The Queen of All Terminals: WezTerm

Authors:

Maya Freedman (Lead) 22QB34@queensu.ca

Ella St. John (Presenter) ella.stjohn@queensu.ca

Pooria Roy (Presenter) pooria.roy@queensu.ca

Elliott Vince elliott.vince@queensu.ca

Caitlin Roach caitlin.roach@queensu.ca

Michelle Kelly 23njc3@queensu.ca

Course: CISC 322 - Software Architecture

Table of Contents:

Abstract **2**
Introduction and Overview **2**
History and Evolution 2
Derivation Process 3
Conceptual Architecture **4**
Subsystems (Components) 6
Division of Responsibilities 7
Concurrency 9
External Interfaces **10**
Data & Control Flow **10**
Name and Data Dictionaries 13
Conclusions & Lessons Learnt **14**
AI Collaboration Report **14**
Works Cited **16**

Abstract

This report examines the software architecture of WezTerm, an open source terminal emulator and multiplexer (Furlong). It looks at how the system is organized and how it evolved into the tool it is today.

We built our understanding primarily from WezTerm's official documentation and its public repository, using the commit history to trace how it grew from a simple GUI terminal into a highly configurable system (Furlong). We describe its architecture as a mostly layered design with client-server and event-driven elements, and we identify five core subsystems: the Multiplexer, Domains, Shell Integration, the Graphics Manager, and the Customization Engine ("Multiplexing"; "Shell Integration"). We then follow the data and control flow through three real use cases, including running a command, loading a custom configuration, and connecting to a remote server. A dedicated discussion of concurrency explains how WezTerm draws on ideas from earlier multiplexers while adding a modern graphics accelerated interface (Marriott; Laumann and Bormann)

Our main finding is that WezTerm's flexibility comes from treating configuration as a small program rather than fixed settings menu, which offers unusual power at the cost of requiring some scripting knowledge ("front_end"). We also note that the project's tightly connected subsystems require contributors to understand how components interact, and that its review process, which rests largely on one maintainer, can become a bottleneck (Furlong).

Introduction and Overview

WezTerm is a cross-platform terminal emulator created by British entrepreneur and open-source developer Wez Furlong, widely used by developers, server maintainers, and hobbyists. WezTerm terminal acts as a supplemental terminal user can use, as opposed to the built-in terminal to a respective operating system. Furlong had identified the feature gaps in built-in terminals, and developed WezTerm to reach a broader audience of terminal users. WezTerm is available for use on a variety of operating systems, such as Linux, Windows, macOS, and Berkeley Software Distribution (BSD) systems. Aside from the terminal being widely accessible, its unique features set it apart from other supplemental terminals; multiplexing, local and remote domains, shell integration, and dynamic user-interface options ("Features").

History and Evolution

WezTerm was first launched on GitHub in 2017. The project uses a continuous-release version control model, which automatically pushes code changes directly to production every night, rather than single-version deployment, which would release bundled versions at planned milestones. Nightly builds update continuously from the main branch, named by date, time, and git commit hash, and Furlong occasionally denotes old nightly versions as officially "stable".

These stable versions are sporadic and often years behind the nightly-updated versions, so they are typically not recommended for daily use.

Furlong made his first commit on December 7th of 2017 with the stated goal of creating "a common core that could run as both a textual terminal multiplexer and a GUI terminal emulator… where the GUI part could automatically provide a native UI around the remotely multiplexed session."("commit c53ca64", 2017). Early contributions focused on rendering text using HarfBuzz and FreeType, two open-source software libraries that work together to handle typography on modern operating systems. From there, Furlong focused on the UI, adding complex colour palettes with options to change text, cell, foreground, and background colours. It is evident from early commit history that creating a visually engaging, user-friendly tool was a priority from day one; emoji compatibility and colour customization were introduced before the window was actually connected to a working terminal ("commit ab74b4c", 2018).

2019 saw improvements including the introduction of a tab bar, bug fixes, window behaviour modifications, and updates for compatibility with different operating systems. Furlong set clear goals for WezTerm, stating that he wished for the project to be OS-independent, accessible from any operating system. Although Furlong's stated goal in 2017 was already a multiplexer, early versions worked as a plain GUI terminal. In 2020, he delivered on his original vision by adding built-in multiplexing. That year, native scrollback search was introduced; instead of just finding matching strings, WezTerm used advanced state toggling, allowing for real-time switching between regular expressions and case-insensitive strings. This would allow users to look back through their terminal sessions to find words, phrases, or tools printed previously. The project reached a major milestone in 2021 when WezTerm shifted away from standard static config (TOML) files to an entirely dynamic, hot-reloading environment scriptable via Lua, a high-performance language that allowed for the ability to turn terminal configs into real, functional programs. Advanced multiplexing and internal API capabilities were introduced in 2022, allowing for persistent, thread-safe, in-process variable storage across live configuration reloads.

