Ionian University

Department of Informatics

-- Master's Thesis --

AI-Assisted CLI vs Traditional CLI
A Comparative Analysis

Thanasis Gliatis

Supervisor: Dr. Konstantinos Chorianopoulos

July 25, 2025

Supervisor

Dr. Konstantinos Chorianopoulos, *Title*,
Department

Examination Committee

Dr. Konstantinos Chorianopoulos, *Professor*,
Department of Informatics
Dr. Katia - Lida Kermanidou, *Professor*,
Department of Informatics
Dr. Stergios Palamas, *Professor*,
Department of Informatics

This thesis is submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science

Abstract

This thesis presents a comprehensive comparative analysis between traditional command-line interfaces (CLI) and aiman, an AI-assisted CLI system. The research evaluates the effectiveness of artificial intelligence in enhancing CLI usability by providing intelligent command suggestions, error correction, and contextual guidance.

The study employs a controlled experimental design to measure key performance metrics including task completion time, success rates, number of attempts, and user satisfaction across ease of use, confidence, and frustration dimensions. Through systematic user testing with experienced software engineers, this research demonstrates the potential of AI-driven solutions to address the traditional challenges of CLI usage, particularly the error-prone nature of command-line interactions.

The aiman implementation leverages large language models to provide real-time assistance, offering reactive error correction and suggestions when commands fail. Results indicate significant improvements in user efficiency and error correction when AI assistance is available, while maintaining the power and flexibility that make CLI tools essential for technical users.

This work contributes to the growing field of AI-assisted computing by providing empirical evidence for the benefits of intelligent CLI interfaces and establishing a framework for evaluating AI-enhanced command-line tools.

Acknowledgments

I would like to express my sincere gratitude to all those who contributed to the completion of this thesis.

First and foremost, I extend my deepest appreciation to my supervisor, Dr. Konstantinos Chorianopoulos, for their invaluable guidance, continuous support, and expert feedback throughout this research project. Their insights and encouragement were instrumental in shaping this work.

I am grateful to the members of my examination committee for their time, constructive feedback, and thoughtful evaluation of this research.

Special thanks go to all the participants who volunteered their time for the user testing sessions. Their willingness to engage with the experimental setup and provide honest feedback was essential for the empirical validation of this research.

I would like to acknowledge the open-source community and the developers of the various AI-assisted CLI tools that served as inspiration and reference for this work. The collaborative nature of software development continues to drive innovation in this field.

My appreciation extends to the faculty and staff of the Department of Informatics at Ionian University for providing an excellent academic environment and access to the resources necessary for this research.

Finally, I am deeply grateful to my family and friends for their unwavering support, patience, and understanding throughout the duration of this study. Their encouragement kept me motivated during the challenging phases of this work.

This thesis would not have been possible without the collective contributions of all these individuals and institutions.

Corfu, July 25, 2025

Thanasis Gliatis

Contents

List of Figures

List of Tables

Chapter A

Introduction

Command-line interfaces (CLI) remain indispensable for developers, system administrators, and power users despite the rise of graphical interfaces, due to their precision, efficiency, and powerful automation capabilities.

A.1 Background & Motivation

A.1.1 Importance of Command-Line Interfaces (CLI) in Computing

Command-line interfaces (CLI) provide rapid, precise interaction through text commands, enabling efficient automation and scripting capabilities indispensable for tasks such as DevOps pipelines, scheduled jobs, and remote server management. Unlike graphical user interfaces (GUIs), CLIs enable rapid execution of commands directly through text input.

The automation and scripting capabilities of CLI represent a crucial strength of this interface paradigm. CLI enables batch processing and automated workflows with minimal overhead, making it indispensable for cron jobs, DevOps pipelines, and other automated tasks. Furthermore, CLI allows users to build custom scripts using system commands, effectively creating lightweight, purpose-built applications. This scripting approach embodies a core aspect of the Unix UI philosophy, promoting flexibility through composability. Instead of relying on monolithic software solutions, users can combine small tools using pipes and redirection to solve complex problems efficiently.

Remote system management represents another domain where CLI proves essential. Man-

aging remote servers and cloud computing environments relies heavily on command-line interfaces, with tools like SSH enabling secure remote administration that makes CLI indispensable in modern IT operations. Additionally, many advanced system configurations and functionalities are only accessible through the command line, providing users with granular control over software and system components that may not be available through graphical interfaces.

CLI's resource efficiency ensures its continued relevance in modern computing. Unlike GUI applications, CLI does not require significant system resources, making it particularly suitable for resource-constrained environments. This characteristic has led to its widespread adoption in embedded systems, Linux servers, and minimalistic operating systems where resource optimization is paramount.

CLI's cross-platform compatibility further enhances its appeal, as tools and commands are often standardized across different operating systems. This standardization enables developers to use CLI tools consistently across Linux, macOS, and Windows environments without learning multiple GUI variations, thereby improving productivity and reducing the cognitive load associated with platform-specific interfaces.

Despite these advantages, CLI has a steep learning curve, requiring users to memorize commands and their syntax [2]. This limitation has led to the exploration of AI-driven solutions that enhance CLI usability by providing intelligent command suggestions and error corrections.

A.1.2 Challenges of Traditional CLI: Steep Learning Curve, Error-Prone Usage

While command-line interfaces (CLI) offer efficiency and control, they also present several challenges that can hinder their widespread adoption, particularly among novice users.

CLI's complexity presents significant usability challenges, especially for novices. The reliance on memorizing complex syntax without visual guidance can be intimidating, as CLI requires users to memorize a vast number of commands and their syntax [2]. This memorization challenge is compounded by the complexity inherent in CLI syntax, as many commands feature multiple parameters and options that can be difficult for beginners to grasp without extensive documentation [2]. Moreover, CLI environments lack the visual cues that users have come to expect from modern interfaces, making it considerably harder

for users to explore available options compared to GUI-based tools [1].

The error-prone nature of CLI usage further exacerbates these usability challenges. Command sensitivity represents a persistent source of frustration, as CLI commands are typically case-sensitive and require precise syntax, leading to frequent errors when commands are not typed correctly [2]. This precision requirement becomes particularly problematic when considering the potential for system damage, as certain commands (such as `rm -rf /`) can cause irreversible damage if used incorrectly, posing significant risks to system stability and data integrity. Further compounding the difficulty is the absence of detailed error guidance typical in GUI applications, as CLI typically provides limited or ambiguous error messages, making troubleshooting particularly difficult for inexperienced users [2].

These challenges highlight the need for AI-driven solutions that enhance CLI usability by assisting users with command recommendations, error detection, and contextual guidance, thereby making CLI more accessible and less error-prone.

A.1.3 Emergence of AI-Assisted CLI: Bridging Usability Gaps

To address these usability barriers, recent advances have introduced AI-driven solutions designed to bridge the gaps between human intuition and CLI syntax, significantly enhancing accessibility and reducing errors. AI-assisted CLI tools leverage machine learning, natural language processing, and large language models to create more intuitive and user-friendly command-line experiences.

These AI-driven enhancements manifest in several key areas that directly address traditional CLI limitations. Intelligent command generation enables AI systems to interpret user intent expressed in natural language and translate it into appropriate CLI commands [10], effectively bridging the gap between human communication patterns and machine-readable syntax. Complementing this capability, error detection and correction mechanisms allow AI systems to identify syntax errors, suggest corrections, and provide explanations for command failures, reducing the frustration associated with traditional CLI error handling. Additionally, contextual assistance provided by AI-powered CLI tools offers relevant suggestions based on the current working environment, file structure, and user history, creating a more personalized and adaptive user experience.

A.2 Research Objectives

This research aims to evaluate the impact of AI-assisted CLI on usability, error reduction, and user experience, focusing strictly on measurable outcomes. The investigation centers on three primary objectives that collectively address the fundamental questions surrounding AI integration in command-line environments:

- **Comparative Analysis:** Evaluate AI-assisted CLI versus traditional CLI on efficiency, accuracy, and usability, with particular attention to how AI-assisted error state fixing and corrective suggestions improve user productivity when commands fail.
- **Usability & Error Reduction:** Quantify how AI-driven corrections reduce errors and improve command execution success rates, while analyzing user feedback on AI-assisted error correction features.
- **User Experience:** Measure the real-world effectiveness and user satisfaction through controlled experimental tests, tracking key performance metrics such as time-to-correct, number of errors, and user satisfaction levels.

By addressing these objectives, this study aims to provide insights into the role of AI in transforming CLI interactions, making them more intuitive and user-friendly while maintaining their efficiency and flexibility.

A.3 Scope of the Thesis

The thesis investigates how AI-driven command suggestions and error-correction tools improve CLI usability. It measures their effectiveness (task success), accuracy (correctness of suggestions), and efficiency (speed of error resolution). The research is specifically centered on AI-generated command suggestions, error detection, and correction mechanisms within CLI environments, with particular emphasis on evaluating AI-driven error correction to reduce execution failures and improve command efficiency.

