In The Termi-Know: User-Friendly Terminals Through Informative Autocomplete

Samuel Murphy
Mentored by Mariam Adegbuyi, Christian de Poortere, and Hannah Newberry
Johns Hopkins University Applied Physics Laboratory ASPIRE program

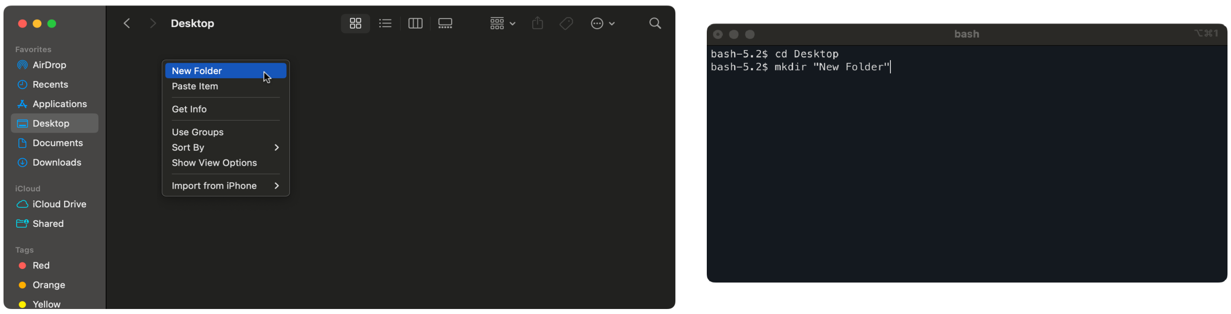
April 2025

1 Introduction

The terminal is a simple, yet powerful computing tool that has held relevance from the early history of consumer electronics to the modern day. Originating from early computers which used typewriters and alphanumeric displays to produce an output, the terminal is a system for communicating with programs through a stream of text. Programs which use the terminal receive characters, most often through a user typing on a keyboard, and respond by writing characters to an output stream. This mode of interaction is known as a command-line interface (CLI). Shells are an important type of terminal program which provide a command-line interface that allows users to interact with a computer's operating system and launch other programs.

As computer display technology has advanced beyond character-based graphics, terminal applications have been largely superseded by applications which use a graphical user interface (GUI) to display information and receive input. Most everyday computer users only interact with software that has a graphical interface, as these apps are typically simpler to learn and utilize. However, terminals remain an integral part of more advanced computer use because of several advantages they hold over GUI-based software. The simple nature of user input in command-line interfaces means that terminal programs can often support more options and use cases than their graphical counterparts. For example, the open-source file format conversion software Ffmpeg is widely used for image and video processing due to the large range of formats it supports, but it lacks a graphical interface with the same capabilities as its command-line interface. Many software developers prefer making terminal applications because they are easier to port between operating systems, require less focus on the implementation of a user interface, and consume less system resources while running. Terminals also allow knowledgeable users to get tasks done more quickly, as they tend to respond to input in less time than graphics-based applications and support scripting features which make it possible to automate repetitive jobs.

The main drawback to terminal applications is that they are often more difficult for new users to learn. The terminal's simple, text-based display is often less capable of communicating what actions are available at any given time compared to a graphical display. As a result, terminal users must consult documentation in order to learn how to perform tasks that a graphical interface may make more obvious. Terminal software frequently



```
bash-5.2$ cd Desktop
bash-5.2$ mkdir "New Folder"
```

Figure 1: On the macOS operating system, users can utilize the graphical interface of the Finder application (shown left) to create a folder by opening a context menu. The same action can be performed in the Bash shell (shown right) with the mkdir command, short for "make directory."

uses acronyms and abbreviations when displaying information or requesting user input for the sake of brevity. For example, in the Bash shell, the command ls displays a list of files in the current directory. Without external knowledge, it can be hard to determine the function of these commands through their text representation.

