIVE)
`subissues`	List of subtasks, each with fields: `name`, `key`, `id`, and `links` (PRs)

Table 3.1: Schema Structure

Stage 2: Project Selection and Hierarchical Filtering From the selected 10 projects, we extracted all parent-child task relationships using JIRA's native issue linkage system through the REST v2 interface. To ensure useful decomposition patterns, we applied structural filtering criteria: each parent issue required a minimum of two subtasks, and acyclicity constraints eliminated circular dependencies within hierarchical structures. This initial extraction yielded 3,765 parent tasks associated with 37,773 subtasks.

Stage 3: Evidence Validation To ensure that decompositions reflected actual engineering practices rather than theoretical planning, we implemented implementation evidence filtering as our primary quality criterion. Each subtask was required to include associated commit messages or pull

[1]FLINK, SPARK, HBASE, BEAM, HIVE, YARN, HDFS, CB, OFBIZ, HADOOP, and IGNITE

request references, demonstrating substantiated development work. This criterion excluded purely theoretical or abandoned decompositions while retaining those with verified implementation evidence. Note that parent issues were not subject to implementation requirements, as they primarily serve organizational functions in coordinating subtask execution. This resulted in a dataset where approximately 92% of subtasks are directly connected to code contributions. The filtering process yielded 2,151 issue pairs. As shown in Figure 3.1, the distribution of subtask counts among these issues is right-skewed, with the 85th percentile at 12 subtasks — indicating that most tasks are decomposed into a relatively small number of subtasks.

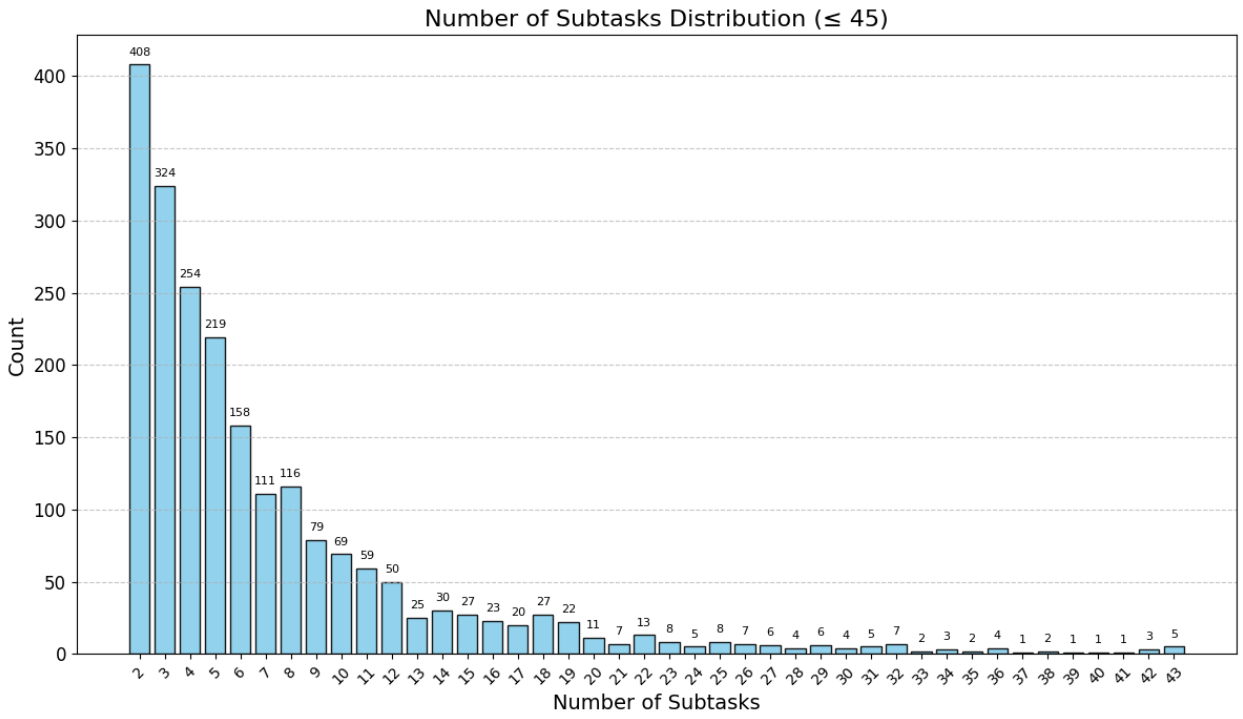

Figure 3.1: Distribution of Number of Subtask

Stage 4: Quality Assurance and Enhancement To ensure data quality, we performed deduplication based on issue IDs to eliminate redundant entries, such as repeated maintenance tasks appearing across multiple projects.

Many subtask descriptions contained abbreviations, pronouns, and context-dependent references that impede comprehension and confuse language models [20]. We employed GPT-4.1 [26] to enhance task descriptions while preserving semantic meaning. The enhancement process utilized five input sources: parent task, subtasks, subtask descriptions, associated commit files, and relevant discussion threads. This context aggregation approach enabled the model to expand abbreviations based on project context, resolve pronouns through discussion analysis, and transform cryptic references into clear, actionable descriptions, forming the final dataset for subsequent analysis.

Table 3.2 presents some examples of this enhancement process. Notable cases include ”BB,” which could ambiguously refer to BlackBerry, black box, build bot, or BitBucket depending on

context, and "WAL" (Write-Ahead Log), a specialized database term that may be unclear to developers unfamiliar with specific architectural patterns.

Table 3.2: Examples of Task Description Enhancement

Original Subtask Description	**Enhanced Subtask Description**
BB - cordova-VERSION.js → cordova.js	BB - Rename cordova-VERSION.js to cordova.js in the Blackberry platform
Add a ./cordova/run project-level script to WP7	Implement ./cordova/run for Windows Phone 7
Fix NPE that is showing up since HBASE-14274 went in	Fix NullPointerException in TestDistributedLogSplitting
WAL split needs to be abstracted from ZK	Abstract Write-Ahead Log (WAL) splitting from ZooKeeper

3.3 Taxonomy of Task Decomposition Patterns Observed in Our Dataset

To understand how software developers structure complex work, we created the dataset of 2,151 high-level tasks (parent issues) and their 16,068 corresponding subtasks. Based on a structural and semantic analysis of these task decompositions. We first did literature review on popular task decomposition strategies in SE domain. Here are the 7 common decomposition patterns.

- **Data-driven Decomposition:** The task is broken down based on the data structures and data flows within the system. This approach organizes software components around the natural flow and processing of information, ensuring that program structure corresponds to the structure of the data it processes [18]. Subtasks under this pattern are structured around data transformation requirements, where each component handles specific data processing operations such as input validation, transformation, or storage.
- **Functional Decomposition:** The task is divided into a hierarchy of mathematical functions or algorithmic procedures. This approach originated from structured programming principles established by Dijkstra [15] and the modular design methodologies of Yourdon and Constantine [41], emphasizing top-down decomposition of program logic. Subtasks typically correspond to implementing specific algorithms, mathematical transformations, or procedural workflows (e.g., parsing input, performing calculations, formatting output) that can be independently developed and tested.
- **Object-Oriented Decomposition:** The task is divided based on real-world entities or conceptual objects from the problem domain. Each class represents a distinct entity, encapsulating its data and behaviors. This approach follows object-oriented principles such as encapsulation, inheritance, and polymorphism, emphasizing domain models over functional

procedures [10]. Subtasks include identifying domain classes (e.g., User, Account, Transaction), defining their attributes and methods, and establishing relationships through inheritance or composition.