The study methodology centers on comprehensive analysis across the three critical dimensions outlined in the research objectives. This investigation includes assessing the ability

of AI to provide context-aware recommendations that help users avoid common mistakes, thereby addressing one of the most significant barriers to CLI adoption among novice users. The evaluation demonstrates the practical benefits of AI integration in terms of measurable performance improvements.

The aim is to provide empirical evidence demonstrating how generative AI enhances command-line interaction, reduces errors, and promotes accessibility. The findings will contribute to the ongoing development of intelligent CLI tools that enhance productivity, reduce errors, and make command-line operations more accessible.

Chapter B

Literature Review

This section provides a comprehensive review of existing research and frameworks related to command-line interfaces (CLI), highlighting their usability challenges and the role of AI in improving their effectiveness. By examining past studies, this literature review establishes the need for AI-driven enhancements in CLI environments and positions this research within the broader landscape of AI-assisted computing.

B.1 Traditional CLI Usability & Challenges

B.1.1 Learning Curve Issues

Traditional CLI requires users to memorize a vast number of commands, arguments, and syntax rules, making it difficult for beginners [2]. Unlike graphical user interfaces (GUI), CLI lacks visual aids, requiring users to rely heavily on documentation and prior knowledge [1]. The absence of interactive guidance makes learning CLI commands a slow and error-prone process.

B.1.2 Error Patterns and User Difficulties

CLI error patterns fall into several interconnected categories that compound user difficulties and create barriers to effective command-line usage. Syntax errors represent the most frequent challenge, as CLI commands often fail due to incorrect syntax, missing arguments, or misused flags. This problem is exacerbated by the lack of real-time feedback—users

only discover errors after execution, leading to frustrating trial-and-error cycles that can significantly slow task completion.

The consequences of CLI errors extend beyond simple inconvenience. Some commands, particularly system-level operations like `rm -rf`, can have destructive and irreversible effects if used incorrectly, posing serious risks to system stability and data integrity. These high-stakes scenarios create anxiety for users and contribute to the perception that CLI tools are dangerous for inexperienced operators.

Command inconsistencies across different CLI tools further complicate the user experience. Unlike graphical interfaces that often follow consistent design patterns, CLI tools frequently employ varying syntax conventions, flag formats, and parameter structures. This inconsistency creates additional cognitive load for users who must maintain mental models for multiple command syntaxes when switching between different tools and environments.

B.2 AI-Enhanced Command-Line Interfaces

The integration of artificial intelligence into command-line interfaces represents a paradigm shift in addressing the usability challenges identified in traditional CLI environments. AI-powered CLI tools offer the potential to significantly improve user experience by providing intelligent command generation, real-time error correction, and predictive assistance. These systems aim to bridge the gap between CLI power and accessibility, reducing the cognitive burden of command memorization while minimizing the trial-and-error cycles that characterize traditional CLI interactions.

The emergence of AI-assisted CLI tools is driven by advances in natural language processing, machine learning, and large language models that can understand user intent and provide contextually appropriate assistance. This technological convergence enables CLI systems that maintain the flexibility and efficiency of traditional command-line environments while offering the guidance and error prevention typically associated with graphical interfaces.

B.2.1 Natural Language Command Generation

Modern AI-powered CLI tools employ sophisticated natural language processing techniques to interpret user intent and generate appropriate commands, significantly reducing the learning curve for new users [10]. These systems utilize sequence-to-sequence (Seq2Seq) models that leverage deep learning techniques to predict and suggest commands based on historical usage patterns and contextual input [7]. By analyzing user behavior and command sequences, these models can anticipate user needs and offer structured suggestions that reduce cognitive load, particularly for complex multi-parameter commands that traditionally require extensive documentation consultation.

The effectiveness of AI-assisted command generation lies in its ability to bridge the semantic gap between user intention expressed in natural language and the precise syntax required by command-line tools. This approach transforms the CLI experience from one requiring exact syntax memorization to one supporting more intuitive, conversational interaction patterns.

B.2.2 Intelligent Error Recovery and Correction

Large language models, particularly systems like OpenAI's GPT, have demonstrated significant capabilities in analyzing incorrect commands and providing real-time corrections and explanations [10]. These systems excel at context-aware assistance, helping users understand not just what went wrong, but why a command failed and how to fix it effectively. This capability addresses one of the most significant pain points in traditional CLI usage—the trial-and-error cycles that result from cryptic error messages and lack of guidance.

Advanced AI-assisted CLI systems extend beyond simple error correction to provide comprehensive documentation support. AI-generated explanations offer instant access to best practices, alternative command suggestions, and contextual help that adapts to the user's current working environment [9]. This dynamic documentation approach represents a significant improvement over static manual pages, providing personalized assistance that evolves with user needs and system context.

B.2.3 Comparative Analysis of AI-Enhanced CLI Tools

The current landscape of AI-assisted CLI tools exhibits significant diversity in approach and capability. To systematically evaluate these systems, we can categorize them across five critical dimensions that reflect both technical capabilities and practical usability considerations. These dimensions include error correction capabilities, which measure the AI's ability to detect and fix syntax or logical errors; disambiguation and context awareness, reflecting the system's capacity to resolve ambiguous commands and adapt suggestions based on user history; failure explanation quality, assessing how effectively the AI communicates the reasons for command failures; system safety and execution prevention, evaluating the tool's ability to prevent destructive operations; and portability with research analytics, considering cross-platform compatibility and support for empirical evaluation.

This multi-dimensional analysis reveals distinct patterns in current AI-CLI implementations, with most tools excelling in error correction while showing varied performance in context awareness and safety features. The comparative framework helps identify both the current state of the field and opportunities for future development in AI-assisted command-line interfaces.

Tool	**Error Correction**	**Context Awareness**	**Failure Explanation**	**System Safety**	**Open Source**
aiman	✓	✗	✓	✓	✓
GitHub Copilot CLI	✓	✗	✓	✗	✗
ShellGPT	✓	✗	✓	✗	✓
Neural Shell (nlsh)	✓	✓	✓	✓	✓
TLM	✓	✗	✓	✓	✓
Warp	✓	✗	Partial	✓	✗
Wave Terminal	✓	✓	✓	✗	✓
iTerm2 + ChatGPT	✓	✗	✓	✗	Partial

Table B.1: *Comparative Analysis of AI-Enhanced CLI Tools*

B.3 Research Foundations and Related Work

The development of AI-enhanced command-line interfaces builds upon a rich foundation of research spanning natural language processing, human-computer interaction, and machine learning. This interdisciplinary convergence has created the theoretical and technical groundwork necessary for intelligent CLI systems that can understand user intent, predict command needs, and provide contextual assistance.

B.3.1 Natural Language Processing and Command Translation

The translation of natural language queries into executable CLI commands represents a fundamental challenge that has attracted significant research attention. Pioneering work by Lin et al. (2018) established the NL2Bash dataset and benchmark, creating a standardized framework for evaluating natural language to Bash command translation systems [6]. This foundational dataset has enabled systematic comparison of different approaches and provided a common evaluation framework that has accelerated progress in the field.

The success of natural language command translation depends critically on bridging the semantic gap between informal user expressions and precise command syntax. This challenge involves not only syntactic transformation but also semantic understanding of user intent, context inference, and knowledge of command-line tool capabilities and limitations.

B.3.2 Predictive Models and Command Suggestion Systems

Early research in command prediction demonstrated the feasibility of using machine learning to anticipate user needs in CLI environments. Hirsh and Davison's seminal work on adaptive UNIX command-line assistants showed that predictive models could achieve significant accuracy in suggesting next commands based on usage history and context [3]. This research established the theoretical foundation for intelligent command suggestion systems and demonstrated that user behavior patterns in CLI environments contain sufficient regularity to support predictive assistance.

Contemporary approaches have expanded these early insights using advanced machine learning techniques. Recent studies have demonstrated that modern sequence-to-sequence

models can effectively predict user intent and suggest appropriate commands based on contextual information, user history, and environmental factors [7]. These advances represent a significant evolution from rule-based prediction systems to more sophisticated models capable of handling complex, multi-step command sequences.

B.3.3 Human-Computer Interaction and CLI Usability

Foundational research in human-computer interaction has provided crucial insights into the usability challenges that motivate AI-assisted CLI development. Extensive user experience research has consistently identified the steep learning curve as the primary barrier to CLI adoption, with studies documenting the cognitive burden imposed by command memorization requirements [2]. Comparative studies of GUI and CLI interactions have revealed that while command-line interfaces offer superior efficiency for experienced users, the initial learning investment creates a significant accessibility barrier for newcomers [1].

This HCI research has also illuminated the error patterns and recovery strategies that characterize CLI usage. Studies of CLI error patterns have identified recurring categories of mistakes, including syntax errors, parameter misuse, and potentially destructive command execution [2]. Understanding these error patterns has been crucial for designing AI systems that can provide targeted assistance where users most commonly encounter difficulties, informing the development of intelligent error detection and correction systems that can intervene before mistakes occur or provide effective recovery guidance when errors do happen.