As of this writing, the latest stable WezTerm release was pushed on February 3rd, 2024. The project has grown significantly from its beginnings as one Rust file in 2017; today it has amassed over 8,700 commits, 35 branches, and 1,800 total files. As a large open-source project, it is maintained by hundreds of people across the world, and anyone is free to fork the repository and submit a pull request to add a feature or fix a bug. It currently has 437 contributors and around 1,800 forks. Following its inception as one developer's passion project, it has since evolved into a cross-platform, highly configurable terminal emulator. Because its source code is freely accessible to read, modify, and redistribute, the project continues to grow and continue improving long after its first release.

Derivation Process

To begin learning about WezTerm, we started with the official documentation at wezterm.org, which provided a complete overview of the project, including details about shell integration, multiplexing, scrollback, and SSH. The documentation provided us with much of the

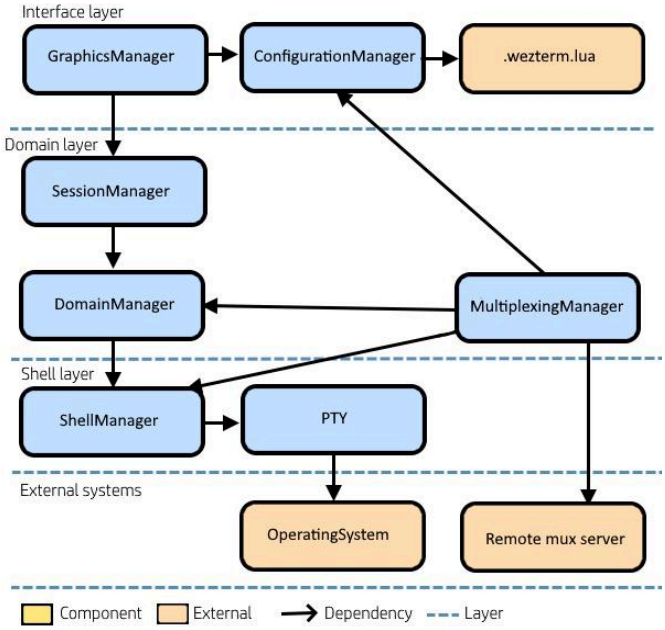
information we needed to analyze its architecture and design choices, serving as the foundation for this report.

To further understand its architecture, we looked into WezTerm's source code, which is fully accessible to read online via a GitHub repository. This allowed us to research specific components, dependencies, and libraries Furlong used. Many of these dependencies were themselves open-source, including HarfBuzz and FreeType, two libraries that work together to render text and handle typography and font styles on modern operating systems.

We also used this repository to trace WezTerm back to its very first commit in December of 2017. This allowed us to track and discuss specific architecture changes, such as the transition from TOML config files to Lua environments. Interestingly, reading through early WezTerm Rust code shows TODO and FIXME comments, demonstrating Furlong's thought process as the project grew. The repository also provided an excellent resource in the form of GitHub Discussion boards, where we could read conversations between Furlong and WezTerm contributors. WezTerm has amassed a large community of developers since its inception in 2017, and reading through discussions gave us an insight into how specific design choices were made, and how other open-source developers began to contribute.

Because WezTerm is a relatively small, non-commercial project, it does not have a large presence online. It does not have a Wikipedia page or many external resources, and much discussion about its architecture is limited to chatrooms and GitHub Discussion board posts. These constraints caused us to look deeper into the official documentation and source code to form our own understanding of how the project works, and how it has evolved over time.

Conceptual Architecture



Through our analysis, we determined that WezTerm uses the layered and client-server architectural styles, while incorporating other styles in certain parts of their software.

Figure 1: WezTerm's conceptual architecture, which uses the layered style paradigm.

Layered Style

Most of WezTerm's user interactions follow a similar flow. First, the user sends an input to a top-level component like the GraphicsManager (either typing a command, pressing custom keybinds, or clicking a GUI button), which routes the input to the relevant shell via the SessionManager, which is on the second layer. Then, that shell's manager communicates with the PTY, which interacts with external systems like the operating system or a remote multiplexing server. The final results are sent back through their original route, returning to the GraphicsManager to display in the top-level GUI.

The layered architecture style fits well for an open-source project like WezTerm. Consider a developer who wants to add a new feature to WezTerm's customization engine. Their changes would only affect the adjacent layers in the architecture, which minimizes the risk of accepting those changes into the codebase. Also, the layered style is easy to understand; restricting the data flow to adjacent layers allows developers to quickly understand how WezTerm's systems interact.

However, a strict layered style might not yield the most efficient throughput. For instance, when inputting a command into the WezTerm GUI, relying on the SessionManager to find the relevant domain might be slower than an object-oriented approach where every pane knows which domain it belongs to, which shell it can access, and so on.

Client-Server Style

Although WezTerm relies on the layered style to route its inputs and outputs, it also relies on the client-server style to support its multiplexing features. Most evidently, connecting to a remote SSH server requires sending traffic over a network. To accomplish this, the user's local WezTerm client connects to the remote SSH server to send commands and receive output from the server. WezTerm also uses the client-server style in cases where it does not communicate with external networks. For instance, its ability to detach and reattach to domains depends on a multiplexer server to host the active domains, so that those domains do not depend on the WezTerm GUI. Similarly, a user can create multiplexing servers, such as a Unix domain, on the same machine as their WezTerm client. In this case, even though WezTerm is not sending data to an external machine, it is still making effective use of the client-server style.