While it is expected that an interface used mostly by experts is more complicated to learn, everyday computer users could also benefit from terminal software if these accessibility problems were addressed. Open-source applications such as those of the GNU project are some of the most likely to adopt command-line interfaces because of the ease of development and deployment they provide. Making the terminal easier to use would increase the reach and utility of these tools. Terminals are also useful for performing uncommon tasks that a graphical interface may not account for. One example is the bulk editing of photos or videos, which most GUI editors have limited support for due to their focus on editing a single file at a time. Without specialized software, command-line scripting is often most effective way to solve these problems, but the complex presentation of CLIs means that it can be frustrating and intimidating for inexperienced users to utilize such tools. Finally, making terminals easier to use would improve the accessibility of jobs like computer science and cybersecurity which frequently rely on CLI-only tools. The goal of this research project is to design a system for increasing the ease-of-use of terminals and examine its effectiveness.

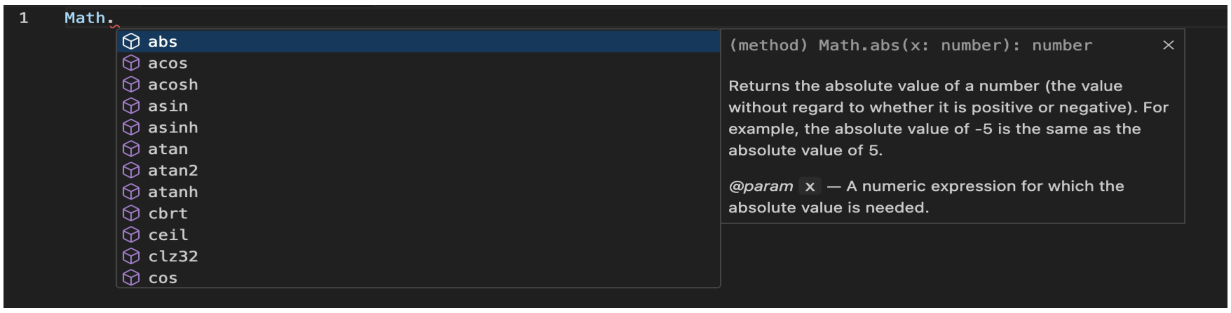
2 Code Completion

Computer programmers write code using text editors that share functional similarities with shells and other command-line interfaces. While programming languages and CLIs are used to achieve different sets of goals, both receive a text input from the user which is separated into keywords that specify functionality or program flow. As a result, programming languages suffer from usability issues that are similar to those of CLIs. Computer programs interact with other systems and applications through a collection of functions known as an app-program interface (API). Like command-line tools, programmers have to consult the documentation of an API to learn what these functions do, which can make it challenging and time-consuming to write new code or trace the behavior of existing code.

Many applications used for writing and testing code, also known as independent development environments (IDEs), provide systems to alleviate the difficulty of keeping track of API documentation. Generally, these systems automatically parse the documentation of an API in order to provide relevant information alongside program code. One of the most widely-available features is code completion, a system which uses information about the program to suggest potential completions as the user types. Many independent development environments show relevant excerpts from documentation next to completion choices, allowing users to figure out what options are available to them without leaving the programming environment. The same information can also be shown when hovering or selecting text, which is useful for figuring out the functionality of existing sections of code. For the purposes of this article, this system, consisting of a combination of code completion and documentation-based synopsis, will be referred to as informative autocomplete.

A study by Jiang Shaokang and Michael Coblenz evaluated 32 developers experienced with the Java programming language using the API for Gmail and found that participants who used autocomplete gained an improved understanding of the API than participants who had to rely on external documentation (Jiang & Coblenz, 2024). Because of the similarities between terminals and programming environments, providing informative autocomplete features to a terminal could lead to a similar increase in understanding, making terminals easier for inexperienced users to learn. The design portion of this study is focused on the implementation of IDE-style informative autocomplete into a terminal.