- **Feature-based Decomposition:** The task is broken down according to user-visible functionality. This strategy aligns with the principles of Feature-Oriented Software Development (FOSD), which treats features as the primary dimension for structuring programs [3]. In practice, this is often operationalized through user stories, which serve as concise descriptions of a desired outcome for a specific user [13, 30]. Subtasks in this pattern correspond to implementing or modifying these features.
- **Component-based Decomposition:** The task is partitioned based on the system's architectural components or modules that must be created or modified. This approach prioritizes the logical structure of the system, aiming to improve comprehensibility and flexibility by encapsulating change within clear boundaries, a principle articulated by Parnas [29]. Subtasks under this pattern typically involve work on distinct classes, services, or modules.
- **Step-based Sequential Decomposition:** The task is divided into a chronological sequence of implementation steps. This represents a procedural breakdown of the work, where subtasks are ordered to reflect dependencies in the development process (e.g., set up database schema, followed by implementing data access layer). This linear approach can be a fundamental part of larger, iterative development frameworks, such as the Spiral Model [9], where each iteration may contain sequentially ordered steps.
- **Test-driven Decomposition:** The task is structured around the creation and validation of test cases. This strategy is the key of Test-Driven Development (TDD), where the development process is guided by a tight loop of writing a failing test, writing the minimal code to pass it, and then refactoring [7]. When this pattern is observed, subtasks are explicitly defined for writing unit tests, integration tests, or fixing failing test cases, often preceding the implementation of the feature itself.

Pattern Observation: Through systematic analysis of our dataset, we identified 3 out of 7 theoretical decomposition patterns after manually examining 15% of the collected data. The observed patterns include feature-based decomposition, component-based decomposition, and step-based decomposition. The remaining patterns, functional and object-oriented decomposition, exhibited significantly smaller granularity than what our dataset encompasses, as our focus was primarily on larger-scale decomposition strategies. Data-driven decomposition was also absent, which can be attributed to the fact that most projects in our sample did not involve data processing pipeline development. The majority of task decompositions involved either creating new features or refactoring existing code. Test-driven decomposition, while theoretically relevant, was not observed in our dataset. Although we identified decomposition tasks focused on creating different types of test cases, we were unable to confirm whether these test cases served as the primary initiative for subsequent code development.

The following examples illustrate the three observed decomposition categories. Feature-based decomposition breaks down functional feature tasks, such as implementing a search result view, where each subtask contributes a distinct capability (filtering, pagination, search controls, etc.) to the same UI component. Component-based decomposition involves identifying which system

components require similar modifications, as demonstrated in our documentation editing example. Step-based sequential decomposition represents tasks that must be completed in a predetermined order to achieve the overall goal, where each step depends on the completion of previous steps and cannot be parallelized.

Decomposition Type	**Parent Task Example**	**Representative Subtasks**
Feature-based	Implement the search results view	- Configure default filters to display results for all platforms - Display matching plugins in a paginated table format - Integrate a search control on the search results page - Implement filtering options for specific platforms
Component-based	Remove ios usage descriptions from plugins and document alternative	- Remove iOS usage descriptions from camera plugin - Remove iOS usage description from media plugin - Remove iOS usage descriptions from media capture plugin - Remove iOS usage descriptions from geolocation plugin - Remove iOS usage description from cordova-plugin-contacts
Step-based Sequential	Implement AWS Snowball Support in Hadoop	- Disable chunked encoding on the S3 client in Hadoop - Test data transfer using `fs -cp` and DistCp - Document the support for AWS Snowball including configuration steps

Table 3.3: Examples of Software Task Decomposition Types

Decomposition Type Classification Given the three observed patterns in our dataset, we used GPT-4.1 to systematically classify decomposition types across all instances. To validate the accuracy of our classification approach, we conducted an inter-rater reliability (IRR) assessment using 100 randomly selected examples from the dataset. For each example, both the LLM and human annotators were provided with the parent task title and description, associated subtasks, and a reference sheet defining the three decomposition types and highlighting key distinguishing features.

IRR Results The evaluation involved two human raters: an active open-source software engineer and the primary researcher. The classification accuracy against the human majority consensus

reached 79.21%, which demonstrates promising performance considering the stringent evaluation criteria requiring unanimous agreement between both human raters and alignment with the LLM-generated labels. The Cohen's Kappa coefficients between the LLM and the human raters were as follows:

- LLM vs Human 1: $\kappa = 0.630$
- LLM vs Human 2: $\kappa = 0.578$

Cohen's Kappa is computed using the formula:

$$\kappa = \frac{P_0 - P_e}{1 - P_e}$$

where P_0 is the observed agreement proportion, and P_e is the expected agreement by chance. The denominator $(1 - P_e)$ accounts for the maximum possible improvement beyond chance agreement.

The moderate Kappa values can be attributed to two primary factors. First, some decomposition instances contained vague or insufficient descriptions, making it difficult to confidently distinguish between feature-based and component-based categories. Second, the limited number of classification categories increases the chance agreement P_e, which reduces the denominator $(1 - P_e)$ in the Kappa formula. This makes Cohen's Kappa more sensitive to disagreements, and can result in lower κ values even when observed agreement is relatively high.

Despite these limitations, the IRR results indicate substantial agreement and support the reliability of our classification methodology.

Category distribution The category distribution (Figure 3.2) reflects the development practices commonly observed in Apache projects. Feature-based decomposition accounts for 1,088 instances (48.6%), representing the most prevalent pattern in our dataset. Component-based decomposition comprises 952 instances (42.5%), while step-based sequential decomposition represents 111 instances (5.0%). This distribution shows the modular and feature-driven development approaches characteristic of Apache projects, where developers typically organize work around discrete functional capabilities rather than sequential procedural steps.

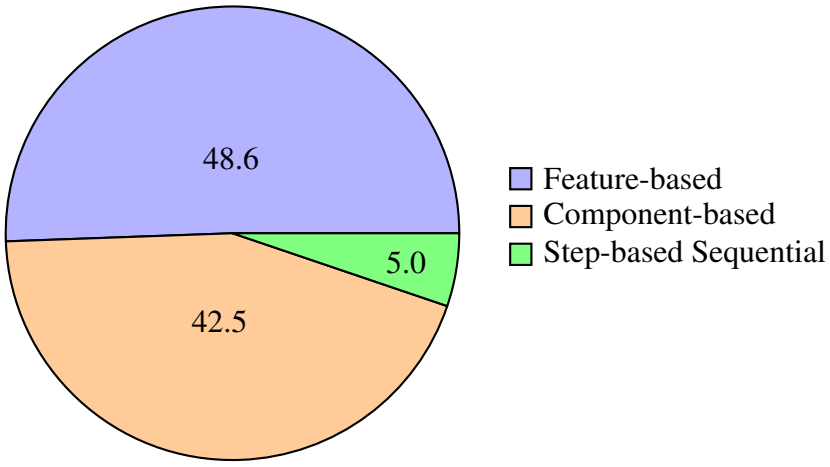

Figure 3.2: Distribution of Decomposition Patterns

3.4 Dataset Scope and Limitation

This dataset fills a gap in software engineering research by providing labeled decomposition types for real-world development tasks. It supports several research applications, including fine-tuning large language models to learn task decomposition strategies. Each task includes detailed descriptions, pull requests, discussion threads, and timestamps, ensuring full traceability and enabling longitudinal analysis of development processes. The rich metadata also supports future research, such as evaluating the effectiveness of decomposition by comparing completion times between standard and decomposed task executions.

