

MyTerminal Design Document

MyTerminal Team

October 23, 2025

Abstract

MyTerminal is a lightweight graphical terminal application implemented with X11/Xlib, optionally accelerated by Pango/Cairo for robust UTF-8 rendering. It provides a shell-like environment supporting pipelines, redirections, history, tabs, background jobs, and a custom `multiWatch` command that executes multiple commands in parallel and streams their outputs with timestamps. This document details the architecture, components, execution model, data flows, and key design decisions.

Contents

1	Introduction	2
1.1	Overview	2
1.2	Repository Structure	3
1.3	Requirements	3
1.3.1	Functional	3
1.3.2	Non-Functional	4
2	Core Architecture and Execution Model	4
2.1	High-Level Components	4
2.2	Event Loop and Scheduling	4
2.3	Data Flow (typical external command)	5
2.4	TerminalWindow	5
2.5	Tab	5
2.6	Command Submission and Scheduling	5
3	Feature: Command Parsing and Multiline Input	6
3.1	Splitting Lines and Commands	6
3.2	Multi-line Input and Continuation	6
4	Feature: Persistent History and Line Editing	6
4.1	History Implementation	6
4.2	Line Editing and Search	7
5	Feature: Autocomplete Functionality	7
5.1	Design	7
5.2	Implementation	7
5.3	Limitations	7
5.4	Usage	8
6	Feature: Pipelines and I/O Redirection	8
6.1	Parsing and Wiring	8
6.2	PTY Usage	8

7	Feature: Background Jobs and Signal Handling	8
7.1	Design	8
7.2	Implementation	9
7.3	Limitations	9
7.4	Usage	9
7.5	Background Detaching (Ctrl+Z)	9
7.6	Signal and Process Group Management	9
8	Feature: Custom Command multiWatch	9
8.1	Goals	9
8.2	Overview	10
8.3	multiWatch Worker Loop Implementation	10
8.4	File/Descriptor Strategy	11
8.5	Cleanup Strategy and Interrupts	11
8.6	Formatting	11
9	Feature: Rendering, Unicode, and UI/UX	12
9.1	Multiline Unicode Input and Rendering	12
9.2	UI/UX Details	12
10	Supporting Details	12
10.1	Commands and Shortcuts	12
10.1.1	Built-in Commands	12
10.1.2	Keyboard Shortcuts	13
10.1.3	Feature-to-Commands Mapping	14
10.2	Error Handling and Edge Cases	14
10.3	Performance Considerations	14
10.4	Build and Configuration	14
10.4.1	Runtime Paths and Permissions	15
10.5	Security Considerations	15
10.6	Testing Strategy	15
10.7	Limitations and Future Work	15
11	Appendix: Key APIs	16
11.1	TerminalWindow	16
11.2	Tab	16
11.3	multiWatch (worker inner loop)	16

1 Introduction

1.1 Overview

MyTerminal combines a minimal graphical terminal UI with a shell runner:

- X11-based windowing, custom rendering and input handling
- Command execution via fork/exec with pipes or PTY when interactive
- Persistent command history, search, and basic autocompletion
- Tabs with independent job state
- Background job support

- **multiWatch**: run N commands in parallel per period; stream outputs with UNIX timestamps and per-command headers; clean up on interrupt/exit

1.2 Repository Structure

- **include/**
 - **gui/TerminalWindow.hpp**: Main app class and public methods
 - **gui/Tab.hpp**: Per-tab UI and execution state
 - **core/History.hpp**: History model with persistence and search
- **src/**
 - **gui/TerminalWindow.cpp**: X11 window, event loop, drawing, input, scheduling
 - **gui/Tab.cpp**: Text rendering helpers and small utilities
 - **core/CommandExecutor.cpp**: Command parsing, built-ins, pipelines, processes, multi-Watch, jobs
 - **core/History.cpp**: History implementation
 - **app/main.cpp**: Program entry (includes exit-time sweeper)
- **Makefile, Makefile.nopango, CMakeLists.txt**: Build scripts (Makefile uses Pango/Cairo; Makefile.nopango disables Pango/Cairo; CMake toggles via `USE_PANGO_CAIRO=ON—OFF`)
- **temp/**: Runtime temporary FIFOs used by **multiWatch**

1.3 Requirements

1.3.1 Functional

- Execute arbitrary commands, including pipelines (`cmd1 | cmd2`), with redirection (`>`, `<`, `2>`).
- Maintain persistent command history (load at startup, append on every command) and provide inline search (`Ctrl+R`).
- Provide basic autocomplete for built-in commands, external executables, and file/directory paths via Tab key.
- Support multiline unicode input, handling characters from various languages and preserving encoding in display.
- Provide line navigation shortcuts: `Ctrl+A` to move cursor to the beginning of the line, `Ctrl+E` to the end.
- Support multi-line input via unmatched quotes and line-continuation (trailing `\`); treat it as a single logical command.
- Support multiple commands in one submission (newline-separated or semicolon-separated outside quotes) and execute them sequentially.
- Provide background detaching via `Ctrl+Z`; continue to drain and print background output. The `'&'` operator is not supported.
- Provide a custom **multiWatch** command that:
 - Executes N commands in parallel for each period.
 - Creates a hidden temporary FIFO per child using the child's real PID: `temp/.temp.<PID>.txt`.