Some shells already provide autocomplete features. However, these have limitations that prevent them from being as effective as IDE-style autocomplete at improving learnability. Bash, the default shell on most Linux-based operating systems, enters the most relevant completion option when the user presses the Tab key. Pressing Tab a second time shows all available completion options. Because completions only show when the user requests them, this system is less helpful for showing new users what options are available to them as they type, and Bash does not provide any description of the completion options that it suggests. Fish, an alternative shell which targets usability and accessibility, shows inline completions as the user types and brings up a detailed list of completions when the user presses Tab. This menu includes brief descriptions of its suggestions which are obtained by parsing documentation in a manner similar to IDEs; however, because Fish is a terminal application, it can only use the terminal's text-based graphics to display these completions, meaning that the full description is often cut off and unable to be read in the suggestion window. Any shell-based autocomplete system also suffers from a lack of compatibility. Shells have many functional differences that might impact a user's decision to choose one; for example, scripting and customization settings are unable to be shared



```
Math.
abs
acos
acosh
asin
asinh
atan
atan2
atanh
cbrt
ceil
clz32
cos
```

(method) Math.abs(x: number): number

Returns the absolute value of a number (the value without regard to whether it is positive or negative). For example, the absolute value of -5 is the same as the absolute value of 5.

@param x — A numeric expression for which the absolute value is needed.

Figure 2: IntelliSense, a system used in Microsoft's Visual Studio Code development environment, is one example of informative autocomplete. In this example, IntelliSense shows the user all functions available in the Math namespace in the JavaScript programming language. A synopsis (shown right) of the selected option is also displayed.

across most shells. This means that not all users would be able to adopt an effective autocomplete system implemented at the shell level.

3 Software Architecture

Since the text-based displays that the terminal system was originally built for are no longer commonly used, most interaction with terminal software occurs through terminal emulators, applications which display a terminal output through a graphical user interface. The terminal emulator takes the role otherwise held by a printer or alphanumeric display, communicating through a process called a pseudoterminal which manages an input and output stream between the emulator and a terminal application. The stream consists of a series of characters, which can either be graphical characters (e.g., a letter or a number) or unprintable characters that represent instructions to perform specific tasks. The terminal emulator displays graphical characters and interprets unprintable characters that the application sends (Drevekenin, 2021). The autocomplete system developed in this paper is implemented through a terminal emulator built for the project. This was done to ensure compatibility with different shells and operating systems and to enable the use of an IDE-style graphical user interface for autocomplete information.

Development of the terminal emulator was conducted in a VirtualBox virtual machine running the Linux-based Ubuntu operating system. The source code is written in C++ and utilizes the QT framework to create a user interface. An outline of the code's structure is shown in Figure 3.

The program includes a simple terminal component which displays program output onscreen and allows the user to enter input through the keyboard. When launched, it uses Linux system calls to create a pseudoterminal and run the Bash shell program. Terminal programs create graphics by manipulating the cursor, the point at which new text is written in the terminal display. When terminals receive printable text from the running program, it is written at the location of the cursor. Unprintable text is sent by the program to instruct the terminal to perform an action or configure a setting. Unprintable text can consist of either a single unprintable character or a series of printable characters preceded by an unprintable character, known as an escape sequence. Some examples of the instructions that sequences provide include repositioning the cursor, changing the style or color of text, and filling or clearing portions of the screen. What escape sequences are available varies between terminals; the most compatible have support for a very large number of different escape sequences to maximize compatibility with terminal programs. As the terminal created for this project is only a proof-of-concept, only the sequences needed to run most common Linux terminal programs are supported.

The autocomplete portion of the program reads information from the terminal emulator to provide suggestions to the user which are obtained by parsing documentation. Many Linux programs that use command-line interfaces are documented through manual pages, a system of standardized reference material. Manual pages are designed to be human-readable, but most follow a consistent format based on the Roff markdown system, enabling the use of automatic parsing to extract information. The Fish shell's autocomplete system uses manual page parsing to provide a description for command arguments that it suggests. The terminal emulator program uses an algorithm based on the Fish implementation to provide a description for both commands and command arguments. It searches the locations where manual pages are stored on the user's system and builds a database of command documentation by parsing each one. Reading manual pages allows the system to provide completion information for a large breadth of terminal programs without the need to create new documentation, as distributing terminal programs