The client-server style offers reliable connections. If the client knows which servers are available, it can easily connect to them. Better yet, in remote SSH servers or servers on a user's local machine, the client-server style is generally safe from bad actors. That being said, the client-server style depends on knowing exactly what servers are available and where. Furthermore, the client-server style is prone to data availability issues: if a resource is held by only one server, and that server goes offline, then the resource is no longer available. Styles like peer-to-peer mitigate this issue.

Other Relevant Styles

Pub-Sub

We noticed that WezTerm's configuration engine uses event listeners to map custom keybinds to specific workflows. In a pub-sub style, the configuration engine might subscribe to events like "the user presses Ctrl+Shift+T", and the GraphicsManager might publish those events when it receives that specific input. Pub-sub could have been used more generally within WezTerm as a way to handle IO management. A SessionManager could subscribe to events generated by the panes, then redirect WezTerm's IO stream to a specific pane once it receives an event. However, this might cause race conditions if the switching protocol is not well-defined. Although pub-sub is present in specific parts of WezTerm, we determined that the layered style is more prevalent overall.

Object Oriented

WezTerm's configuration engine makes good use of the object-oriented style. In its .wezterm.lua file, WezTerm allows users to access a variety of modules and objects like TabInformation and MuxWindow. Some WezTerm objects are purely informational (e.g. the PaneInformation object), while others have methods that the user can invoke. This allows the user to register custom keybinds, configure their WezTerm GUI, or add new behaviours for certain events that happen in WezTerm. If done right, the object-oriented style could have been a strong alternative to the layered style. If each pane knew where it belonged, what it could access, and the current state of the user's session, then object-oriented would streamline many operations. On the contrary, WezTerm's many domains, tabs, and panes make it difficult to organize objects into a cohesive, easily-manageable system. For those reasons, it makes sense that WezTerm uses a layered style instead of object-oriented.

Subsystems (Components)

1. Multiplexer

WezTerm is most notably known and used for its multiplexing function, where users can have multiple terminals open at once, and these terminals in the multiplexer domain stay open in the background even when not being directly used. This function is especially applicable for work developing, maintaining, and experimenting with servers, as the multiplexing allows the connection to remain even when you are away from your computer ("Multiplexing").

2. Domains

WezTerm organizes where a terminal session runs using domains. By default, a session runs in the local domain, meaning the shell runs on the same machine as the WezTerm window and ends when the window closes. WezTerm can also attach to remote domains, where the shell runs on another machine while the user interacts with it through the local WezTerm window. Connections can be made over SSH, over a Unix socket, or over TLS to a WezTerm multiplexer server running on the remote host.

3. Shell Integration

Shell integration lets the terminal understand what the shell is doing instead of treating output as undifferentiated text. WezTerm supports this through special escape sequences that the shell (via a small configuration script that the user sources) emits to mark things like where each prompt starts, where a command's output begins and ends, and what the current working directory is. With these markers, WezTerm can, for example, open new tabs and panes in the same directory as the current one, and jump between prompts or select a single command's output. Without integration the terminal still works, but these context-aware features are not available (“Shell Integration”).

4. Graphics Manager

Beneath the configuration layer, WezTerm handles the visual side of the terminal itself. It draws the terminal using the GPU, which keeps scrolling and redrawing smooth even with large amounts of output. A windowing layer hides the differences between operating systems so that the same interface runs on Windows, macOS, and Linux. Text rendering is a major focus: WezTerm supports multiple fonts with fallbacks, ligatures, and color emoji, and it can display images inside the terminal. Users can also change how the interface looks, including color schemes, window opacity, tab bar style, and background images. Interface features such as scrollback, search, and quick select (which highlights items like URLs and file paths for easy copying) are built on this same rendering foundation.

5. Customization Engine

WezTerm is configured through a Lua file rather than a fixed settings menu, which gives users far more control than a typical list of options. Because Lua is a full programming language, a configuration can include logic: it can adjust settings by operating system, define custom key bindings, build up its own tab titles, and respond to events such as a window being created or a pane's title changing. WezTerm exposes its features to these scripts through a documented Lua API, and it ships with sensible defaults, so a user can start with an empty file and add only what they want to change. This design treats the configuration as a small program that extends the application. The trade-off is that users need some scripting knowledge, which makes the quality of the documentation important. WezTerm's official docs address this with a reference page for each configuration option and API function, along with worked examples that users can copy and adapt.