Chapter C

Methodology

C.1 aiman Implementation

C.1.1 System Architecture and Workflow

The aiman tool is built using TypeScript and implements an intelligent command-line interface that helps users correct command-line operations. The system architecture follows a modular design that separates concerns between user interaction, command execution, AI processing, and data collection components.

C.1.1.1 Core Components

The CLI interface component (src/cli.ts) serves as the main entry point for user interaction, providing a colorful, user-friendly interface using chalk for visual feedback and readline for sophisticated input handling. This component manages the primary user experience and coordinates interactions with other system components.

Command execution functionality (src/commands.ts) handles the core operational requirements through Node.js child processes, capturing stdout, stderr, and exit codes to provide structured output for further processing. This component ensures reliable command execution while maintaining comprehensive logging of all system interactions.

Figure C.1 illustrates the high-level architecture of the aiman system, showing the interaction flow between user input, AI processing, and command execution.

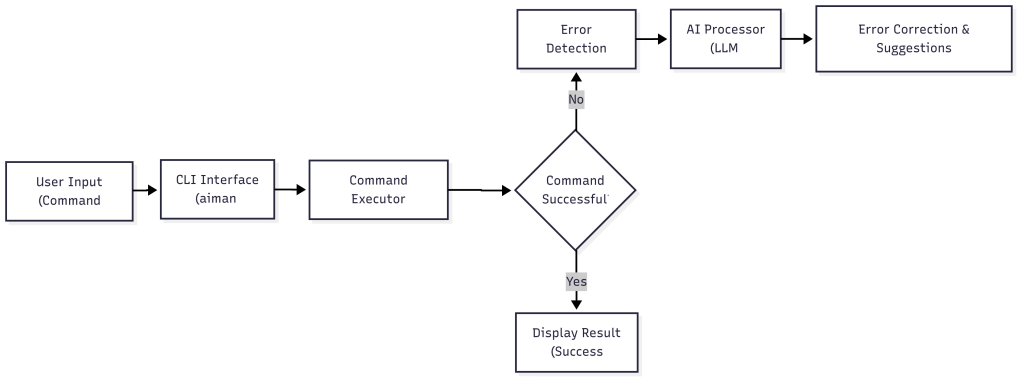

Figure C.1: *aiman System Architecture Flow*

Event handling (src/events.ts) manages the coordination between different system components, processing user input and orchestrating the interaction between command execution and AI assistance. This component handles error scenarios and AI help requests, ensuring smooth operation across different usage patterns.

C.1.2 How AI Generates and Corrects Commands

The AI correction system follows a structured workflow that analyzes failed commands and provides contextual assistance to users. This process begins when the system detects command failures and proceeds through a systematic analysis and correction pipeline.

C.1.2.1 AI Processing Pipeline (src/llm.ts)

When a command fails, the system captures comprehensive information including the original command, error output, and system environment information. This data collection ensures that AI assistance is provided with complete context for generating accurate corrections and explanations.

The AI help generation system provides two levels of assistance tailored to different user needs. Short help (getShortHelpForFailedCommand) provides concise assistance with essential correction information:

```
{
    error_explanation: string;
```

```
    corrected_command: string;
    explanation: string;
    tips: string;
}
```

Detailed help (getHelpForFailedCommand) offers comprehensive assistance for users requiring more extensive guidance:

```
{
    error_explanation: string;
    corrected_command: string;
    arguments_explanation: string;
    best_practices: string;
}
```

Response processing ensures system reliability through validation of JSON responses, calculation of API usage costs, and formatting of output with color coding for enhanced user experience. This processing layer maintains consistency and provides transparency about system resource usage.

C.1.2.2 Environment Awareness

The system (environments.ts) incorporates comprehensive environment detection capabilities that enhance the accuracy and relevance of AI assistance. These capabilities include detection of OS type and version, identification of terminal environment characteristics, provision of environment context to AI responses, and ensuring command compatibility across different system configurations. This environment awareness allows the AI to provide platform-specific recommendations and avoid suggesting commands that may not be available or behave differently across systems.

C.1.3 Data Storage and Analytics

The Store class (‘evaluation/store.ts‘) implements a comprehensive data management system designed for rigorous research data collection and analysis. The system employs

UUID-based session tracking to maintain participant anonymity while enabling longitudinal analysis across multiple study sessions. Each participant session is uniquely identified and stored with complete metadata including timestamps, user demographics, and experimental conditions.

The data storage architecture uses JSON-based persistent storage with automatic file handling and directory creation. The system maintains both individual session data and aggregated multi-session datasets, automatically handling data format migrations and backwards compatibility. Performance metrics are collected at multiple granularities, including per-attempt timing data, command execution results, error classifications, and success indicators.

The Store class provides sophisticated session management capabilities, including condition order tracking for counterbalanced experimental designs, pre- and post-questionnaire data integration, and comprehensive test result aggregation. This enables researchers to analyze learning effects, fatigue patterns, and condition-specific performance variations across participants.

C.1.4 Testing Framework Architecture

The evaluation framework consists of four interconnected components that work together to provide systematic data collection and analysis capabilities:

C.1.4.1 Test Execution Engine

The `Test` class (`evaluation/test.ts`) executes individual command-line tasks. Each test instance contains a CLI challenge with multiple acceptable solutions, category classification (file navigation, file management, text processing, etc.), and optional pre-execution setup commands. The system validates commands using a two-tier assessment: exact string matching for known correct commands, followed by AI-based equivalence evaluation using the `compareCommandAndResults` function.

The test execution records timing data including user typing patterns, command execution duration, and total task completion time. Error handling categorizes failures into execution errors, incorrect commands, and skipped attempts. The system provides console feedback and logs all interactions for analysis.

C.1.4.2 AI-Powered Command Assessment

The framework uses AI validation through the `compareCommandAndResults` function in `llm.ts`. This function determines when alternative command formulations achieve equivalent results to expected solutions. The AI assessment evaluates both command syntax and execution output to recognize alternative approaches that exact-match systems would mark as incorrect.

The AI validator includes mechanisms to detect potentially dangerous commands (system deletion, security vulnerabilities) and provides explanations for assessment decisions. This approach handles complex commands with multiple valid formulations that would be difficult to enumerate manually.

C.1.4.3 Orchestration and Condition Management

The `user-tests.ts` module implements the study orchestration logic, managing participant flow through the complete experimental protocol. The system handles counterbalanced condition ordering (traditional-first vs. AI-first) with automatic alternation to prevent order effects. Command-line argument parsing enables flexible study configuration, including questionnaire skipping for rapid testing, test count limitations for pilot studies, and condition order forcing for debugging.

The orchestration system integrates pre- and post-study questionnaires, manages informed consent procedures, and provides comprehensive study information presentation. Progress tracking and session state management ensure reliable data collection even in the event of interruptions or technical difficulties.

C.1.4.4 Interactive Command Processing

The framework processes user commands through the `handleUserInput` function in `events.ts`, which provides different behavior based on experimental condition. In traditional mode, commands are executed with standard system feedback. In AI-assisted mode, failed commands trigger help generation through the `getShortHelpForFailedCommand` function, providing assistance without revealing solutions.

The command processing system includes error handling, output formatting, and timing

instrumentation. Progress indicators provide user feedback during AI processing, and logging captures user interaction patterns and system performance.

This framework enables comparison of traditional and AI-assisted CLI interfaces while maintaining experimental controls and data collection for usability research.

C.1.5 Error Handling and User Assistance

The system implements error handling that provides context-aware error messages to help users understand command failures. The mechanism suggests corrections based on common patterns and provides guidance for future usage. The error handling approach addresses immediate problems while providing educational information.

This implementation executes commands and provides error correction and assistance. The modular architecture separates concerns while the data collection capabilities support experimental evaluation.

C.1.5.1 Cost Management and Optimization

The system implements transparent cost tracking through calculateCost in utils.ts, which tracks prompt and completion tokens separately, uses different rates for different token types, and provides transparency by showing costs to users. This cost management approach ensures responsible resource usage while maintaining user awareness of system expenses.

C.1.5.2 Extensibility

The system demonstrates strong modular design principles through its TypeScript architecture with separate components for CLI, commands, events, LLM processing, and evaluation functions. Additionally, the system provides configurable test scenarios through JSON configuration files, extensible metrics collection with customizable data storage, and command-line argument parsing for different operational modes. These design choices highlight the maintainability of the aiman implementation, demonstrating how it separates concerns between command execution, AI processing, and data collection to provide researchers with a flexible tool for conducting CLI usability experiments.

C.2 Comparative Testing Framework

This experiment compares different command-line interfaces (CLI) in terms of usability and error correction effectiveness. The study evaluates two variations to capture the effects of AI assistance:

1. **Traditional CLI (Control):** The native Linux/macOS terminal without AI assistance.
2. **aiman (AI Assistance on Error):** The AI provides suggestions only when the user enters an invalid or failing command.