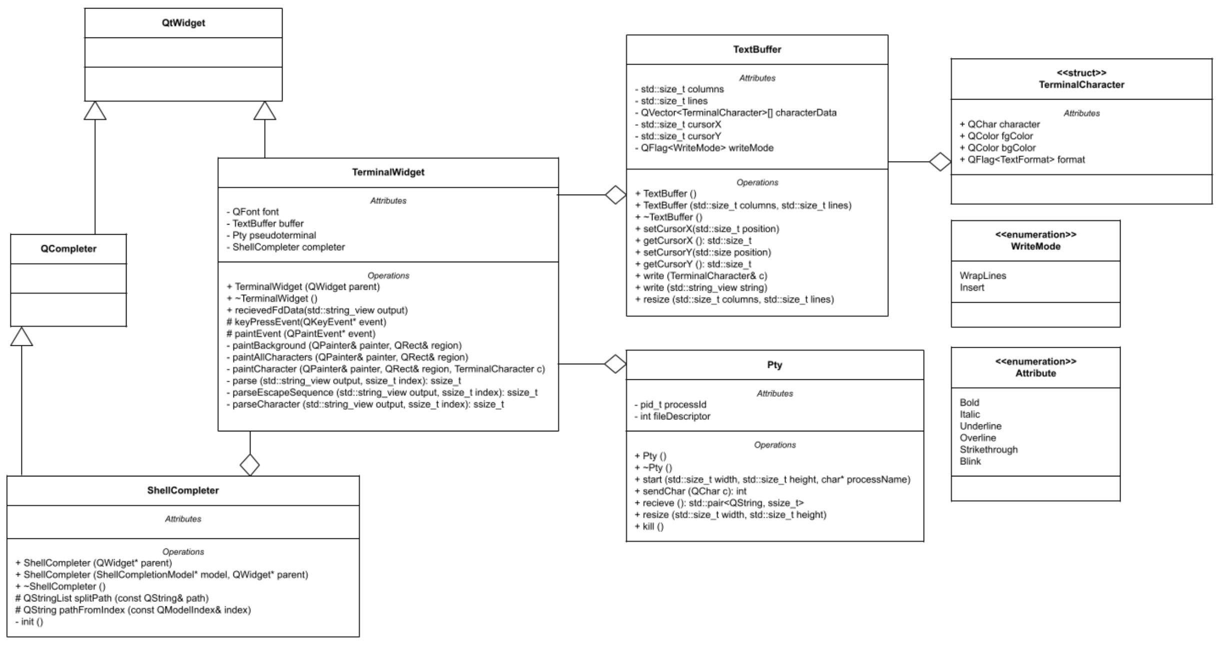

QtWidget
QCompleter
TerminalWidget
Attributes
- QFont font
- TextBuffer buffer
- Pty pseudoterminal
- ShellCompleter completer
Operations
+ TerminalWidget (QWidget parent)
+ ~TerminalWidget ()
+ recievedFdData(std::string_view output)
# keyPressEvent(QKeyEvent* event)
# paintEvent (QPaintEvent* event)
- paintBackground (QPainter& painter, QRect& region)
- paintAllCharacters (QPainter& painter, QRect& region)
- paintCharacter (QPainter& painter, QRect& region, TerminalCharacter c)
- parse (std::string_view output, ssize_t index): ssize_t
- parseEscapeSequence (std::string_view output, ssize_t index): ssize_t
- parseCharacter (std::string_view output, ssize_t index): ssize_t
ShellCompleter
Attributes
Operations
+ ShellCompleter (QWidget* parent)
+ ShellCompleter (ShellCompletionModel* model, QWidget* parent)
+ ~ShellCompleter ()
# QStringList splitPath (const QString& path)
# QString pathFromIndex (const QModelIndex& index)
- init ()
TextBuffer
Attributes
- std::size_t columns
- std::size_t lines
- QVector<TerminalCharacter>[] characterData
- std::size_t cursorX
- std::size_t cursorY
- QFlag<WriteMode> writeMode
Operations
+ TextBuffer ()
+ TextBuffer (std::size_t columns, std::size_t lines)
+ ~TextBuffer ()
+ setCursorX(std::size_t position)
+ getCursorX (): std::size_t
+ setCursorY(std::size position)
+ getCursorY (): std::size_t
+ write (TerminalCharacter& c)
+ write (std::string_view string)
+ resize (std::size_t columns, std::size_t lines)
<<struct>>
TerminalCharacter
Attributes
+ QChar character
+ QColor fgColor
+ QColor bgColor
+ QFlag<TextFormat> format
<<enumeration>>
WriteMode
WrapLines
Insert
Pty
Attributes
- pid_t processId
- int fileDescriptor
Operations
+ Pty ()
+ ~Pty ()
+ start (std::size_t width, std::size_t height, char* processName)
+ sendChar (QChar c): int
+ recieve (): std::pair<QString, ssize_t>
+ resize (std::size_t width, std::size_t height)
+ kill ()
<<enumeration>>
Attribute
Bold
Italic
Underline
Overline
Strikethrough
Blink