However, several limitations must be acknowledged. First, the dataset exhibits significant language and technology stack bias. Apache Foundation projects are heavily backend-focused, resulting in a dominance of Java, while underrepresenting mobile stacks (e.g., Swift, Kotlin) and web front-end frameworks (e.g., React, Vue). This technological bias may limit generalizability to projects with different user interface requirements and development paradigms. Second, the exclusive focus on Apache Foundation projects may reduce applicability to organizations with different development cultures, project scales, or governance structures. The Foundation's emphasis on open-source collaboration and modular architecture may not reflect decomposition practices in proprietary or smaller-scale projects.

Future work should expand the dataset to include a broader range of programming languages, development contexts, and organizational settings to enhance the generalizability of insights into task decomposition in software engineering.

Chapter 4

A Framework for Automated Task Decomposition

This chapter constructs an automated task decomposition framework using the exploratory dataset from the previous chapter. The framework validates the effectiveness of hierarchical task breakdown in software engineering agents. We implement and evaluate a decomposition system that enhances coding agents' problem-solving capabilities. The dataset serves dual purposes: informing the decomposition component design and providing training data for fine-tuning a specialized language model. We benchmark this fine-tuned model against established baselines including GPT-4.1-mini to comprehensively evaluate decomposition-enhanced automation.

4.1 Hierarchical Task Management Framework

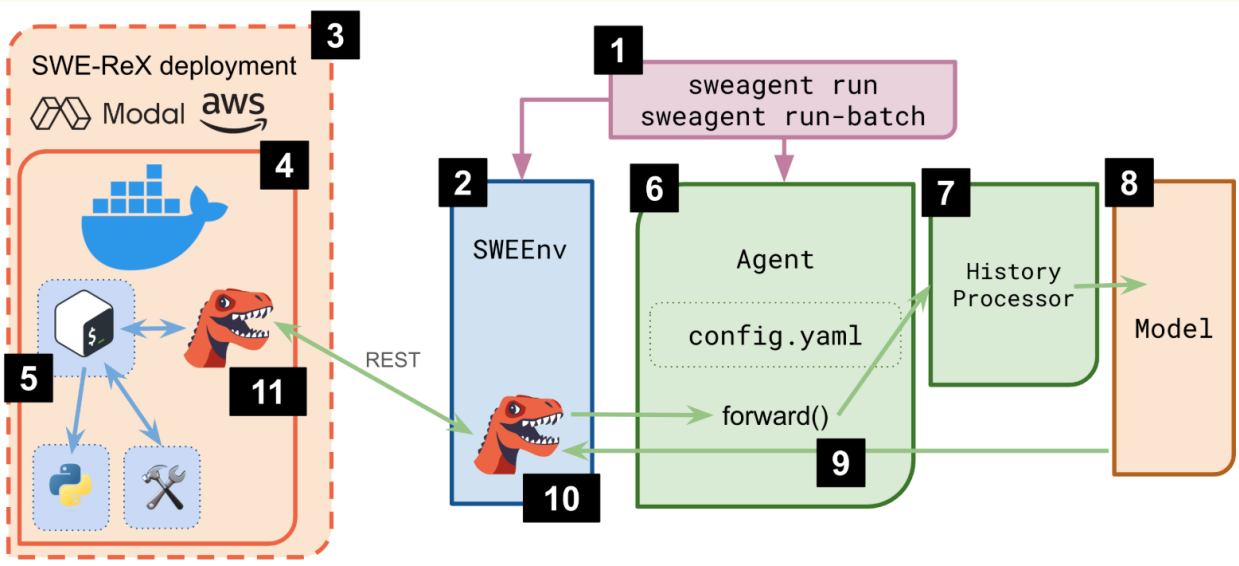

Figure 4.1: SWE-Agent Architecture [40]

This section presents the design and implementation of a hierarchical task decomposition

framework for enhancing automated software engineering agents. The proposed system extends the SWE-Agent [40] architecture by integrating a new task decomposition tool with persistent memory management capabilities.

4.1.1 System Architecture and SWE-Agent Integration

The task decomposition system is implemented as an integrated component within the SWE-Agent. To enable effective task management, a specialized tool was developed and integrated into the agent's toolkit, establishing persistent memory mechanisms for tracking decomposition state across agent execution cycles.

The SWE-Agent framework (Figure 4.1) provides a modular architecture for autonomous software engineering through coordinated interaction between the Agent component, execution environment (SWEEnv), and model interface. Our task decomposition system extends this architecture through two key contributions: (1) a decomposition tool integrated within the existing tools/ directory structure, and (2) a containerized side model that serves as a guardrail for task completeness verification, integrated directly into the Agent's forward() method with access to the current execution context.

The decomposition tool integrates at the agent level, enabling seamless invocation through the standard command interface while accessing both the execution environment and history management capabilities. The containerized side model operates as a verification guardrail within the forward() execution cycle, leveraging the existing context access mechanisms to assess task completion status. This integration approach maintains full compatibility with existing SWE-Agent workflows while extending the framework's capabilities for hierarchical task management and automated progress validation.

4.1.2 Hierarchical Task Structures

The hierarchical task decomposition employs a tree-based data structure where each node represents either a primary software engineering issue or a derived subtask (Figure 4.2). The tree structure maintains parent-child relationships through unique task identifiers, enabling bidirectional navigation and dependency tracking throughout the decomposition hierarchy.

Each task node contains essential metadata including task description, current status, task creation and update time. The root node corresponds to the original issue, while leaf nodes represent atomic, directly executable subtasks. Intermediate nodes serve as logical groupings that decompose complex requirements into manageable components.

The system implements a depth-limited hierarchy to prevent excessive decomposition, with a maximum depth limit of 3 levels as illustrated in Figure 4.2. Task relationships are encoded using a dictionary representation, facilitating efficient traversal and modification operations. The hierarchical task structure is implemented as a class-based system, where every update triggers an automatic JSON file update within the Docker environment for persistence and state management.

4.1.3 Task State Management

The system implements a persistent state file mechanism that maintains continuity across agent execution cycles, enabling consistent tracking of task decomposition progress. The task state transi-