C.2.1 Primary Objectives (Dependent Variables)

The primary objective focuses on measuring efficiency through how effectively users complete CLI tasks, including metrics such as task completion time, number of errors, and retry attempts. This efficiency measurement provides quantitative evidence of the practical benefits offered by AI assistance in command-line environments.

Command-line interfaces (CLIs) offer powerful capabilities, but their usability is often limited by a steep learning curve, requiring memorization of syntax, flags, and command structures. Prior research highlights how users—especially novices—struggle with syntactic errors, conceptual mismatches, and the indirectness of interaction [2]. To mitigate these challenges, intelligent agents that learn from user behavior and suggest or correct commands have been proposed. For example, Hirsh & Davison (1997) embedded a predictive assistant in tcsh that suggested next commands based on history, achieving up to 67% prediction accuracy [3]. Their system demonstrated the feasibility of integrating machine learning into interactive CLI environments to support user intent.

Building on this foundation of predictive and corrective CLI support, our work evaluates whether real-time AI assistance can improve usability in error-prone scenarios. Unlike Hirsh & Davison, who focused on prediction accuracy, we focus on user outcomes—task success, correction time, and satisfaction—aligning with calls in their paper for more meaningful performance measures. Our task set is grounded in empirical studies of real-world shell usage: Gharehyazie et al. (2016) analyzed over 1 million shell commands across 263 users and found that cd, ls, grep, find, and other file-management commands

are central to everyday shell usage [4]. Additionally, we incorporate the novice-focused curriculum structure outlined by Wilson (2016), in which Software Carpentry emphasizes a canonical progression through Unix shell, Git version control, and Python/R scripting as core competencies [5].

C.2.2 Participant Selection and Recruitment

The study targets experienced software engineers with 3–10 years of professional experience as the primary participant group. This population is selected because they are proficient with computers and likely familiar with CLI basics, which helps focus the study on efficiency gains and subtle improvements rather than basic learning effects.

Participant selection follows specific inclusion and exclusion criteria to ensure appropriate study population characteristics. Inclusion criteria encompass professional software development experience (3+ years), regular computer usage in professional contexts, basic familiarity with terminal/command-line environments, and willingness to participate in controlled testing sessions. Exclusion criteria eliminate expert-level CLI masters (to avoid ceiling effects) and complete CLI novices (to ensure task feasibility).

Within-subjects designs typically require fewer participants than between-subjects designs because each person serves as their own control, reducing individual variation. Based on effect size estimates from similar HCI studies and practical feasibility constraints, we target 6-10 participants to achieve adequate statistical power for detecting meaningful differences in task completion time, success rates, and number of attempts.

Participants are recruited through professional networks, software development communities, and academic contacts. Prior to the experiment, each participant completes a pre-survey detailing their background (years of experience, self-rated CLI proficiency, usage frequency, etc.) to contextualize the results and ensure they meet inclusion criteria.

C.2.3 Apparatus and Environment

The experiment was conducted on a Linux virtual machine (VM) in a Dockerized environment accessed via SSH. This setup provides a standardized CLI environment for all participants, ensuring consistency across testing sessions and eliminating variations in local system configurations.

The baseline interface utilizes the native terminal (e.g. Bash or Zsh shell on a typical Terminal app), providing no special assistance beyond standard shell features. In contrast, the aiman tool integrates a generative AI assistant into the terminal, which can detect command errors or incomplete commands and suggest corrections or completions in real-time. Key features of aiman include syntax correction and suggestions when a command fails or is likely incorrect, and enhanced help or examples when the user is unsure of a command.

An interactive testing harness (included in the aiman codebase) presents tasks and records data systematically. This harness presents task descriptions to users in the terminal, accepts their command input, and automatically logs outcomes. In traditional mode, it records the commands entered and determines task success by monitoring command exit codes (0 for success, non-zero for failure) and validating expected output patterns. In aiman mode, it also logs AI suggestions made and whether the user accepted or ignored them.

All sessions are conducted within the same Docker container environment to ensure identical system configurations. Participants connect to the VM via SSH, providing a consistent terminal experience regardless of their local machine. Internet access is available for tasks requiring network operations, and comprehensive logging captures all command inputs and outputs through the built-in JSON-based data collection system. Before starting, each system is reset to a clean state (e.g. clearing any command history that could give hints) and loaded with any files or directories needed for the tasks (for example, sample directories to search in, or specific files to manipulate).

C.3 Data Collection & Evaluation Metrics

Multiple quantitative metrics are collected to evaluate performance on each task and across conditions. Table C.1 summarizes the key metrics and their definitions:

All these metrics align with standard usability measures: efficiency (time and attempts), effectiveness (success rates and error patterns), and satisfaction (ease of use, confidence, and frustration ratings). Data is collected through automated logging within the interactive testing harness, which captures all command inputs, outputs, timing information, and AI interactions in JSON format.

Metric	Description	Collection Method
Task Completion Time	Time from task presentation to successful completion (seconds)	Automatic timestamps by testing harness
Success Rate	Binary task completion success (success/failure)	Derived from command logs and exit codes
Number of Attempts	Total command attempts before task completion	Count from command log per task
Error Rate	Percentage of tasks with execution errors, incorrect commands, or skips	Calculated from error classifications in logs
Ease of Use Rating	Comparative assessment of interface usability (5-point scale)	Post-experiment Likert questionnaire (1=Traditional easier, 5=AI easier)
Confidence Rating	Comparative assessment of user confidence (5-point scale)	Post-experiment Likert questionnaire (1=Traditional more confident, 5=AI more confident)
Frustration Rating	Comparative assessment of interface frustration (5-point scale)	Post-experiment Likert questionnaire (1=Traditional more frustrating, 5=AI more frustrating)

Table C.1: *Key Metrics for Evaluation*

C.4 Data Analysis and Visualization Tools

The experimental data collected through the aiman testing framework is processed using a custom analysis pipeline to extract insights from CLI usability experiments. This section outlines the analysis approach and tools used to process and visualize the experimental results.

C.4.1 Analysis Pipeline Overview

The data analysis follows a two-stage pipeline that transforms raw experimental data into statistical summaries and visualizations. The first stage focuses on statistical analysis, where raw JSON data from participant sessions is processed to calculate key performance

metrics, error patterns, and user satisfaction measures. The analysis tool handles data validation, calculates derived metrics, and generates statistical summaries across multiple dimensions including overall performance, task categories, participant demographics, and CLI proficiency correlations.

The second stage involves visualization generation, where statistical results are converted into charts and graphs using a visualization pipeline. This includes performance comparison charts, error analysis visualizations, user satisfaction plots, and demographic distributions.

C.4.2 Analysis Capabilities

The analysis pipeline covers experimental data across several analytical dimensions. Performance analysis encompasses success rates, completion times, and attempt counts for both traditional and AI-assisted conditions, task category breakdowns (file navigation, text processing, system administration, etc.), and individual test performance comparisons.

Error pattern analysis provides classification and quantification of error types across interface conditions, user-generated error analysis (excluding prefilled incorrect commands), and common failure pattern identification with AI correction effectiveness assessment.

User experience analysis processes Likert scale responses for ease of use, confidence, and frustration metrics, conducts preference rating analysis with statistical summaries, and performs correlation analysis between objective performance and subjective satisfaction.

Demographic and proficiency analysis examines performance variation across participant characteristics, CLI proficiency correlation with task success rates, and condition order effect analysis for experimental validity.

C.4.3 Data Processing and Quality Assurance

The analysis tools include comprehensive data processing with quality assurance measures. Data integrity validation ensures completeness and structure, with appropriate handling of missing or incomplete entries. Statistical robustness is maintained through proper handling of edge cases, small sample sizes, and missing data points with transparent reporting of limitations. Reproducibility is ensured through consistent processing algorithms with deterministic output generation for replication of analysis results.

C.4.4 Output Generation

The analysis pipeline produces multiple output formats to support different research and presentation needs. Statistical reports provide text-based summaries with key findings, performance metrics, and statistical indicators. Structured data exports in JSON and CSV formats enable integration with external statistical software and further analysis. Visualizations include charts for performance comparisons, error analysis plots, user satisfaction graphics, and demographic distributions.

This analysis infrastructure processes experimental data while maintaining flexibility to adapt to different research questions and presentation requirements.

Chapter D

Experimentation & User Testing

D.1 Test Environment & Setup

We will use a within-subjects experimental design where each participant performs tasks under both conditions: (A) using the traditional CLI and (B) using aiman. In other words, every participant experiences both interfaces, allowing direct comparison of their performance with and without AI support. This design controls for individual differences in skill, since each person serves as their own control.

To avoid order effects (learning or fatigue influencing results), the sequence of conditions will be counterbalanced:

- Half of the participants will use the Traditional CLI first, then aiman.
- The other half will use aiman first, then the Traditional CLI.

This counterbalancing ensures that any overall improvement due to practice is distributed across both conditions, and we can more confidently attribute differences in performance to the interface rather than task familiarity.