Figure 3: UML diagram showing the class structure of the terminal emulator program. Some classes and class members are omitted for brevity.

alongside a manual page is an extremely common practice in Linux software development. However, because manual pages are designed to be read by humans and not automatically indexed, some pages have formatting inconsistencies that prevent them from being parsed for command information. Once completion data has been collected, it is stored in a tree data structure where each command is a node with its argument options as child nodes.

4 Survey

We administered a survey to evaluate the created autocomplete system's effectiveness in providing easy access to the information needed to perform common tasks in the Linux terminal. Three APL staff members, including two mentors for the project, participated in an experiment consisting of a timed activity and a questionnaire. The survey was given through a MacBook Pro. Google Forms was used to host the activity and survey questions, and participants used a VirtualBox virtual machine installed on the system to interact with the terminal emulator.

In the timed activity section, participants used the terminal emulator to answer 4 questions designed to represent information gathering tasks frequently employed during everyday terminal use. Activities included searching for files that meet certain conditions, analyzing text files, and querying information about the operating system. The question form gave confirmation upon entering a correct answer. Participants were asked to leave questions blank if they could not find a solution. This section of the survey had a 15-minute time limit, but participants who experienced technical problems were allotted additional time. After finishing the timed activity, participants answered a 6-question survey consisting of 3 Likert scale questions and 3 optional free response questions. This section had no time limit.

Of the four factual questions in the timed activity section, three were answered correctly by all participants. The remaining question, which asked participants to use the ls command to find the author of a file, was answered correctly by two out of three participants. Two participants reported that this question was difficult to solve due to a visual error. The terminal emulator has a fixed character length which is shorter than the data table which ls displays, causing each row of the table to take up multiple lines and making the output confusing to read.

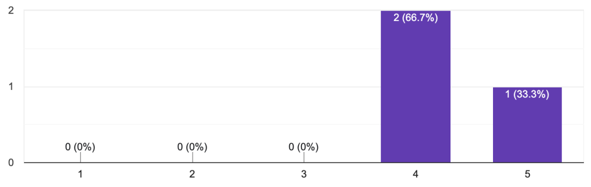


Figure 4: Participant responses to statement "The informative autocomplete system was useful for accessing information needed to answer questions in the assessment."

All three participants reported that the system helped them answer questions. Although some had existing background with the Linux terminal, they relied on autocomplete to answer most questions. One reported that the system was a convenient reference for users who know that a command or argument exists, but forget the exact format and wording used to invoke it.

In the subsequent free response question which asked what changes would improve the system in this regard, participants generally indicated changes to the user interface, such as improved text wrapping and better support for keyboard navigation of the autocomplete window.

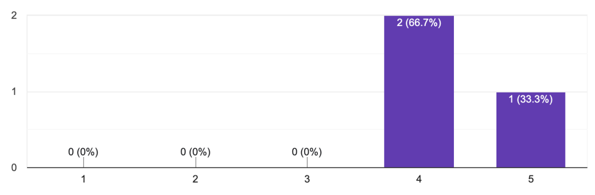


Figure 5: Participant responses to statement ”The informative autocomplete system was useful for entering commands and completing tasks faster.”