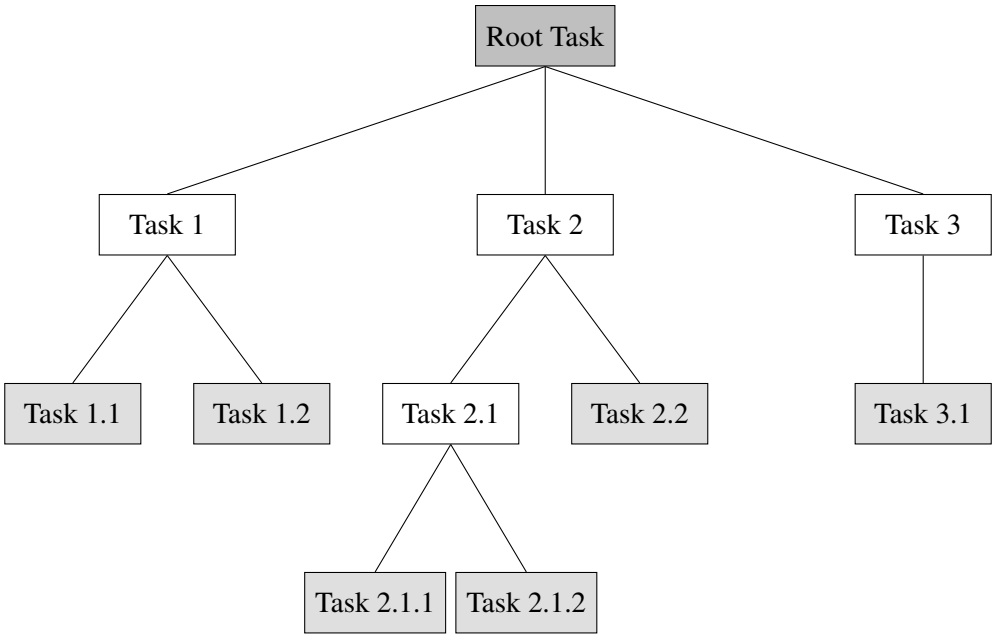

Figure 4.2: Hierarchical Task Decomposition with 3-Level Depth Limit

tion graph is shown in Figure 4.3. Each task begins in the PENDING state and transitions through a finite set of defined states: IN_PROGRESS, COMPLETED, and FAILED. The state machine governs valid transitions such as PENDING $\rightarrow$ IN_PROGRESS, PENDING $\rightarrow$ COMPLETED, or PENDING $\rightarrow$ FAILED. Once a task is marked IN_PROGRESS, it can either complete successfully (COMPLETED) or fail (FAILED). This structured state management allows the agent to reason about task status, identify incomplete or failed subtasks, and resume progress accurately in subsequent runs.

State Verification Prior to each action step within the SWE-Agent execution loop, the system prompts the language model to assess the completion status of the current task using the prompt: CHECK WHETHER THE CURRENT FOCUS TASK IS COMPLETED. IF COMPLETED, PLEASE CALL MANAGE_TASK_STATE TOOL TO UPDATE THE STATUS OF THE TASK. This verification process involves analyzing the execution trajectory and determining whether the active task objectives have been satisfied. Upon verification of task completion, the system automatically invokes the task state management tool to update the current task status and advance to the subsequent task in the execution queue. When all decomposed tasks reach completion, the system generates the final patch output and terminates the software engineering workflow.

4.2 Implementation of Decomposition Strategies

4.2.1 Strategy 1: Mandatory Single-level Planning

This approach enforces systematic task decomposition by requiring the agent to create a complete plan before beginning any work. The strategy is implemented through explicit instructions in

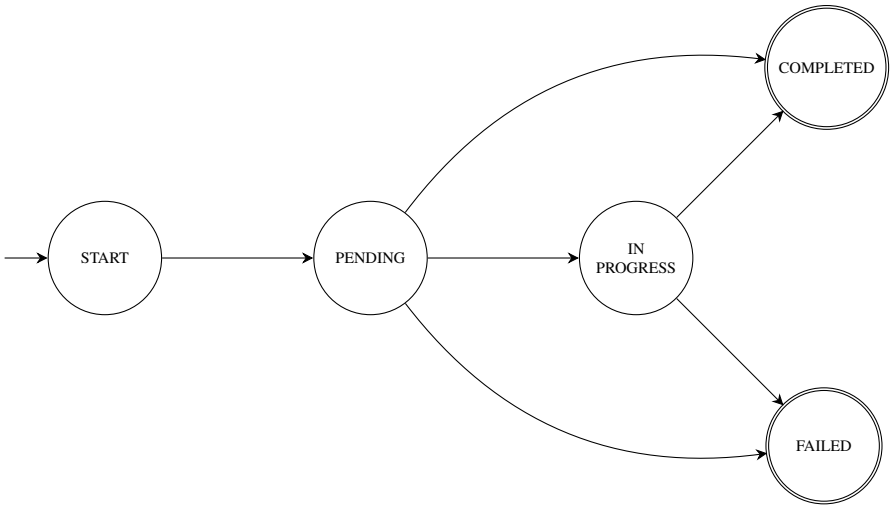

Figure 4.3: Task State Machine for the Decomposition Agent

the system prompt. These instructions mandate that tasks must be broken down before execution starts. Planning is constrained to a single hierarchical level. All subtasks are identified upfront and executed sequentially according to the predetermined structure. This approach defines task completeness through systematic enumeration of all necessary subtasks. The enumeration ensures comprehensive coverage of the problem space before implementation begins.

4.2.2 Strategy 2: Dynamic Multi-level Planning

This strategy enables dynamic decomposition decisions during task execution. The agent can determine whether decomposition is necessary based on the complexity it encounters. The approach utilizes unit granularity assessment. In this assessment, the agent evaluates whether further decomposition would provide meaningful benefit to task completion. Decomposition triggers are activated when specific conditions are met. These triggers activate when the agent determines that a task has not reached its fundamental unit level. They also activate when the agent determines that subdivision would enhance execution effectiveness. The strategy supports recursive decomposition with a maximum depth of three hierarchical levels. This enables multi-level task hierarchies that evolve based on intermediate findings and trigger conditions.

Both strategies were evaluated on SWE-Bench Verified instances using issue resolve rate as the primary metric. The comparative analysis focuses on the effectiveness of predetermined versus adaptive decomposition approaches. This analysis examines how each approach handles diverse software engineering challenges.

4.2.3 Tool Interface and Integration

The task decomposition tool is designed with a structured interface that operationalizes the empirical insights derived from our dataset analysis. Based on our previous findings, the tool supports both feature-based and component-based decomposition strategies, which emerged as the most prevalent decomposition patterns in real-world software engineering practices.

The decomposition process employs automated subtask enumeration. The language model analyzes task descriptions and complexity to determine the optimal number of subtasks. The system enforces a maximum limit of 12 subtasks per decomposition level to prevent over-fragmentation. This threshold corresponds to the 85th percentile observed in our Apache dataset analysis, providing an empirically grounded constraint that would suit most real-world scenarios while maintaining execution efficiency.

The tool interface follows a standardized command structure designed for seamless integration with the SWE-Agent framework:

```
decompose_task <task_description> <num_subtasks>
<decomposition_type> [<parent_id>] [<state_file>]
```

The function parameters are detailed in Table 4.1, providing flexibility for both single-level and hierarchical decomposition scenarios. The integration maintains persistent state management through JSON serialization, ensuring task hierarchies and execution context are preserved across agent execution cycles.

Parameter	**Type**	**Required**	**Description**
`task_description`	string	Yes	Description of the complex task that needs to be decomposed
`num_subtasks`	string	Yes	Number of subtasks to create (must be less than 12)
`decomposition_type`	string	Yes	Type of decomposition (Feature-based, Component-based)
`parent_id`	string	No	Parent task ID to decompose under an existing task
`state_file`	string	No	Path to the state file location

Table 4.1: Parameters for `decompose_task` Command

4.3 Experiment Evaluation Setup

4.3.1 Benchmark Selection

The experimental evaluation followed a two-phase design to ensure both rapid prototyping and rigorous benchmarking of the proposed task decomposition framework. In Phase 1, we conducted initial validation using the SWE-Bench Lite development dataset [21], which consists of 23 instances. This subset was intentionally chosen for its manageable size, making it well-suited for iterative debugging and refinement of the methodology. All instances in this set pertain to Python-based open-source projects, covering a range of bug types with varying levels of complexity. Some tasks involve straightforward fixes, while others require deeper reasoning or broader codebase understanding. This diversity allowed us to observe how the decomposition strategy performs under different levels of task difficulty and ambiguity. Additionally, the lightweight nature of the dataset enabled rapid evaluation without incurring substantial computational cost.

Phase 2 employed the SWE-Bench Verified dataset for more rigorous assessment [12]. This benchmark contains 500 human-verified instances with validated issue descriptions, solution patches, and ground truth. Unlike other SWE-Bench variants that often contain ambiguous or unsolvable problems, the Verified dataset removes infeasible cases to eliminate noise. This filtering ensures that performance metrics reflect genuine problem-solving ability rather than dataset artifacts.