Division of Responsibilities

Since Wezterm is an open source project, anyone can contribute through accessing WezTerm’s Github Repository and following the development guidelines outlined in its documentation. Developers looking for tasks can use the Issues tab on WezTerm’s GitHub, where issues are labeled by type (such as bug, enhancement, and codebase-maintenance), and by platform. (“Issues”, Furlong) For developers new to the project, there is a “Good first issues” tab that highlights issues that have relative approachability for first time contributors. If developers are looking for more essential tasks to tackle, they can work on release-gating items marked with

the "Next Release Checklist" issue. (“Contribute”, Furlong). The WezTerm development documentation outlines how to make a local build of the code, how to write documentation for new or changed behavior, and optionally build the docs locally to preview them. Developers interested in contributing can communicate asynchronously using Github, and use real-time chat channels (such as a Matrix room) for quick questions and discussions. (“contributing”, Fidanza)

Because the project can be split up into various components (see Subsystems (Components)), developers with specific expertise can work in parallel on components that fit their skillset the best. For example, a developer without networking knowledge may not be able to help with a bug related to the Local and Remote Hosting component, but if they know about graphics and text rendering, they could improve how fonts and styles are displayed in the User-Interface System . Table 1 depicts the main components and the essential skillsets for developers to work on them:

Component	Developer Knowledge Required
Graphics Manager (UI)	Graphics windowing and text-rendering skills
Multiplexer	Process and session management skills
Domains	Networking, SSH, and security knowledge
Shell Integration	Low-level operating system knowledge and familiarity with terminal standards
Customization Engine	Lua scripting, config design, ability to write clear documentation

Table 1: Development skills required for work on each component

While some parallel work on components can occur, developers must be aware that no component of the system works in isolation, and should understand how their component interacts with others when doing development work. For example, when a user splits a pane in the User-Interface System, the interface asks the Multiplexing component to start a new shell, and Multiplexing relies on the Local and Remote Hosting component to decide whether that shell runs on the local machine or a remote one. A change to how any one of these three handles the request could break the other two, so any change to one of these components must consider its interactions with the other two.

Before submitting a pull request, the documentation outlines specific commands to run the full test suite, explains how to write additional tests, and instructs the developer to perform a code-formatting check. This ensures that the continuous integration system, which automatically verifies that the tests pass and the code is correctly formatted, will not fail the build. Pull requests merge once they are deemed acceptable by the primary reviewer and continuous integration passes. Because this review appears to rest largely on one person working in their spare time, it

can become a bottleneck, and some pull requests wait a long time for a response. (“contributing”, Fidanza)

Concurrency

Concurrency takes unusual shapes in WezTerm and is so seamless that users may not notice that its underlying architecture relies heavily on concurrent processes and tasks. WezTerm draws on the best ideas from Tmux (Marriott) and GNU Screen (Laumann and Bormann), both of which are terminal multiplexers. This differs from traditional terminals on macOS, Linux, and Windows, where closing the window usually interrupts and kills the process. WezTerm combines these terminal multiplexer concepts with additional features, such as GPU acceleration (Furlong) and the ability to connect to serial ports for hardware prototyping and debugging (“Serial Ports”). In contrast to other terminals, which may require plugins or custom scripts to connect to serial output at a specified baud rate (“Serial Ports”), WezTerm combines this functionality with multiplexing to create a useful tool for accelerating development and improving the developer experience. Similar to Tmux, WezTerm also allows users to connect to remote hosts and use multiplexers on them (“Multiplexing”). This is especially useful because processes on these servers are often intended to run 24/7, including websites, bots, databases, email servers, and other applications that require a script or process to remain active throughout their lifecycle. WezTerm refers to each operating system environment as a domain and provides different configuration and setup options for each one (“Multiplexing”). This domain abstraction places local and remote connections under one umbrella, the machine running WezTerm is called the Local Domain, while any remote host is an additional domain that WezTerm can optionally connect to (“Multiplexing”).

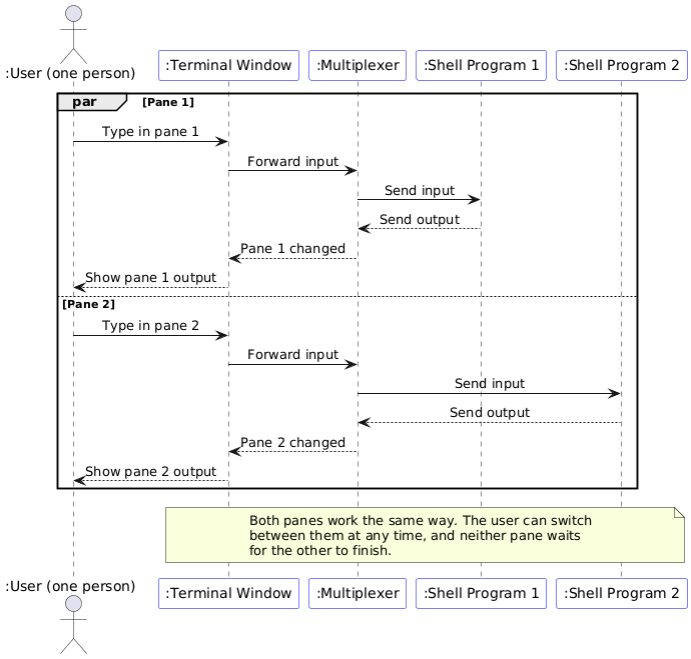


Figure 2 shows WezTerm handles concurrency by running multiple terminal panes in a single window. A user can open two distinct panes that run different programs in parallel. These panes cannot access the other’s content through WezTerm’s protocols. (They could still communicate through a third-party service connecting them)