Efficiency is an important element of ease-of-use. Improvements in this area would make the terminal more convenient for both experienced and new users. Therefore, participants were asked if the autocomplete system facilitated faster task execution. All three participants agreed, implying that they experienced a perceived improvement in efficiency due to the autocomplete system. A study comparing the time to complete tasks with and without the autocomplete system would provide a factual backing to this subjective result.

In the free response attached to this question, one participant indicated that a keyboard shortcut for entering autocomplete would improve the system's ability to facilitate efficient terminal use. Another response suggested improving the ”speed of the application.” The performance speed of the program was limited for this survey as it was running in a virtual machine, resulting in delayed feedback to user input. This response indicates that poor performance has a negative effect on the speed and convenience of terminal use.

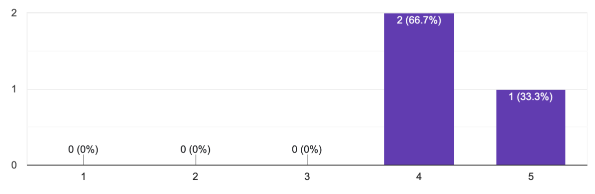


Figure 6: Participant responses to statement ”The informative autocomplete system could help inexperienced users learn the Linux terminal.”

All three participants also agreed that the system could help newer terminal users learn. Most participants had an existing knowledge level higher than that of a beginner, so their agreement indicates an inference of how the system would help others made based on their own experience. Regardless, the answers to this question indicate that informative autocomplete has a positive effect on accessibility and ease-of-use in a terminal environment.

The free-response attached to this question similarly asked what changes would improve the system's ability to facilitate learning. Answers to this question all involved visual changes, with one participant suggesting ”an interactive help directory to experience commands” and another indicating that additional use of colors in the autocomplete interface would ”vary the presentation and make it more engaging.”

5 Product

The completed application functions as a terminal emulator while providing autocomplete suggestions. As the user types in a shell, their input is read and compared with the database of manual page information to provide completion options through a pop-up graphical interface. For the proof-of-concept application created for this paper, only the Bash shell is supported, and command information is only shown while the user is typing. Increased compatibility and the adoption of more features of IDE-style informative autocomplete are potential improvements to be made in the future.

The source code of the proof-of-concept application is available for download on GitHub at https://github.com/murphsj/aspire-terminal.

6 Conclusion

This study investigated a potential strategy for improving terminal usability through the development of a terminal emulator application with informative autocomplete. The developed terminal emulator parses the manual pages associated with terminal programs to present information and suggestions to the user about the options and functionality available to them as they type. A survey was conducted to evaluate the potential benefits that an autocomplete system could have on making the terminal easier for new users to learn.

Participant feedback and accuracy in answering questions in the timed activity indicate that the proof-of-concept terminal application was successful in improving the accessibility of the terminal. Future research could improve the validity of this result by measuring the impact of informative autocomplete on users learning the terminal for the first time, and by evaluating a larger number of participants.

Future areas of improvement for the informative autocomplete system include additional features and increased compatibility. Showing information about commands and arguments that are hovered or selected with the mouse cursor would allow users to better trace the behavior of commands that have already been entered. Allowing users to jump to the portion of documentation that a command description comes from would make it easier to learn specific details or related topics that a summary may not be able to cover. To make the system more suitable for practical use, future research could focus on widening the terminal emulator's support for escape sequences or decoupling the autocomplete system from a custom terminal emulator altogether by developing it as a standalone program that integrates into other terminals.

References

Drevekenin, A. (2021, November). *Anatomy of a terminal emulator*. https://poor.dev/blog/terminal-anatomy

Jiang, S., & Coblenz, M. (2024). An analysis of the costs and benefits of autocomplete in IDEs. *Proceedings of the ACM on Software Engineering*, *1*(FSE). https://doi.org/10.1145/3660765