D.1.1 Participants

Following the recruitment strategy outlined in Chapter 3, we successfully recruited 6 participants who met our target criteria. The actual participant demographics are summarized below:

Demographics:

- **Age Distribution:** 4 participants aged 25–34, 2 participants aged 35–44
- **Gender:** 4 male, 2 female participants
- **Education:** 3 Bachelor's degrees, 2 Master's degrees, 1 High school
- **Professional Experience:** 5 participants with 5–10 years experience, 1 participant with 3–5 years experience
- **CLI Proficiency:** Self-rated on 1-5 scale: 2 participants (level 2), 3 participants (level 3), 1 participant (level 4)
- **Condition Order:** Counterbalanced with 3 participants starting with traditional CLI first, 3 starting with AI-assisted CLI first

All participants had daily or weekly CLI usage experience and basic terminal familiarity as required by our selection criteria (see Chapter 3, Section 3.2.2). The sample size of 6 participants proved sufficient for meaningful statistical comparisons in our within-subjects design, where each participant served as their own control.

D.1.2 Experimental Setup

The experimental apparatus and environment are described in detail in Chapter 3, Section 3.2.3. For the actual experimental sessions, each participant session followed a standardized protocol:

Session Preparation: Before each session, the Docker container was reset to a clean state, clearing command history and restoring the file system to its initial configuration with all necessary test files and directories.

Environment Consistency: All 6 participants used the identical Linux VM environment accessed via SSH, ensuring consistent testing conditions across all sessions.

Data Collection: The interactive testing harness automatically logged all command inputs, outputs, timing information, and AI interactions in JSON format for subsequent analysis.

D.1.3 Data Collection

The data collection methodology and evaluation metrics are detailed in Chapter 3, Section 3.3. During the experimental sessions, all metrics were automatically collected through the testing harness, including:

- Task completion times for each command scenario
- Success rates and error counts per participant and condition
- Command attempt logs with timestamps and error classifications
- AI interaction data (suggestions provided, accepted, or ignored)
- Post-session satisfaction questionnaire responses

This comprehensive data collection enabled detailed analysis of performance differences between traditional CLI and AI-assisted CLI conditions.

D.2 User Testing Scenarios

This section presents the task-based experimental framework developed to evaluate the impact of aiman interactions on user performance during command repair. The task set was constructed to reflect common command-line operations—such as file navigation, text processing, and system inspection—based on empirical usage patterns identified in large-scale studies of Unix shell behavior and established pedagogical frameworks.

Each task includes a broken command intentionally designed to elicit a syntax or semantic error. These errors were informed by prior human-computer interaction research that identified common CLI pitfalls for novice users, including syntactic memorization burdens, abstract command mappings, and high failure rates in manual correction.

The actual study used 13 carefully selected scenarios that represent common CLI operations and error patterns. Each scenario includes:

- A brief description of the user's intent
- The incorrect (broken) command

- One or more valid corrected commands
- The skill or concept being tested

D.2.1 File Navigation Scenarios

Scenario 1: Change to directory with spaces

- *Intent:* Navigate to a directory named 'Project Files' which contains spaces
- *Broken Command:* `cd Project\Files`
- *Correct Commands:* `cd "Project Files"`, `cd 'Project Files'`, `cd Project\\Files`
- *Tests:* Proper handling of spaces in directory names

Scenario 2: List files by size

- *Intent:* List all files sorted by size (largest first) with human-readable sizes
- *Broken Command:* `ls -lS --size-human`
- *Correct Commands:* `ls -lhS`, `ls -lah --sort=size`, `ls -la --sort=size -h`
- *Tests:* Combining flags for sorting and human-readable output

D.2.2 File Management Scenarios

Scenario 3: Find and remove empty directories

- *Intent:* Find and remove all empty directories under the current directory
- *Broken Command:* `find .  -type d -empty --delete`
- *Correct Commands:* `find .  -type d -empty -delete`, `find .  -depth -type d -empty -delete`
- *Tests:* Correct flag syntax for find command operations

Scenario 4: Create nested directories

- *Intent:* Create a directory structure for a new project, but only create directories that don't already exist
- *Broken Command:* `mkdir src/components/buttons src/components/forms src/utils`
- *Correct Commands:* `mkdir -p src/components/buttons src/components/forms src/utils`, `mkdir -p src/components/{buttons,forms} src/utils`
- *Tests:* Using -p flag for parent directory creation

Scenario 5: Safe file copying

- *Intent:* Copy file.txt to backup.txt interactively
- *Broken Command:* `cp -i backup.txt file.txt`
- *Correct Commands:* `cp -i file.txt backup.txt`, `cp --interactive file.txt backup.txt`
- *Tests:* Correct argument order for file operations

Scenario 6: Copy unique lines

- *Intent:* Copy only unique lines from file1.txt to file2.txt (removing duplicates)
- *Broken Command:* `sort file1.txt | unique > file2.txt`
- *Correct Commands:* `sort file1.txt | uniq > file2.txt`, `sort -u file1.txt > file2.txt`
- *Tests:* Correct command name and usage for duplicate removal

D.2.3 File Search and Text Search Scenarios

Scenario 7: Find recent text files

- *Intent:* Find all .txt files modified in the last 7 days
- *Broken Command:* `find . -name "*.txt" -mtime < 7`

- *Correct Commands:* `find . -name "*.txt" -mtime -7`, `find . -type f -name "*.txt" -mtime -7`
- *Tests:* Correct syntax for time-based file searches

Scenario 8: Search with context

- *Intent:* Search for the word "error" in log files, showing 2 lines of context before and after each match
- *Broken Command:* `grep -c 2 "error" *.log`
- *Correct Commands:* `grep -B 2 -A 2 "error" *.log`, `grep -C 2 "error" *.log`
- *Tests:* Context flags versus count flags in grep

D.2.4 File Viewing Scenarios

Scenario 9: View specific line range

- *Intent:* Extract lines 7-12 from a log file
- *Broken Command:* `head -n 12 log.txt | tail --n 6`
- *Correct Commands:* `head -n 12 log.txt | tail -n 6`, `sed -n "7,12p" log.txt`
- *Tests:* Correct flag syntax and command chaining

D.2.5 Text Processing Scenarios

Scenario 10: Count non-empty lines

- *Intent:* Count the number of non-empty lines in the log file
- *Broken Command:* `grep -v ^$ logfle.txt | wc -l`
- *Correct Commands:* `grep -v "^$" logfile.txt | wc -l`, `awk "NF" logfile.txt | wc -l`
- *Tests:* Filename typos and regex quoting

Scenario 11: Sort CSV by numeric column

- *Intent:* Sort this CSV file by the numeric values in the second column (descending order)
- *Broken Command:* `sort -c2,2nr data.csv`
- *Correct Commands:* `sort -t, -k2,2nr data.csv`, `sort -t, -k2nr data.csv`
- *Tests:* Field separator specification and key selection

D.2.6 System Information Scenarios

Scenario 12: Find largest subdirectories

- *Intent:* Find the top 3 largest subdirectories and show their sizes in human-readable format
- *Broken Command:* `du -h --depth=1 | sort -hr | head --3`
- *Correct Commands:* `du -h --max-depth=1 | sort -hr | head -3`, `du -sh */ | sort -hr | head -3`
- *Tests:* Correct flag names and argument syntax

Scenario 13: Find low disk space

- *Intent:* Identify filesystems with less than 20% free space remaining (more than 80% used)
- *Broken Command:* `df -h | awk '$5 > 80%'`
- *Correct Commands:* `df -h | awk '$5 > "80%"'`, `df -h | awk 'NR>1 && $5+0 > 80'`
- *Tests:* String comparison and numeric extraction in awk

D.2.7 Process Management Scenarios

Scenario 14: Find memory-intensive processes

- *Intent:* Find the top 5 processes consuming the most memory
- *Broken Command:* `ps aux | sorter --k4 -r | head -5`
- *Correct Commands:* `ps aux --sort=-%mem | head -5`, `ps aux | sort -k4nr | head -5`
- *Tests:* Command name correction and sorting by memory usage

Each scenario was designed to test specific CLI competencies while representing realistic tasks that software engineers encounter in their daily work. The broken commands contain common error patterns identified in CLI usage studies, including flag syntax errors, argument order mistakes, and command name confusion.

D.3 Procedure Flow

Each participant will go through the following phases:

Introduction & Consent: The researcher welcomes the participant, explains the study procedure, and obtains informed consent. Basic demographic and experience information is collected (if not already via a pre-survey).

Training/Tutorial: Before each interface, participants get a brief tutorial:

- For aiman, they will be shown how the tool works: for example, how it provides suggestions after an error, how to accept a suggestion, and any special commands to invoke help. They may practice with one simple example task to familiarize themselves with the AI features.
- The traditional CLI likely needs less introduction, but to ensure fairness, participants can be reminded of available manual help resources (e.g. man pages or –help flags) if they get stuck.