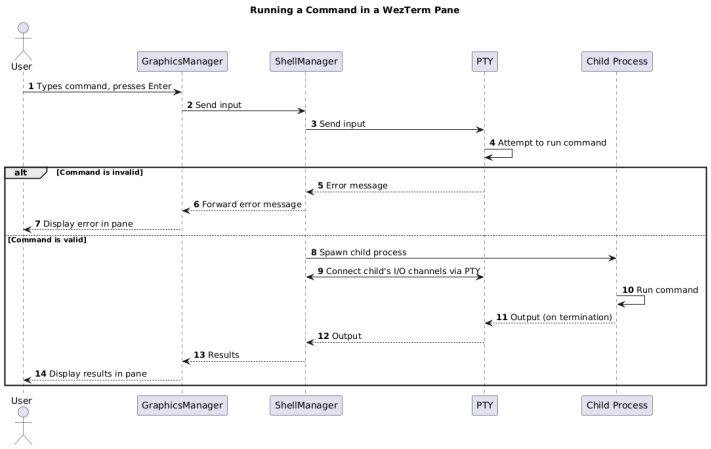
Figure 2: Concurrency in WezTerm

External Interfaces

WezTerm has no bug production database and it relies on memory and plain files instead. It keeps fast in-memory caches with LRU and LFU policies for glyphs, images, and terminal lines so the GUI can draw quickly while it runs. It also uses a small SQLite file cache but only inside the colour sync helper tool and not in the main terminal path. For long term storage it uses Lua for the main config file and plugincode which the user writes and the Lua engine reads at startup, and it uses plain text for logs, session recording, SSH settings, and keys so the user and remote hosts can share trust and the developer can check faults later.

It sends messages through live channels that use standard system technology. The GUI takes keyboard and mouse input and returns rendered text with images and clipboard data for the user. The PTY pipe carries input and window size one way and VT output and exit status the other way for local shells and editors. Local control works through the command line tool and through a multiplexer server over Unix domain sockets on one machine and over TCP with TLS between machines. Remote work uses SSH channels and serial lines for shell traffic and login, and plugins get transferred over HTTPS with Git before they are stored as files in the runtime folder.

Data & Control Flow



WezTerm facilitates many standard terminal processes, and it introduces several new features such as multiplexing, GUI configuration, and detachment/reattachment. This section describes the data and underlying process for three common use cases of WezTerm.

Figure 3: *Sequence diagram for running commands in a WezTerm pane.*

Use Case #1: Running a Command

In WezTerm, users can run some commands via the GUI, while other commands require the user to create a custom keybinding in the .wezterm.lua file. The following steps describe the conceptual data and control flow for running commands in WezTerm.

1. *In a WezTerm pane, the user types on keyboard & press enter.*
2. *The GraphicsManager receives the input and sends it to the ShellManager.*

3. *The pane's ShellManager sends the input to the PTY, which attempts to run the command.*
4. *If the input command is valid, the ShellManager spawns a new child process and connects its IO channels to it via the PTY. If the input command is invalid, then the PTY returns an error message to the ShellManager, which forwards it to the GraphicsManager to display it to the user.*
5. *When the command terminates, the child process sends its output to the ShellManager via the PTY, then the ShellManager sends those results to the GraphicsManager to display it to the user.*

Figure 3 illustrates steps 1 to 5 above. The alt block separates the two outcomes of step 4: if the PTY cannot run the command, it returns an error that the ShellManager forwards to the GraphicsManager for display. If the command is valid, the ShellManager spawns a child process connected through the PTY, and the output reaches the user when the command terminates.

Use Case #2: Customizing WezTerm with the Lua config script.

The following steps describe how WezTerm loads custom configurations, how it listens for new customization updates, and how it communicates the configuration to the GUI.

1. *The user must create a .wezterm.lua file in their chosen configuration directory of their WezTerm application.*
2. *The ConfigurationManager attempts to locate the configuration file. If the file is not found, it uses the default WezTerm configuration. If the ConfigurationManager finds the file, it attempts to load the file and set it as the current configuration.*
3. *The ConfigurationManager sends its selected configuration to the GraphicsManager (this can be a custom configuration or the default.)*
4. *The GraphicsManager updates the WezTerm GUI with the given configuration and registers event listeners for the custom keybinds.*
5. *If a user presses Ctrl+Shift+R, the ConfigurationManager force-reloads the configuration. Then, it sends the configuration to the GraphicsManager which updates the WezTerm GUI.*
6. *While WezTerm is running, the ConfigurationManager watches the configuration file so that if a user changes the file's contents, it will automatically load the new configuration file.*
7. *After successfully loading the new configuration file, the ConfigurationManager sends the new configuration to the GraphicsManager, which updates the WezTerm GUI and overwrites the keybind listeners.*
8. *The GraphicsManager listens for custom keybindings during user input; upon receiving such keystrokes, it fetches the command that is attached to those keybindings and sends it to the current pane's ShellManager.*

   Note: Users can have window-specific configuration (done via the config file), and they can override config values through the command line.