To contextualize our dataset choices, table 4.2 summarizes the characteristics of the major SWE-Bench variants used in our evaluation, including instance counts, validation status, and typical use cases.

Dataset Variant	Instances	Validated	Use Case
SWE-Bench Lite Dev	23	×	Fast iteration, debugging, ablation studies
SWE-Bench Lite Test	300	×	Lightweight benchmarking
SWE-Bench Full Train	19,000	×	Large-scale pretraining and fine-tuning
SWE-Bench Full Dev	225	×	Baseline model development
SWE-Bench Full Test	2,290	×	Broad-scale evaluation with noise
SWE-Bench Verified	500	✓	Reliable benchmarking with human validation

Table 4.2: Comparison of SWE-Bench Dataset Variants

4.3.2 Model Configuration and setups

Experimental Model Variants The evaluation employed three model configurations to systematically assess the impact of task decomposition on software engineering performance:

- **Baseline Models**: GPT-4.1 [26] and GPT-4.1-Mini [27] without decomposition.
- **Enhanced Variants**: GPT-4.1 and GPT-4.1-Mini with Strategy 1 (mandatory single-level planning) or Strategy 2 (dynamic multi-level planning).
- **Fine-tuned Model**: Qwen3 1.7B [32] trained on our decomposition dataset using Strategy 1 with GPT-4.1-mini as agent model.

Fine-tuned Model Configuration The Qwen3 1.7B model was fine-tuned specifically for task decomposition using the parameters detailed in Table 4.3.

Experiment Control All experiments followed standardized protocols to ensure reproducible comparisons. Baseline agent executions used deterministic sampling (temperature = 0.0) to eliminate stochastic variation. Decomposition components operated with temperature = 0.7 to enable diverse reasoning during task breakdown. A budget of $0.50 was allocated per problem instance, with token limits adjusted accordingly based on model pricing. Each run executed in isolated Docker environments using the standard SWE-Bench evaluation harness with automated scoring.

Parameter	Value
Base Model	Qwen3 1.7B
Training Format	Alpaca
Cutoff Length	2048 tokens
Learning Rate	0.0003
Training Epochs	3
LR Scheduler	Cosine
Optimizer	AdamW
LoRA Rank	8
LoRA Alpha	16

Table 4.3: Fine-tuning configuration parameters for Qwen3 1.7B model

4.4 Results and Performance Analysis

4.4.1 Initial Validation on SWE-Bench Lite

In our initial validation on the SWE-Bench Lite development split, we tested three model configurations to establish baseline performance without task decomposition. The full-sized GPT-4.1 model resolved 7 out of 23 issues (30.4%), while the smaller GPT-4.1 Mini variant resolved 6 out of 23 issues (26.1%). When we applied our first decomposition strategy (S1) to GPT-4.1 Mini, the resolution count remained the same at 7 out of 23 (30.4%). Although the decomposition-enhanced model resolved one additional issue compared to the non-decomposed GPT-4.1 Mini, the improvement is negligible given the small sample size. These results suggest that decomposition does not provide a measurable improvement on this subset (Table 4.4).

Manual inspection indicates that many Lite issues are either too vague or inherently difficult. This leads to high variance in model performance and may obscure any potential gains from decomposition. Therefore, we chose to conduct all further evaluations using the SWE-Bench Verified dataset, which contains manually curated issues better suited for assessing decomposition effectiveness.

Model Configuration	Resolved / Total	Success Rate
GPT-4.1 (no decomposition)	7 / 23	30.4%
GPT-4.1 Mini (no decomposition)	6 / 23	26.1%
GPT-4.1 Mini + decomposition S1	7 / 23	30.4%

Table 4.4: Performance on the SWE-Bench Lite

4.4.2 Full Evaluation on SWE-Bench Verified

The evaluation on the SWE-Bench Verified split shows that task decomposition improves the resolution capability of the compact GPT-4.1 Mini model. Without decomposition, GPT-4.1 Mini

resolves 152 out of 500 issues (30.4%). Strategy 1 increases this to 189 (37.8%), a 24.3% relative improvement. Strategy 2 leads to 178 resolutions (35.6%), a 17.1% gain.

Notably, Strategy 1 enables GPT-4.1 Mini to slightly outperform the full-sized GPT-4.1 model without decomposition, which resolves 182 issues (36.4%). This suggests that decomposition can partially compensate for reduced model capacity. Strategy 2 also narrows most of the performance gap, although it does not surpass GPT-4.1.

Impact of Cost Limitations on Decomposition Performance However, our $0.50 budget constraint did affect the evaluation result, particularly for decomposition-enhanced models. As shown in Table 4.5, models using decomposition strategies hit the budget limit much more frequently than their non-decomposed counterparts. GPT-4.1 Mini with Strategy 1 reached the budget constraint in 43 instances compared to only 8 instances for the baseline model. Similarly, Strategy 2 hit the budget limit in 38 cases.

This pattern indicates that decomposition strategies require additional computational resources to execute their structured approach effectively. The high number of budget-constrained instances (38-46 for decomposition vs. 8-23 for baselines) suggests that our evaluation may have underestimated the true potential of decomposition approaches. Many instances that were marked as "failed" may have been successfully resolved given sufficient computational budget to complete the decomposition process.

The GPT-4.1 model shows a moderate number of budget hits (23 instances) despite not using decomposition, which reflects its higher per-token cost compared to the Mini variant. This cost differential explains why the full model hits budget constraints more frequently than the Mini baseline despite similar token consumption patterns.

Model Architecture and Decomposition Effectiveness The results for Qwen3 models provide additional insights into the effectiveness of task decomposition across different model architectures. Both Qwen3 without finetuning (Qwen3-no-ft) and with finetuning (Qwen3-ft) plus Strategy 1 achieve competitive performance (37.0% and 37.6% respectively), also demonstrating the benefits of decomposition. Qwen3-ft with Strategy 1 slightly outperforms the full-sized GPT-4.1 baseline (37.6% vs 36.4%).

Interestingly, both Qwen3 variants also show high budget constraint rates (44-46 instances), suggesting that the computational overhead of decomposition is consistent across different model architectures. However, while Qwen3-ft shows improvement over the GPT-4.1 baseline, the gains are relatively modest and do not substantially exceed the performance achieved by GPT-4.1 Mini with decomposition.

Limitations of Transfer Learning Two main factors could limit the performance of Qwen3-ft. First, there is granularity mismatch between the fine-tuning dataset and SWE-Bench. The Apache project issues used for fine-tuning typically involve higher-level system decomposition across different platforms and architectural components, whereas SWE-Bench issues are focused on localized bug identification and resolution within specific codebases. This fundamental difference in problem scope means the decomposition patterns learned during fine-tuning may not directly transfer to the more targeted debugging tasks in SWE-Bench.

Second, there exists a domain shift between the training and evaluation data. The fine-tuning dataset is heavily skewed toward Apache projects, which are primarily Java-based and emphasize enterprise-scale software architecture, while all SWE-Bench projects are Python-based with different coding patterns, library ecosystems, and debugging methodologies. This language and ecosystem mismatch may limit the model's ability to effectively apply learned decomposition strategies to the Python-centric evaluation environment.

Result Summary As shown in Figure 4.4, both decomposition strategies raise the performance of the compact model beyond its baseline. These results support the conclusion that task decomposition improves real-world issue resolution, even for smaller models. However, the high frequency of budget-constrained executions suggests that the observed improvements represent a conservative estimate of decomposition effectiveness. Future evaluations should consider both cost-constrained and unconstrained settings to fully characterize the performance potential of decomposition-enhanced approaches.