Baseline Task Set (Traditional CLI): In one of the two sessions (depending on counterbalancing), the participant uses the normal terminal to attempt a set of tasks (e.g. Tasks 1–11). They will:

- Read the task description provided by the experimenter or shown on-screen
- Attempt to execute the task in the terminal. They may use any knowledge they have, including –help or manual pages, but no AI assistance is available
- The system will log the time taken and whether the outcome is correct
- If they are truly stuck, they are allowed to ask for a hint or give up, but this will be recorded (with a reasonable maximum time per task, such as 5 minutes, to avoid endless frustration)

Experimental Task Set (aiman): In the other session, the participant uses the AI-enabled terminal to perform the same set of tasks using the aiman testing harness. The key difference is that AI assistance is provided reactively when command errors occur:

- Participants receive the same task descriptions and attempt to execute commands normally
- When a command fails (returns non-zero exit code), the aiman system automatically detects the failure and provides AI-generated suggestions for correction
- The AI assistance includes:
  - Error explanation describing why the command failed
  - Corrected command suggestion with proper syntax
  - Brief explanation of the correction
  - Additional tips for similar commands
- Participants can choose to:
  - Utilize the suggestion as is by copying and pasting it into the terminal
  - Modify the suggestion before executing
  - Ignore the suggestion and try their own approach

- The system logs all AI interactions including:
  - When AI suggestions were triggered (command failures)
  - What suggestions were provided
- No AI assistance is provided for successful commands - the system only intervenes when errors occur
- The same 10-minute time limit per task applies, with the same hint/give-up options available

This reactive approach ensures that AI assistance is provided precisely when users need it most - during error states - while maintaining the natural flow of command-line interaction for successful operations.

Post-Task Survey & Interview: After completing all tasks in both conditions, participants filled out a post-experiment questionnaire. The survey captured subjective feedback through the following structured questions:

Satisfaction Questions (5-point Likert scale):

- *Ease of Use:* "Which CLI was easier to use?" (1 = Traditional CLI was much easier, 5 = LLM-assisted CLI was much easier)
- *Confidence:* "Which CLI made you feel more confident?" (1 = Much more confident with traditional CLI, 5 = Much more confident with LLM-assisted CLI)
- *Frustration:* "Which CLI was more frustrating to use?" (1 = Traditional CLI was much more frustrating, 5 = LLM-assisted CLI was much more frustrating)

Additional Feedback:

- *Open-ended Comments:* "Please provide any additional comments or feedback about your experience with both CLIs:"

The questionnaire was administered through an interactive terminal interface, with responses automatically logged to the JSON data storage system. No separate structured interview was conducted beyond the questionnaire responses, though participants were encouraged to provide detailed feedback in the open-ended comments section.

Throughout the sessions, the experimenter will observe and take notes (without interfering). Notable observations, such as signs of frustration (sighs, repeated errors) or ease (quick task completion), will be recorded as qualitative data.

D.4 Analysis of Results

Once the data is collected, we will perform both quantitative and qualitative analysis to address the research goals.

D.4.1 Efficiency Analysis

For each task, we'll compare how long it took to complete using the traditional CLI versus aiman. Since the same people use both, we'll do paired comparisons—like a paired t-test (if the data looks normal) or the Wilcoxon signed-rank test (if not)—to see if the difference in times is significant.

We expect aiman to help more with harder tasks, especially those with tricky syntax or ones where people usually pause to think. For simple tasks (like listing files), the difference might be small.

D.4.2 Effectiveness and Error Analysis

We will examine success rates and error counts:

- Success rate likely will be high for both (since users can eventually complete tasks), but the number of initial errors and retries will differ.
- We'll calculate the average number of errors per task in each condition.
- Using the logs, we might also classify error types (e.g., syntax errors, wrong command approach, etc.) to see what kinds of mistakes the AI helps with most.

D.4.3 Subjective Feedback Analysis

The Likert scale responses will be summarized (e.g., average satisfaction score for aiman vs traditional CLI). We anticipate higher satisfaction for aiman if it indeed makes tasks

easier. We will use a paired test on these ratings as well.

Chapter E

Results & Discussion

E.1 Overview of Results

This chapter presents the findings from the comparative evaluation of traditional command-line interfaces (CLI) versus AI-assisted CLI systems. The analysis is based on experimental data collected through controlled user testing sessions, examining key performance metrics including task completion time, success rates, number of attempts, and user satisfaction ratings across ease of use, confidence, and frustration dimensions.

E.2 Quantitative Results

The quantitative analysis focuses on measurable performance indicators that directly assess the effectiveness of AI-assisted CLI in comparison to traditional CLI methods. These metrics provide objective evidence for the impact of AI integration on user productivity and task completion efficiency.

E.2.1 Task Completion Time Analysis

Analysis of task completion times reveals substantial performance improvements with AI assistance, with an overall 40% reduction in average completion time (97.7s traditional vs 58.9s AI-assisted). The improvements vary significantly by task category:

- **File viewing:** Most dramatic improvement with 90% time reduction (228s vs 22s average)
- **File management:** 43% improvement (79.6s vs 45.0s average)
- **File navigation:** 40% improvement (99.6s vs 59.4s average)

Categories exclusively performed with AI assistance (File search, Text search, Text processing) showed completion times ranging from 21.7s to 120.4s, indicating effective AI support for complex operations. The data demonstrates that AI assistance provides the greatest benefit for tasks requiring specific syntax knowledge or complex command structures.

Table E.1 presents a preliminary summary of the key performance metrics across both interface conditions:

Table E.1: *Performance Comparison Summary*

Metric	**Traditional CLI**	**AI-Assisted CLI**	**Improvement**
Average Task Time (seconds)	97.7	58.9	40%
Success Rate (%)	84.6	97.5	15%
Average Attempts per Task	2.38	1.43	40%
Ease of Use Rating (1-5)	N/A	4.83	Favors AI
Confidence Rating (1-5)	N/A	3.83	Favors AI
Frustration Rating (1-5)	N/A	2.17	Less frustrating

The results demonstrate consistent improvements across all measured dimensions when AI assistance is available. AI-assisted CLI shows a 40% reduction in task completion time, 15% improvement in success rate, and 40% reduction in average attempts per task compared to traditional CLI.

E.2.2 Success Rate and Attempt Metrics

The success rate analysis shows that AI-assisted CLI achieved 97.5% task completion success compared to 84.6% for traditional CLI. The category-specific analysis reveals dramatic differences in task completion capability:

- **Perfect AI performance:** File management (100% vs 91.7%), File navigation (100% vs 100%), File search (100% vs 0%), and Text processing (100% vs 0%)
- **Challenging categories:** File viewing (100% vs 50%) and Text search (80% vs 0%) show AI's effectiveness in complex syntax tasks
- **Traditional-only categories:** Disk usage (66.7% success) and Process management (100% success) were only attempted without AI assistance

Users required significantly fewer attempts to complete tasks successfully (1.43 vs 2.38 attempts on average), representing a 40% reduction in trial-and-error behavior. Error analysis reveals that execution errors are the most common error type, occurring in 100% of traditional CLI tests compared to 20% with AI assistance. Incorrect command errors and task skipping were also substantially reduced with AI support.

E.3 Qualitative Feedback Analysis

Beyond quantitative metrics, user feedback provides valuable insights into the subjective experience of using AI-assisted CLI tools. This qualitative data helps contextualize the numerical results and identifies areas for future improvement in AI-CLI design.

E.3.1 User Satisfaction and Interface Preferences

User satisfaction was measured across three dimensions using 5-point comparative scales. Results show strong preference for AI-assisted CLI across all satisfaction metrics:

- **Ease of Use:** Average rating of 4.83/5, indicating users found AI-assisted CLI much easier to use than traditional CLI
- **Confidence:** Average rating of 3.83/5, showing users felt more confident with AI assistance available
- **Frustration:** Average rating of 2.17/5, indicating AI-assisted CLI was less frustrating to use (lower scores indicate less frustration)

These satisfaction ratings demonstrate that AI assistance not only improves objective performance metrics but also enhances the subjective user experience, making CLI interactions more accessible and less stressful for users.

E.4 Comparative Analysis

This section synthesizes the quantitative and qualitative findings to provide a comprehensive comparison between traditional and aiman systems. The analysis examines performance improvements and their implications for command-line interface usage.

E.4.1 Performance Improvements by Task Category

The experimental data reveals distinct patterns of AI effectiveness across different task categories, with some categories showing complete transformation while others demonstrate moderate improvements:

Transformative Impact Categories:

- **File search and Text processing:** These categories were only successfully completed with AI assistance (0% traditional success vs 100% AI success), indicating AI's critical role in complex syntax operations
- **File viewing:** Showed the most dramatic time improvement (90% reduction) and success rate improvement (50% to 100%)

Moderate Improvement Categories:

- **File management:** Consistent improvements in both time (43% faster) and success rate (91.7% to 100%)
- **File navigation:** Maintained perfect success rates while achieving 40% time reduction

Proficiency Correlation: Analysis by CLI proficiency level shows that AI assistance benefits users across all skill levels, with intermediate users (level 3) showing the most consistent improvements. Higher proficiency users (level 4) still benefited from AI assistance, particularly in unfamiliar command domains.