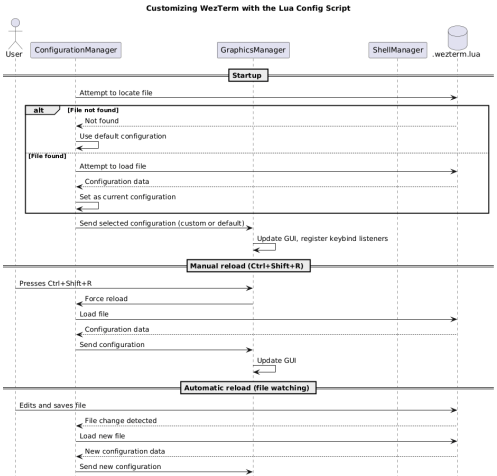


Figure 4 shows the four scenarios in steps 1 to 8: startup, manual reload with Ctrl+Shift+R, automatic reload when the file changes, and custom keybindings. The alt block in the startup scenario shows the fallback to the default configuration when .wezterm.lua is not found, so the GraphicsManager always receives a configuration to apply.

Figure 4: Customizing WezTerm with the Lua config script.

Use Case #3: Connecting to a remote SSH server

The following sequence describes how WezTerm manages the act of connecting to a remote SSH server.

1. *The GraphicsManager receives a command from the user to connect to a remote SSH server. The command contains the name of the desired server.*
2. *The MultiplexingManager fetches the SSH server's information from the ConfigurationManager, which fetches it from the loaded configuration (see Task 3).*
3. *The MultiplexingManager attempts to establish a connection with the remote server using the information obtained in step 2.*
4. *The MultiplexingManager establishes a successful, encrypted connection, or it receives an error message if the connection fails (which it propagates to the ShellManager and GraphicsManager).*
5. *If the user has already been authenticated in that session (e.g. when splitting an existing SSH pane), they may continue without needing to reauthenticate.*
6. *The remote SSH server authenticates the MultiplexingManager via a key signature.*
7. *The MultiplexingManager calls upon the ShellManager and forwards the authentication request to the GraphicsManager for user input.*
8. *Once the user inputs their key signature to the GraphicsManager, it makes its way back to the MultiplexingManager via the ShellManager. The MultiplexingManager sends the key signature through the encrypted channel.*
9. *Once the SSH server authenticates the MultiplexingManager, the client and remote server now have a secure, encrypted communication channel through which they can send messages and commands.*
10. *The MultiplexingManager sends a probe command to check if the remote server has a WezTerm multiplexer running. If the remote server does not have a running WezTerm SSH multiplexer, the remote server sends back an error, which the MultiplexingManager propagates to the GraphicsManager for display.*
11. *Upon successful identification of a WezTerm SSH multiplexer on the remote server, the MultiplexingManager assigns the connection to the appropriate DomainManager on the client side.*
12. *While authenticated, the user can send commands to the GraphicsManager via typing or keybinds. Then, the GraphicsManager routes the commands to the current ShellManager, which notifies the MultiplexingManager and forwards the command's bytes to the remote SSH server.*
13. *The remote SSH server's PTY and shell executes the command, then sends the output back to the MultiplexingManager.*
14. *The MultiplexingManager sends the output back to the relevant ShellManager.*
15. *The ShellManager notifies the GraphicsManager and displays the information on the current pane.*

Name and Data Dictionaries

Name Dictionary	
Name	**Description**
GraphicsManager	Renders the GUI and handles user input (keyboard, keybinds)
ShellManager	Manages a single shell (PTY, input/output, and child processes)
SessionManager	Coordinates panes and tabs: picks the domain for a new pane
DomainManager	Manages one domain's tabs and panes: handles attach/detach
MultiplexingManager	Handles remote SSH and multiplexer server connections
ConfigurationManager	Loads, watches, and hot reloads the Lua config
Customization Engine	Subsystem name for the configuration system
Multiplexer	Subsystem that keeps sessions running and splits panes

Data Dictionary	
Term	**Meaning**
Domain	An environment where a terminal session runs (local or remote)
Local Domain	The machine running WezTerm
Remote Domain	A host reached over SSH, a Unix socket or TLS
Multiplexer Server	A WezTerm process on a remote host that hosts mux sessions
Pane	A single terminal view running its own shell and PTY
Tab	A container holding one or more panes
Window	The top level GUI element holding tabs
PTY (pseudo-terminal)	The interface linking a shell to its terminal
Shell	The program interpreting user commands inside a pane
Escape sequences	Markers emitted by the shell to support shell integration
Scrollback	Saved terminal output the user can scroll and search

Conclusions & Lessons Learnt

We initially assumed that WezTerm would be a relatively small program, we were surprised by the amount of components it requires and how much attention is given to nice-to-have features such as fonts, color schemes, and background images. While the configuration seemed complex at first, we learnt it also allowed users to manage settings, keybinds, and remote connections in one place and see changes immediately. We had initially also questioned why someone would use a separate terminal (WezTerm) when operating systems already provide one. Learning about multiplexing, remote domains, and shell integration answered this.