Model Configuration	**Resolved / Total**	**Exit_cost**	**Success Rate**
GPT-4.1 (no decomposition)	182 / 500	23	36.4%
GPT-4.1 Mini (no decomposition)	152 / 500	8	30.4%
GPT-4.1 Mini + Strategy 1	189 / 500	43	37.8%
GPT-4.1 Mini + Strategy 2	178 / 500	38	35.6%
Qwen3-no-ft + Strategy 1	185 / 500	44	37.0%
Qwen3-ft + Strategy 1	188 / 500	46	37.6%

Table 4.5: Performance on SWE-Bench Verified

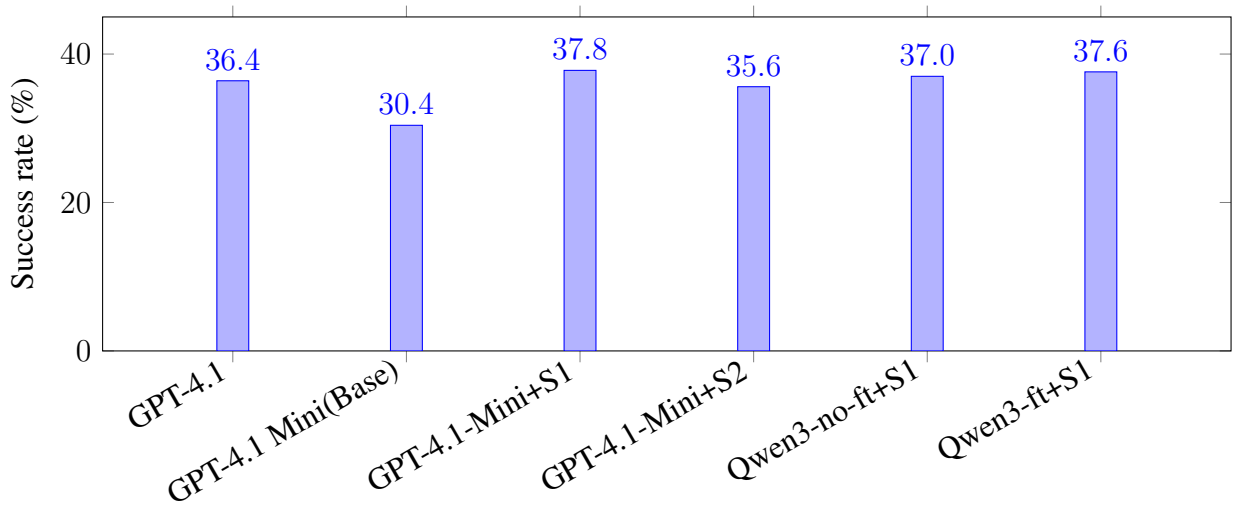

Figure 4.4: Success-rate on SWE-Bench Verified

4.4.3 Qualitative Analysis of Agent Behavior

To complement the quantitative results in Table 4.5 and Fig. 4.4, we manually analyzed repair traces produced by three configurations: the no-decomposition baseline, Strategy 1 (static plan), and Strategy 2 (dynamic plan). Strategy 1 successfully resolved 38 issues that the baseline failed to address.

A common failure pattern involves bugs that span multiple control-flow stages, such as parsing, validation, execution, and output. In these cases, missing logic in one phase can silently propagate errors downstream. For example, in Django's `call_command`, mutually exclusive options passed as keyword arguments bypass `argparse`, causing the validation phase to be skipped entirely. Without decomposition, the model often overlooks this gap. Strategy 1 addresses the problem by explicitly breaking the task into distinct stages: input normalization, argument validation, command execution, and output verification. This structure helps the agent surface hidden assumptions and insert missing checks at the correct point in the pipeline.

Another failure mode arises from tightly coupled or poorly abstracted modules. When system responsibilities are not clearly separated, models tend to apply fixes in the wrong place. In Django's `makemigrations`, the planner calls `allow_migrate()` on every model instead of scoping the call to the target app. This causes failures in sharded databases where some shards do not contain the full set of models. Without clear module boundaries, the model may attempt to fix the router logic directly, leading to incorrect or brittle patches. Strategy 1 helps avoid this by structuring the repair into steps that clarify component responsibilities. The model focuses the change on the planner's call site, leaving unrelated modules untouched.

Compared to Strategy 1, Strategy 2 performs less reliably on these patterns. Although it dynamically adjusts to new observations, it often struggles to identify the right granularity of decomposition. In several cases, it attempted to reproduce, diagnose, patch, and test a bug in a single step, resulting in superficial or irrelevant fixes, such as inserting a type cast unrelated to the root cause. This highlights the value of explicit sub-task planning: Strategy 1 encourages the model to isolate fault localization, hypothesis generation, and fix implementation, reducing the chance of overgeneralized edits.

However, for the remaining unresolved issues in SWE-Bench, our analysis suggests that decomposition limitations were rarely the primary bottleneck. While we lack systematic quantification of these failure modes, our impression from examining unsuccessful repair attempts indicates that most failures stem from fundamental limitations in the model's ability to comprehend complex problem contexts or execute the necessary reasoning steps, rather than inadequate task decomposition per se. This suggests that while improved decomposition strategies can meaningfully expand the range of solvable problems, they cannot address the core challenges posed by the most difficult issues in the benchmark.

Chapter 5

Discussion

Our experiments demonstrate a 24% relative improvement in issue resolution on SWE-Bench through structured task decomposition, even under computational constraints (GPT-4.1-mini, limited tokens). This finding confirms that decomposition enhances problem-solving structure and establishes a foundation for building effective human-augmentation tools in software engineering. The observed benefits are particularly significant given the limitations of the models involved, suggesting that even greater value could be realized when such methods are applied to human contributors.

While large language models benefit from structured task breakdown despite having access to extensive compute and comprehensive code knowledge, human contributors — especially those with limited expertise — stand to gain even more. Human cognition is constrained by working memory and domain familiarity, and decomposition offers a systematic way to manage these limitations. By dividing complex tasks into smaller, coherent subtasks, decomposition enables human contributors to engage with software engineering problems in ways that align more closely with their cognitive capabilities.

This is particularly beneficial for newcomers, who often struggle with the complexity of large-scale systems. Unlike LLM that can process an entire codebase in a single pass, humans acquire knowledge incrementally. Structured decomposition facilitates this process by creating a scaffolded learning path, allowing individuals to begin with simpler subtasks and gradually build the knowledge and confidence required to tackle more complex components. This approach offers a more principled alternative to ad hoc onboarding practices such as labeling certain issues as "good first tasks," which often fail to convey how subtasks fit into the larger system architecture.

Beyond individual learning, structured decomposition also enhances collaboration. By clearly delineating subtask boundaries, it enables experienced maintainers to focus on high-level architectural decisions while newcomers contribute to well-scoped units of work. This division of labor allows for efficient utilization of expertise across a development team and fosters inclusive participation by reducing the barrier to entry.

To better understand the limitations of our approach, we conducted a focused analysis of 40 randomly selected failure cases drawn from the 311 issues where GPT-4.1 Mini with Strategy 1 was unsuccessful. These failures were primarily attributable to the limitations of the underlying language model rather than flaws in the decomposition strategy itself. In most cases, the model struggled to reason about tasks involving high-complexity modules or specialized frameworks outside its training distribution. Moreover, decomposition was less effective when domain-specific

knowledge was required to generate accurate subtasks. Nonetheless, even in these challenging scenarios, decomposition still improved resolution rates compared to non-decomposed baselines, reinforcing its general utility.