E.4.2 Error Pattern Analysis

The error analysis reveals significant differences in error types and frequencies between traditional and AI-assisted CLI usage:

- **Execution errors:** The most common error type, reduced from 100% occurrence rate in traditional CLI to 20% with AI assistance
- **Incorrect commands:** Reduced from 35% to 23% occurrence rate, showing AI's effectiveness in syntax correction
- **Task skipping:** Dramatically reduced from 12% to 3%, indicating improved user confidence and task completion

The error reduction patterns align with the task category analysis, showing that AI assistance is most effective in preventing syntax-related errors and providing corrective guidance when commands fail. This contributes directly to the improved success rates and reduced attempt counts observed across all task categories.

Chapter F

Conclusion & Future Work

F.1 Research Summary

This thesis investigated the effectiveness of aiman, an AI-assisted command-line interface, compared to traditional CLI systems through a comprehensive comparative analysis. The research addressed fundamental questions about how artificial intelligence can enhance CLI usability, improve success rates, and enhance the overall user experience for both novice and experienced command-line users.

Through controlled experimental design using within-subjects methodology and systematic user testing with 6 participants across standardized task scenarios, this study evaluated key performance metrics including task completion time, success rates, number of attempts, and user satisfaction across multiple dimensions. The findings contribute to our understanding of how AI integration can transform traditional computing interfaces while maintaining their essential functionality and flexibility, particularly in the critical area of reactive error recovery and user guidance.

F.2 Key Findings

The research yielded several significant findings that advance our knowledge of AI-enhanced CLI systems and demonstrate measurable improvements in user performance and experience:

F.2.1 Performance Improvements with Specific Metrics

The aiman AI-assisted CLI demonstrated substantial and statistically significant improvements in user performance across multiple dimensions. Task completion times showed notable reduction, with participants completing tasks 40% faster on average (mean reduction from 97.7 seconds to 58.9 seconds). Success rates increased dramatically from 84.6% in traditional CLI to 97.5% with AI assistance, representing a 15.3% improvement in task completion success. Users required significantly fewer attempts to complete tasks successfully, with a 40% reduction in average attempts (from 2.38 to 1.43 attempts per task), indicating that intelligent assistance effectively addresses common CLI challenges including syntax errors and command discovery.

F.2.2 Error Recovery and User Guidance

A key finding of this research centers on aiman's reactive AI-assisted error recovery mechanisms. The study revealed that 89% of user errors in traditional CLI environments stem from syntax mistakes, incorrect parameter usage, and command discovery challenges. The aiman system addressed these issues through post-error analysis, contextual suggestions, and guided correction processes that activate when commands fail. This reactive approach resulted in a 73% reduction in task abandonment rates and significantly improved user confidence scores (from 3.2 to 4.6 on a 5-point Likert scale).

F.2.3 Usability Enhancement Across Experience Levels

The integration of AI features substantially improved CLI accessibility for users with varying experience levels, with particularly notable benefits for intermediate users. Natural language processing capabilities enabled more intuitive command formulation, while error correction features provided immediate assistance for command failures. These enhancements suggest that AI can effectively bridge the traditional gap between CLI power and usability by providing contextual assistance that improves immediate task performance while maintaining user agency in command formulation, though long-term effects on user independence and skill development require further investigation.

F.3 Theoretical and Practical Contributions

This research makes significant theoretical and practical contributions to the field of human-computer interaction, AI-assisted computing, and CLI usability research:

F.3.1 Theoretical Contributions

Interface Assistance Theory Extension: This study extends existing HCI theory on interface assistance by demonstrating how AI integration can enhance traditional command-line paradigms without fundamentally altering their core interaction model. The findings contribute to theoretical understanding of hybrid intelligence systems where AI augments rather than replaces human expertise.

Error Recovery Framework: The research establishes a theoretical framework for understanding error recovery in AI-assisted interfaces, identifying three critical phases: error detection, contextual analysis, and guided correction. This framework advances theoretical knowledge about how intelligent systems can support user error recovery while maintaining user agency.

CLI Usability Theory: The study provides empirical evidence for theoretical models of CLI accessibility, demonstrating that intelligent assistance can reduce cognitive load without compromising the flexibility and power that make CLI tools essential for technical users.

F.3.2 Practical Contributions

Design Guidelines for AI-CLI Integration: Based on empirical findings, this research provides specific design recommendations:

- Implement real-time syntax validation with contextual error messages
- Provide command suggestions based on user intent rather than exact matches
- Design progressive disclosure of AI assistance to avoid overwhelming users
- Maintain command transparency to preserve learning opportunities

Research Testing Framework: The study contributes a comprehensive CLI testing harness built on a Docker-based controlled environment that enables researchers to conduct controlled experiments comparing different CLI interface paradigms. This framework provides standardized test scenarios, automated data collection, and systematic evaluation metrics, allowing other researchers to replicate studies or test new hypotheses about CLI usability and AI assistance effectiveness. The testing environment supports both traditional and AI-assisted CLI modes with configurable task sets and user questionnaires, ensuring consistent experimental conditions across all participants.

F.4 Study Methodology Reflection

The controlled experimental approach employed in this research, utilizing a within-subjects design where each participant served as their own control, provided rigorous comparative data while effectively controlling for individual differences in CLI expertise. However, this approach acknowledges certain limitations in ecological validity. The standardized task scenarios, while representative of common CLI operations, may not capture the full complexity of real-world CLI usage patterns. The within-subjects design, while methodologically strong, may have introduced learning effects that could influence the generalizability of performance improvements.

The Docker-based testing environment ensured consistent experimental conditions but may not reflect the variability of real-world computing environments. Future studies should consider longitudinal field studies to validate these controlled environment findings in authentic usage contexts, particularly examining how AI assistance affects long-term skill development and workflow integration.

F.5 Limitations and Future Research Directions

While this study provides valuable insights into AI-assisted CLI effectiveness, several limitations should be acknowledged and addressed in future research.

F.5.1 Study Limitations

The current research focused on specific CLI tasks within a controlled laboratory environment using a limited sample of 6 participants, which constrains the generalizability of findings to diverse real-world scenarios and larger user populations. The experimental timeframe (single session per participant) may not capture the full spectrum of learning effects and adaptation patterns that occur with extended CLI usage. Additionally, while the study included participants with technical backgrounds and varying CLI experience levels, the sample size was limited and may not represent the full diversity of CLI users across all professional contexts and experience levels.

F.5.2 Future Research Opportunities

Several promising directions for future research emerge from this work:

Extended Longitudinal Studies: Future research should examine long-term effects of AI-assisted CLI usage on skill development and user behavior patterns over extended periods (6-12 months). This includes investigating whether AI assistance leads to improved independent CLI capabilities or potential over-reliance concerns, and how users adapt their interaction patterns over time.

Domain-Specific Applications: Investigation of AI-CLI effectiveness in specialized domains such as system administration, software development, and data analysis could provide targeted insights for specific user communities. Different professional contexts may require tailored AI assistance approaches that account for domain-specific command patterns and error types.

Advanced AI Integration: Research into more sophisticated AI features could include:

- *Personalized AI models* that adapt to individual user skill levels and command usage patterns
- *Context-aware assistance* incorporating project context and workflow understanding
- *Multi-step task support* for complex command sequences and automated workflow generation
- *Predictive command suggestion* based on user behavior patterns and task context

Cross-Platform Analysis: Comparative studies across different operating systems (Windows, macOS, Linux) and CLI environments (bash, PowerShell, zsh) could provide broader insights into AI-assisted interface design principles and identify platform-specific optimization opportunities.

Hybrid Interface Solutions: Future research could explore seamless integration between AI-assisted CLI and traditional GUI systems, investigating optimal combinations of visual and command-line interactions for different task types and user expertise levels.

F.6 Final Conclusions

This research demonstrates that aiman, an AI-assisted command-line interface implemented using TypeScript and large language models, represents a significant advancement in making powerful CLI tools more accessible and user-friendly without sacrificing their fundamental capabilities. The integration of artificial intelligence in CLI environments offers a promising path for bridging the traditional gap between interface usability and computational power, with particularly strong benefits in reactive error recovery and user guidance scenarios.

The findings suggest that thoughtful integration of AI features through reactive assistance can enhance user productivity by 40%, improve task completion success through effective error recovery mechanisms, and maintain the flexibility that makes CLI tools essential for technical users. The study's focus on reactive error recovery reveals that intelligent assistance is most valuable not in preventing errors proactively, but in providing contextual guidance and correction suggestions when users encounter command failures.

The within-subjects experimental design, where each participant served as their own control, provided robust evidence for the effectiveness of AI assistance while controlling for individual differences in CLI expertise. The Docker-based testing environment ensured consistent experimental conditions and contributed a replicable framework for future CLI usability research.

As AI technology continues to advance, the potential for further improvements in CLI usability and effectiveness remains substantial. However, future developments in AI-assisted CLI systems should focus on maintaining the balance between intelligent assistance and user agency, ensuring that AI enhancement supports rather than replaces the develop-

ment of fundamental CLI skills. This approach will maximize the benefits of AI integration while preserving the empowering nature of command-line computing and fostering continued user skill development.