We also learnt how important consistent communication is. We divided the report into sections early on, and while everyone understood their own section, we did not always have a complete understanding of how the components were connected. This became clear when we combined our work, as some component names and descriptions were inconsistent. More frequent communication and meetings would have helped us catch this earlier. For the next assignment we plan to stay in more consistent communication and meet more frequently so our report is more cohesive as a group.

In conclusion, WezTerm is a cross-platform terminal emulator and multiplexer that appears simple but relies on many components working together. Our architecture divides responsibilities among components such as the GraphicsManager for input and display, the ShellManager and PTY for executing commands, the DomainManager and MultiplexingManager for local and remote domains, and the ConfigurationManager for loading and reloading Lua configuration. Our sequence diagrams show that even running a simple command involves several components. Features such as session detaching and reattaching, remote SSH domains, and hot-reloading configuration help distinguish WezTerm from standard operating-system terminals.

AI Collaboration Report

For this report, we chose to use Claude Sonnet 5.5. We chose Claude as our model provider because its output is more succinct and objective than ChatGPT and other LLMs. ChatGPT tends to agree with the user for the sake of maintaining a positive interaction; Claude does not sugarcoat its responses, which we prefer. We chose Sonnet 5.5 because we did not find it necessary to use any advanced models, nor did we want to pay for increased token usage.

While researching and writing our report, we used Claude for several tasks. For instance, we used it to scan the WezTerm GitHub repository to help us build a conceptual understanding of the software. When one of our team members completed their section, we would sometimes send that section to Claude and ask how it related to our respective sections. Also, when we completed a draft of our conceptual architecture, we fed it to Claude to compare it with the WezTerm documentation. In some cases, Claude identified some of our conceptual misunderstandings. For instance, we said a DomainManager manages specific tabs, but Claude corrected us by noting that a WezTerm tab can contain panes from several domains. More generally, though, we used Claude to review the WezTerm documentation to piece together how the software works.

Each group member used their own AI models for their sections. In a collaborative project like this, it does not make sense for one person to be the sole AI user, because that would bottleneck our progress. Our models are independent from each other; we did not use the exact same context or use a shared Claude plan. Here is a sample prompt from when we were conceptualizing the control flow:

I'm writing a conceptual architecture for WezTerm to learn how it works. My task is to figure out how the data & controls flow between the components. My group has defined the following components, each with their own responsibilites:
1) MultiplexingManager: Handles remote SSH and multiplexer server connections. Works with the domain manager to handle the connection for remote SSH domains.
2) DomainManager: Each WezTerm domain has its own domain manager. A domain manager has context over all its tabs (and child panes) and can be used for detaching, reattaching, etc.
3) ShellManager: Each shell has a shell manager (handles the PTY, input/output transmission, etc.) and communicates with the domain manager to inform it of any changes.
4) GraphicsManager: The graphics manager renders the GUI. It receives information from all of the domain managers so it knows what to render.
5) CustomizationEngine: The customization engine handles the loading and hot reloading of the .wezterm.lua customization file. It works with the GraphicsManager to update the GUI with the new configuration.
I'll go into more detail by describing 5 key tasks soon. But for a conceptual architecture of WezTerm, are these components a good start? Is there anything VITAL that I'm missing?

Claude responded to that prompt by saying it was a strong start but lacked a few key components like a TerminalEmulator component and a component for handling user input. We wanted to maintain a simple, high-level view until we dive into the codebase in Assignment 2, so we delegated the terminal emulation tasks to our ShellManager component. To address our lack of an input handler, we delegated user input handling to our GraphicsManager component.

When we were uncertain of Claude's outputs, we would open the relevant citations, which were usually from wezterm.org or the documentation of another terminal product like tmux. For more general knowledge, sites like YouTube provided trustworthy sources of information. We would always proofread our model's output, and we made sure to cross-reference it with the other information in our report to make sure that everything was coherent. Since we largely wrote the finished report without the help of AI, we proofread it ourselves.

We estimate that AI contributed to 26% of this report. To calculate this number, each member gave an estimate for how much AI they used in their section of the report. Then, we divided the sum of those estimates by 6 (our group size). As mentioned in the AI use cases subsection, we mainly used AI to understand the systems, not generate the report itself. Moreover, when we wanted to verify the AI model's claims, we visited the documentation for WezTerm and other terminals (e.g. tmux) to make sure the AI's output was legitimate. For these reasons, we believe that our 26% estimate is accurate.

Integrating a virtual AI teammate accelerated our progress through the assignment. While working independently, we would use Claude to ask clarifying questions, rather than waiting for other members to reply in our group chat. That said, sometimes it created more work. For instance, we initially thought that the data and control flow required a deep-dive into the codebase. If we talked to our teammates earlier, we could have saved a few hours of confusion.

Our AI usage did not cause any challenges within our group, because we used AI as a tool, not a crutch. By doing so, we knew what was going on in each of our sections and could discuss them with our groupmates, rather than outsourcing all our thinking to our AI models.