While these cases expose the boundaries of decomposition utility, they also highlight important differences between decompositions optimized for AI agents and those that best support human contributors. AI models benefit from fine-grained decomposition tailored for execution efficiency, whereas human contributors may prefer coarser-grained subtasks that preserve autonomy and facilitate conceptual understanding. Additionally, human-centered decomposition must account for team dynamics, communication costs, and the evolving expertise of contributors. These observations suggest that effective human-augmentation tools should support adaptive granularity, allowing users to tune the level of detail to match their preferences and skill levels.

Several challenges must be addressed to translate these insights into practical systems. One such challenge is the balance between static and dynamic decomposition. In our experiments, static strategies outperformed dynamic ones, but this may not hold in human-centered settings where contributors are continuously learning. Decomposition strategies that adapt to an individual's growth trajectory,creating increased complexity as competence improves, could provide a more effective learning experience.

Evaluating such systems also presents methodological difficulties. Task completion alone is an inadequate measure of effectiveness when human learning is a core objective. Future work must explore additional metrics such as contributor satisfaction, retention, and learning outcomes, likely requiring longitudinal user studies. Moreover, any deployment of these systems must consider real-world integration constraints. For decomposition tools to be adopted in practice, they must integrate seamlessly with existing development workflows, including project management platforms, code review pipelines, and communication tools, without disrupting established team norms.

Another limitation stems from the generalizability of our findings. Our decomposition model was trained on Java-based Apache projects, while evaluation was conducted on Python-based SWE-Bench issues. This mismatch highlights the need to test decomposition strategies across diverse programming languages and project types. It remains an open question whether patterns observed in modular, backend-heavy systems such as Apache generalize to mobile, front-end, or smaller-scale development contexts.

Despite these limitations, our results inspire a range of practical tool designs. Adaptive learning environments could leverage automated decomposition to construct personalized learning trajectories for newcomers, dynamically adjusting complexity based on contributor progress. Similarly, collaborative interfaces could allow experienced developers to curate and revise decompositions before distributing them to the wider team, supporting onboarding and reducing the burden on individual mentors. Integrating decomposition into development environments—coupled with navigation tools, documentation, and version control—could offer context-aware support that helps contributors situate their work within the broader codebase.

Our study contributes several foundational insights for the design of such tools. Empirical validation shows that decomposition improves task performance, providing evidence for its practical value. The identification of three decomposition patterns — feature-based, component-based, and step-based — offers a taxonomy that can guide strategy selection based on task characteristics. Our analysis of computational tradeoffs clarifies how to balance tool effectiveness with resource constraints, while the failure case study helps delineate the current boundaries of decomposition efficacy.

In light of these findings, new research directions emerge. A key challenge is granularity calibration: determining the optimal decomposition detail for contributors with different experience levels and goals. This requires metrics that go beyond efficiency to assess user engagement and learning outcomes. Additionally, understanding how decomposition strategies transfer across domains — especially between projects with different architectural styles, contributor bases, and tooling ecosystems — is important for generalizability.

Finally, the social dynamics of tool adoption must not be overlooked. Automated decomposition tools interact with mentorship structures, communication norms, and team collaboration practices. Their long-term impact on project health, innovation, and contributor retention must be assessed through longitudinal studies that track contributors over extended periods.

Addressing these challenges will require interdisciplinary collaboration across software engineering, human-computer interaction, and learning sciences. Only by combining insights from these domains can we design human-augmentation tools that not only improve task efficiency but also foster learning, collaboration, and sustainable growth in open-source software ecosystems.

Chapter 6

Conclusion

This thesis establishes task decomposition as a promising foundation for human-augmentation in software engineering. The 24% improvement in AI agent performance provides strong evidence that structured task breakdown creates genuine value, while our empirical analysis of real-world decomposition patterns from Apache projects offers practical guidance for tool development.

Rather than simply matching newcomers with easy tasks, this work focuses on constructing accessible routes that lead to significant and meaningful work. Rather than limiting newcomers to peripheral contributions, decomposition enables transformation of complex issues into structured learning experiences while maintaining connection to important project goals.

The path forward is clear: the technical foundation and empirical insights presented here enable the next critical step of validating decomposition effectiveness through human studies with real newcomers in live open source projects. The tools, datasets, and analytical frameworks developed in this research provide the necessary basis for building and evaluating human-augmentation systems that can meaningfully improve newcomer onboarding and long-term project sustainability.

The ultimate goal — AI systems that provide structured problem breakdowns enabling humans to engage in implementation, learning, and creative reasoning — represents a productive division of cognitive labor that leverages the complementary strengths of human and artificial intelligence. This work marks a significant step toward that goal.

Bibliography

[1] *Applying WBS Attributes and Concepts*, chapter 2, pages 19–40. John Wiley Sons, Ltd, 2008. ISBN 9780470432723. doi: https://doi.org/10.1002/9780470432723.ch2. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/9780470432723.ch2. 2.1

[2] Ra'Fat Al-Msie'deen. Requirements traceability: Recovering and visualizing traceability links between requirements and source code of object-oriented software systems, 2023. URL https://arxiv.org/abs/2307.05188. 2.1

[3] Sven Apel and Christian Kästner. An overview of feature-oriented software development. *Journal of Object Technology*, 8(4):1–36, July–August 2009. doi: 10.5381/JOT.2009.8.4.C1. URL https://www.cs.cmu.edu/~ckaestne/pdf/JOT09_OverviewFOSD.pdf. 3.3

[4] Daman Arora, Atharv Sonwane, Nalin Wadhwa, Abhav Mehrotra, Saiteja Utpala, Ramakrishna Bairi, Aditya Kanade, and Nagarajan Natarajan. Masai: Modular architecture for software-engineering ai agents, 2024. URL https://arxiv.org/abs/2406.11638. 2.2

[5] Astral. uv. https://github.com/astral-sh/uv, 2023. GitHub repository, accessed 2025-07-16. 3.1

[6] Atlassian. Jira software. https://www.atlassian.com/software/jira, 2025. Accessed: 2025-07-16. 3.1

[7] Kent Beck. *Test-Driven Development: By Example*. Addison-Wesley, Boston, MA, 2003. ISBN 978-0321146533. 3.3

[8] Dane Bertram, Amy Voida, Saul Greenberg, and Robert Walker. Communication, collaboration, and bugs: the social nature of issue tracking in small, collocated teams. In *Proceedings of the 2010 ACM Conference on Computer Supported Cooperative Work*, CSCW '10, page 291–300, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781605587950. doi: 10.1145/1718918.1718972. URL https://doi.org/10.1145/1718918.1718972. 2.1

[9] Barry W. Boehm. A spiral model of software development and enhancement. *ACM SIGSOFT Software Engineering Notes*, 11(4):14–24, 1988. doi: 10.1145/12944.12948. 3.3

[10] Grady Booch. *Object-Oriented Analysis and Design with Applications*. The Addison-Wesley Object Technology Series. Addison Wesley, 3rd edition, 2007. ISBN 9780321509376. 3.3

[11] Frederick P. Brooks. *The Mythical Man-Month: Essays on Softw*. Addison-Wesley Longman Publishing Co., Inc., USA, 1st edition, 1978. ISBN 0201006502. 2.1