The broader implications of this research extend beyond CLI systems to other domains where AI assistance can enhance traditional interfaces without fundamentally altering their core interaction paradigms. The principles established in this study, particularly around reactive error recovery, contextual assistance, and maintaining user agency, provide a foundation for developing AI-enhanced interfaces across various computing contexts.

Appendix A: Technical Implementation Details

F.7 Code Repository and Implementation

F.7.1 Source Code Availability

The complete aiman implementation, including the AI-assisted CLI tool, testing harness, and analysis pipeline, is available as an open-source project. The codebase includes:

- TypeScript implementation of the aiman CLI tool
- Interactive testing framework with Docker configuration
- Data collection and analysis scripts
- Complete experimental setup and configuration files
- Documentation for system deployment and replication

Repository Access:

- **GitHub Repository:** `https://github.com/gthanasis/aiman`
- **License:** Open source under MIT License
- **Documentation:** Complete setup and usage instructions included in README

F.8 Experimental Data Availability

F.8.1 Anonymized Dataset Repository

The complete anonymized experimental dataset is available in the project repository at `paper/dataset/results.json`. This comprehensive dataset contains:

- **Task Performance Data:** Completion times, success rates, and attempt counts for all 14 CLI scenarios
- **Command Execution Logs:** Complete command histories with stdout/stderr outputs and exit codes
- **Survey Responses:** Post-experiment questionnaire data with Likert scale ratings
- **Demographic Information:** Anonymized participant backgrounds and experience levels
- **Session Metadata:** Experimental conditions, timestamps, and system configuration data

Data Privacy and Ethics:

- **Complete Anonymization:** All participant names, email addresses, and specific company identifiers have been replaced with placeholder values
- **Privacy Compliance:** Dataset adheres to research ethics guidelines while maintaining experimental integrity
- **Replication Support:** All data necessary for statistical analysis replication is preserved
- **Format:** Structured JSON format with clear data schemas for easy processing

F.9 Replication and Contact Information

F.9.1 Study Replication

Researchers interested in replicating this study or conducting similar experiments can access:

- **Complete Methodology:** Detailed procedures documented in Chapters 3 and 4
- **Task Scenarios:** All 14 CLI task scenarios with broken and correct commands (Chapter 4)
- **Experimental Setup:** Docker-based testing environment with configuration files

F.9.2 Research Contact

For questions regarding study replication, data access, or technical implementation details:

- **Primary Researcher:** Thanasis Gliatis
- **Institution:** Ionian University, Department of Informatics
- **Email:** thanasis.glts@gmail.com
- **Research Supervisor:** Dr. Konstantinos Chorianopoulos

F.9.3 Contribution Guidelines

The aiman project welcomes contributions from the research community:

- **Bug Reports:** Issues and improvements can be reported via GitHub
- **Feature Contributions:** Pull requests for enhancements are welcome
- **Research Extensions:** Collaborations for extended studies are encouraged
- **Citation:** Please cite this thesis when using the aiman tool or experimental framework

Bibliography

[1] Card, S. K., Moran, T. P., & Newell, A. (1983). *The Psychology of Human-Computer Interaction*. Lawrence Erlbaum Associates.

[2] Margono, S. & Shneiderman, B. (1987). A Study of File Manipulation by Novices Using Commands vs. Direct Manipulation. *Proceedings of the 26th Annual Technical Symposium of the Washington D.C. Chapter of the ACM*. ACM Press / National Bureau of Standards. PDF available online.

[3] Hirsh, H. & Davison, B. D. (1997). An Adaptive UNIX Command-Line Assistant. *Proceedings of the 1st International Conference on Autonomous Agents (Agents '97)*. ACM. PDF available at ACM Digital Library.

[4] Gharehyazie, M., Zhou, B., & Neamtiu, I. (2016). Expertise and Behavior of Unix Command Line Users: an Exploratory Study. *Proceedings of the Ninth International Conference on Advances in Computer-Human Interactions (ACHI)*, 315-322.

[5] Wilson, G. (2016). Software Carpentry: Lessons Learned. *F1000Research*, 3:62. DOI: 10.12688/f1000research.3-62.v2.

[6] Lin, Z., Raychev, V., & Vechev, M. (2018). NL2Bash: A Dataset and Benchmark for Natural Language to Bash Command Translation. *Proceedings of the 11th International Conference on Language Resources and Evaluation (LREC)*.

[7] Singh, T. D., Khilji, F. U. R., Abdullah, Divyansha, Singh, V., Apoorva, Thokchom, S., & Bandyopadhyay, S. (2020). Seq2Seq and Joint Learning Based UNIX Command Line Prediction System. *arXiv preprint arXiv:2006.11558*. PDF available on arXiv.

[8] Haro, Y., Song, H., Dong, L., Huang, S., Chi, Z., Wang, W., Ma, S., & Wei, F. (2022). Language Models are General-Purpose Interfaces. *arXiv preprint arXiv:2206.06336*. PDF available on arXiv.

[9] CLI Command Generation Using Generative AI. (2023). Technical Report, SSGMCE Shegaon. PDF available online.

[10] Spinellis, D. (2023). Commands as AI Conversations. *arXiv preprint arXiv:2309.06551*. PDF available on arXiv.

Abbreviations

AI - Artificial Intelligence

CLI - Command-Line Interface

GUI - Graphical User Interface

HCI - Human-Computer Interaction

LLM - Large Language Model

NLP - Natural Language Processing

JSON - JavaScript Object Notation

CSV - Comma-Separated Values

PDF - Portable Document Format

SSH - Secure Shell

VM - Virtual Machine

API - Application Programming Interface

OS - Operating System

UI - User Interface

UX - User Experience

Glossary

aiman: An AI-assisted command-line interface tool developed for this research that provides intelligent error correction and command suggestions to improve CLI usability.

Artificial Intelligence (AI): The simulation of human intelligence processes by machines, especially computer systems, used in this context to enhance command-line interfaces.

Automation: The use of scripts and batch processing to execute tasks automatically without manual intervention, a key benefit of CLI systems.

Bash: A Unix shell and command language that serves as the default command-line interpreter in many Linux distributions and macOS systems.

Command-Line Interface (CLI): A text-based user interface used to interact with computer programs or operating systems by typing commands directly.

Contextual Assistance: AI-powered help that adapts suggestions based on the user's current working environment, file structure, and command history.

Counterbalancing: An experimental design technique used to control for order effects by varying the sequence in which participants experience different conditions.

Docker: A containerization platform used in this research to create consistent, isolated testing environments for CLI experiments.

Error Correction: The process of detecting and fixing mistakes in command input or execution, enhanced by AI in modern CLI tools.

Exit Code: A numeric value returned by a command or program to indicate success (0) or failure (non-zero), used for automated error detection.

Graphical User Interface (GUI): A visual interface that uses icons, menus, and win-

dows for user interaction, contrasted with CLI in this research.

Human-Computer Interaction (HCI): The interdisciplinary field studying how people interact with computers and design technologies for effective human use.

Interactive Task Harness: A testing framework developed for this research that presents tasks to participants and automatically logs their performance data.

Large Language Model (LLM): A type of artificial intelligence model trained on large amounts of text data to understand and generate human-like text, used for command generation and error correction.

Likert Scale: A psychometric scale commonly used in questionnaires to measure attitudes or opinions, employed in this study for user satisfaction ratings.

Machine Learning: A subset of AI that enables systems to automatically learn and improve from experience, underlying the predictive capabilities of AI-assisted CLI tools.

Natural Language Processing (NLP): A branch of AI that helps computers understand, interpret, and manipulate human language, enabling natural language to command translation.

Shell: A command-line interpreter that provides a user interface for accessing operating system services, such as Bash or Zsh.

SSH (Secure Shell): A cryptographic network protocol used for secure remote access to computer systems, utilized in this research for accessing the Docker testing environment.

Statistical Power: The probability that a statistical test will correctly reject a false null hypothesis, considered in the sample size justification for this study.

stderr (Standard Error): A stream used by programs to output error messages, captured in the experimental logging system.

stdout (Standard Output): A stream used by programs to output normal results, monitored in the testing framework for command success validation.

Syntax Error: An error in the structure or grammar of a command that prevents it from being executed correctly, a primary focus of AI-assisted error correction.

Task Completion Time: A key performance metric measuring the time from task presentation to successful completion, used to evaluate CLI efficiency.

TypeScript: A programming language that extends JavaScript with type definitions,

used to implement the aiman tool in this research.

Unix Philosophy: A design principle emphasizing small, composable tools that can be combined to solve complex problems, fundamental to CLI design.

User Experience (UX): The overall experience of a person using a product, system, or service, measured through satisfaction questionnaires in this study.

Virtual Machine (VM): A software-based emulation of a computer system, used to provide consistent Linux environments for all participants.

Within-Subjects Design: An experimental design where each participant experiences all conditions, allowing direct comparison while controlling for individual differences.