One major lesson we learned is that just because you *can* use AI to do a task, that does not mean you *should*. Without the proper context and constraints, LLMs will overcomplicate their output or write something completely out of scope for the task at hand. When needing to clarify important details or requirements, human-to-human communication usually works best.

Works Cited

"Attach - Wez's Terminal Emulator." *Wezterm.Org*, 2018, https://wezterm.org/config/lua/MuxDomain/attach.html. Accessed 5 Oct. 2026.

Cloudflare. "What Is SSH? | Secure Shell (SSH) Protocol." *Cloudflare.Com*, 2023, https://www.cloudflare.com/learning/access-management/what-is-ssh/. Accessed 5 Oct. 2026.

"Configuration - Wez's Terminal Emulator." *Wezterm.Org*, 2018, https://wezterm.org/config/files.html. Accessed 6 Oct. 2026.

"Detach - Wez's Terminal Emulator." *Wezterm.Org*, 2018, https://wezterm.org/config/lua/MuxDomain/detach.html. Accessed 5 Oct. 2026.

"Features." *Wez's Terminal Emulator*, https://wezterm.org/features.html. Accessed 4 Oct. 2026.

Fidanza, Simone. "Contributing to Wezterm", *GitHub*, https://github.com/wezterm/wezterm/blob/main/CONTRIBUTING.md#contributing-to-wezterm

"front_end." *Wez's Terminal Emulator*, https://wezterm.org/config/lua/config/front_end.html. Accessed 4 Oct. 2026.

Furlong, Wez. "Contribute." *GitHub,* https://github.com/wezterm/wezterm/contribute Accessed 4 Oct. 2026.

Furlong, Wez. "Issues." *GitHub,* https://github.com/wezterm/wezterm/issues Accessed 4 Oct. 2026.

Furlong, Wez. "wez/wezterm." *GitHub*, https://github.com/wez/wezterm. Accessed 4 Oct. 2026.

Furlong, Wez. "wez/wzterm/commit ab74b4c". *GitHub.* https://github.com/wezterm/wezterm/commit/ab74b4c73174313ac622d4a240a33b0128cadcf3. Jan. 26 2018. Accessed Oct. 5 2026.

Furlong, Wez. "wez/wezterm/commit 08bcf1". *GitHub*. https://github.com/wezterm/wezterm/commit/0ab8cf1eec64a600091c8ec638432ff658fe23f4. Jan. 28 2018b. Accessed Oct. 5 2026.

Laumann, Oliver, and Carsten Bormann. “GNU Screen.” *Screen - GNU Project*, 1987, https://www.gnu.org/software/screen/.

Marriott, Nicholas. “Home · tmux/tmux Wiki · GitHub.” *Tmux Wiki*, 2007, https://github.com/tmux/tmux/wiki. Accessed 4 October 2026.

“Multiplexing.” *Wez’s Terminal Emulator*, https://wezterm.org/multiplexing.html. Accessed 4 Oct. 2026.

Parsa Khazaeepoul. “What Is a Terminal Multiplexer? Tmux, Zellij, Herdr, and Pane.” *RunPane by Dcouple*, Pane, 4 Oct. 2026, https://runpane.com/what-is-a-terminal-multiplexer. Accessed 5 Oct. 2026.

“Quick Selected Mode” *Wez’s Terminal Emulator*, https://wezterm.org/quickselect.html. Accessed 4 Oct. 2026.

R/Wezterm, www.reddit.com/r/wezterm/. Accessed 2 Oct. 2026.

“Scrollback.” *Wez’s Terminal Emulator*, https://wezterm.org/scrollback.html. Accessed 4 Oct. 2026.

“Serial Ports & Arduino.” *Wez’s Terminal Emulator*, https://wezterm.org/serial.html. Accessed 4 Oct. 2026.

“Shell Integration” *Wez’s Terminal Emulator*, https://wezterm.org/shell-integration.html. Accessed 4 Oct. 2026.

“SSH.” *Wez’s Terminal Emulator*, https://wezterm.org/ssh.html. Accessed 4 Oct. 2026.

Wang, Jessica. “Demystifying the Terminal: How It Works Behind the Scenes.” *DEV Community*, 6 Sept. 2023, https://dev.to/therubberduckiee/demystifying-the-terminal-how-it-works-behind-the-scenes-50h6. Accessed 4 Oct. 2026.

“WezTerm - Wez’s Terminal Emulator.” Wezterm.Org, 2018, https://wezterm.org/index.html. Accessed 2 Oct. 2026.

"wez/wezterm/commit c53ca64". GitHub, Dec. 2017, https://github.com/wezterm/wezterm/commit/c53ca64c33d1658602b9a3aaa412eca9c6544294#diff-b335630551682c19a781afebcf4d07bf978fb1f8ac04c6bf87428ed5106870f5. December 7 2017. Accessed 5 Oct. 2026.

“Workspaces / Sessions - Wez’s Terminal Emulator.” *Wezterm.Org*, 2018, https://wezterm.org/recipes/workspaces.html. Accessed 6 Oct. 2026.