[12] Neil Chowdhury, James Aung, Chan Jun Shern, Oliver Jaffe, Dane Sherburn, Giulio Starace, Evan Mays, Rachel Dias, Marwan Aljubeh, Mia Glaese, Carlos E. Jimenez, John Yang, Leyton Ho, Tejal Patwardhan, Kevin Liu, and Aleksander Madry. Introducing swe-bench verified. OpenAI Blog, Feb 2025. URL https://openai.com/index/introducing-swe-bench-verified/. Updated; human-validated subset of SWE-bench released. 4.3.1

[13] Mike Cohn. *User Stories Applied: For Agile Software Development*. Addison-Wesley, Boston, MA, 2004. ISBN 978-0321205681. 3.3

[14] OlivierL. deWeck. Requirements definition, decomposition, allocation, and validation. MIT OpenCourseWare – 16.842 Fundamentals of Systems Engineering (Fall2015), 2015. URL https://ocw.mit.edu/courses/16-842-fundamentals-of-systems-engineering-fall-2015/. "Requirements are decomposed in a hierarchical structure... these high-level requirements are decomposed into functional and performance requirements... further decomposed and allocated among the elements and subsystems". 2.1

[15] E.W. Dijkstra. *Notes on structured programming*, pages 1–82. APIC studies in data processing. Academic Press Inc., United States, 1972. ISBN 0-12-200550-3. 3.3

[16] Jerry FitzGerald and Ardra F. FitzGerald. *Fundamentals of systems analysis: using structured analysis and design techniques (3rd ed.)*. John Wiley & Sons, Inc., USA, 1987. ISBN 0471885975. 2.1

[17] Zoe Hoy and Mark Xu. Agile software requirements engineering challenges-solutions - a conceptual framework from systematic literature review. *Information*, 14(6), June 2023. ISSN 2078-2489. doi: 10.3390/info14060322. Wil be Gold OA - MDPI journal. 2.1

[18] M. A. Jackson. *Principles of Program Design*. Academic Press, Inc., USA, 1975. ISBN 0123790506. 3.3

[19] Shankar Kumar Jeyakumar, Alaa Alameer Ahmad, and Adrian Garret Gabriel. Advancing agentic systems: Dynamic task decomposition, tool integration and evaluation using novel metrics and dataset. In *NeurIPS 2024 Workshop on Open-World Agents*, 2024. URL https://openreview.net/forum?id=kRRLhPp7CO. 2.2

[20] Yanjie Jiang, Hui Liu, and Lu Zhang. Evaluating and improving chatgpt-based expansion of abbreviations, 2024. URL https://arxiv.org/abs/2410.23866. 3.2

[21] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=VTF8yNQM66. 2.2, 4.3.1

[22] An Ju, Hitesh Sajnani, Scot Kelly, and Kim Herzig. A Case Study of Onboarding in Software Teams: Tasks and Strategies, March 2021. URL http://arxiv.org/abs/2103.05055. arXiv:2103.05055 [cs]. 1

[23] Xinyi Li, Sai Wang, Siqi Zeng, Yu Wu, and Yi Yang. A survey on llm-based multi-agent systems: workflow, infrastructure, and challenges. *Vicinagearth*, 1(9), 2024. doi: 10.1007/s44336-024-00009-2. Open access. 2.2

[24] Fangru Lin, Emanuele La Malfa, Valentin Hofmann, Elle Michelle Yang, Anthony G. Cohn, and Janet B. Pierrehumbert. Graph-enhanced large language models in asynchronous plan reasoning. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org, 2024. 3.1

[25] Ravi Kiran Mallidi et al. User story decomposition in agile development: Crud splitting, business rule extraction, and workflow-based methods. In *Proceedings of the XXth International Conference on Agile Software Development*, pages 123–135, City, Country, 2021. doi: 10.1234/example.doi. 2.1

[26] OpenAI. GPT-4.1 Technical Report. `https://openai.com/index/gpt-4-1`, 2024. 3.2, 4.3.2

[27] OpenAI. GPT-4.1-mini model card. `https://platform.openai.com/docs/models/gpt-4`, 2024. 4.3.2

[28] Ruwei Pan and Hongyu Zhang. Modularization is better: Effective code generation with modular prompting, 2025. URL `https://arxiv.org/abs/2503.12483`. 2.2

[29] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, December 1972. ISSN 0001-0782. doi: 10.1145/361598.361623. URL `https://doi.org/10.1145/361598.361623`. 3.3

[30] Jeff Patton. *User Story Mapping: Discover the Whole Story, Build the Right Product*. O'Reilly Media, Sebastopol, CA, 2014. ISBN 978-1491904909. 3.3

[31] Roger S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill Higher Education, 5th edition, 2001. ISBN 0072496681. 2.1

[32] Qwen Team. Qwen3 1.7B: A Strong Open LLM. `https://huggingface.co/Qwen/Qwen3-1.7B`, 2024. 4.3.2

[33] Eric S. Raymond. *The cathedral and the bazaar: musings on Linux and open source by an accidental revolutionary*. O'Reilly Media, Beijing; Cambridge; Farnham; Köln; Paris; Sebastopol; Taip, 2., überarb. und erw. a. edition, 2001. ISBN 0-596-00108-8. URL `http://www.catb.org/~esr/writings/cathedral-bazaar/cathedral-bazaar/`. With a foreword by Bob Young. 1

[34] Yongliang Shen, Kaitao Song, Xu Tan, Wenqi Zhang, Kan Ren, Siyu Yuan, Weiming Lu, Dongsheng Li, and Yueting Zhuang. Taskbench: benchmarking large language models for task automation. In *Proceedings of the 38th International Conference on Neural Information Processing Systems*, NIPS '24, Red Hook, NY, USA, 2025. Curran Associates Inc. ISBN 9798331314385. 3.1

[35] Ian Sommerville. *Software Engineering*. Addison-Wesley Publishing Company, USA, 9th edition, 2010. ISBN 0137035152. 2.1

[36] Christoph Stanik, Lloyd Montgomery, Daniel Martens, Davide Fucci, and Walid Maalej. A Simple NLP-Based Approach to Support Onboarding and Retention in Open Source Communities. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 172–182, September 2018. doi: 10.1109/ICSME.2018.00027. URL `https://ieeexplore.ieee.org/document/8530027/`. ISSN: 2576-3148. 1

[37] Davide Taibi, Valentina Lenarduzzi, Muhammad Ovais Ahmad, and Kari Liukkunen. Com-

paring communication effort within the scrum, scrum with kanban, xp, and banana development processes. *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering*, 2017. URL https://api.semanticscholar.org/CorpusID:20567594. 2.1

[38] PyTorch Team. Pytorch. https://github.com/pytorch/pytorch, 2023. GitHub repository, accessed 2025-07-16. 3.1

[39] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, and et al. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems*, volume 35, pages 24824–24837, 2022. 2.2

[40] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: agent-computer interfaces enable automated software engineering. In *Proceedings of the 38th International Conference on Neural Information Processing Systems*, NIPS '24, Red Hook, NY, USA, 2025. Curran Associates Inc. ISBN 9798331314385. (document), 1, 2.2, 4.1

[41] Ed Yourdon and Larry L. Constantine. *Structured design : fundamentals of a discipline of computer program and systems design*. Prentice Hall, Englewood Cliffs, N.J, 1979. ISBN 0138544719. 3.3

[42] Kechi Zhang, Jia Li, Ge Li, Xianjie Shi, and Zhi Jin. CodeAgent: Enhancing code generation with tool-integrated agent systems for real-world repo-level coding challenges. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar, editors, *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 13643–13658, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.737. URL https://aclanthology.org/2024.acl-long.737/. 2.2