- Reads all child outputs via `poll()` over those FIFO descriptors and streams output as it arrives.
- Prints per-command headers with UNIX timestamp and separator framing.
- Cleans up on Ctrl+C: terminate cycle children, unlink all FIFOs, exit worker; restore scrollbar in UI.
- Removes lingering temp files when the shell closes (`atexit` sweep) and also sweeps on worker startup.
- Render ANSI-colored output; optionally use Pango/Cairo for robust UTF-8 shaping.
- Provide tabs and basic UI affordances (scrollbar, prompt, separators between pasted commands).

1.3.2 Non-Functional

- Linux/X11 environment; C++17.
- Single-process GUI (no threads) with nonblocking I/O to keep UI responsive.
- Clean failure modes: if exec fails, report errors; avoid orphaned FIFOs; sweep temp directory on exit/start.
- Reasonable performance for typical command output and up to dozens of `multiWatch` commands.

2 Core Architecture and Execution Model

2.1 High-Level Components

TerminalWindow X11 window, event loop, input handling, command scheduling, rendering pipeline.

Tab Per-tab state: buffers, job descriptors, queues, continuation state, `multiWatch` snapshot.

Command Execution Parser + launcher for built-ins, pipelines, PTY selection, and process I/O wiring.

multiWatch Worker A forked child that orchestrates per-period parallel children and streams via FIFOs.

History Persistent store for commands; search and best-match suggestions.

2.2 Event Loop and Scheduling

- The X11 loop handles input events (`KeyPress`, `Mouse`), periodic repaint ticks, and file-descriptor polling for child output.
- `runNextCommand()` dequeues and dispatches commands sequentially; built-ins return quickly and chain to next; external commands advance when reaped.
- Between commands from a pasted batch, a visible separator line is printed to avoid output confusion.

2.3 Data Flow (typical external command)

1. User submits text; it is split into logical commands while respecting quotes and semicolons.
2. A command is parsed into pipeline stages with redirections.
3. The process tree is forked; pipes/PTY are wired.
4. Parent monitors nonblocking read ends; chunks are appended to `scrollback` after ANSI processing.
5. On EOF and successful `waitpid()`, scheduling advances.

2.4 TerminalWindow

Primary UI controller (see `TerminalWindow.hpp`):

- X11 setup, event loop, drawing: tab bar, text area, scrollbar
- Input handling: UTF-8 IME (XIM/XIC), line editing, history search (Ctrl+R), autocomplete
- Command submission and scheduling across `Tab::pendingCmds`
- Process I/O pump: nonblocking read from child stdout/stderr; background drains
- Prompting and transcript building with simple ANSI parsing and coloring
- Optional Pango/Cairo rendering for multilingual text

2.5 Tab

Per-tab state container (`Tab.hpp`):

- Text buffers: `scrollback`, `input`
- Cursor and scrolling metrics
- Foreground job: `childPid`, `childPgid`, and FDs
- Background jobs: list of `BackgroundJob`
- Continuation state for multi-line commands (quotes/backslashes)
- Queue: `pendingCmds` for sequential execution
- `multiWatch` session flag and saved `scrollback` snapshot

2.6 Command Submission and Scheduling

1. The user enters a line; `MyTerminal` echoes the prompt and line.
2. The input may contain:
 - Multiple commands separated by newlines or semicolons (outside quotes)
 - Continuation lines for unmatched quotes or line-continuation backslashes
3. Parsed commands are enqueued into `Tab::pendingCmds`.
4. `runNextCommand()` dequeues and dispatches one at a time:

- Built-ins execute synchronously and now call `runNextCommand()` when done.
 - External programs launch via `fork/exec`. Completion is detected through nonblocking I/O pump and `waitpid()` (`WNOHANG`); then `runNextCommand()` continues.
5. For clarity with multi-command pastes, a visual separator line is printed between commands.

3 Feature: Command Parsing and Multiline Input

3.1 Splitting Lines and Commands

Split by newline respecting quotes. We walk bytes and toggle flags `inS/inD` for single/-double quotes. Newlines outside quotes delimit commands. Whitespace-only fragments are discarded.

```
inS=false, inD=false, cur=""
for c in input:
    if c=='"' && !inS: inD=!inD; cur+=c; continue
    if c=="'" && !inD: inS=!inS; cur+=c; continue
    if c=='\n' && !inS && !inD: push(cur if non-empty non-whitespace); cur=""; continue
    cur+=c
push(cur if non-empty non-whitespace)
```

Split by semicolon respecting quotes. Applied per logical line for sequences like `cmd1; cmd2`.

3.2 Multi-line Input and Continuation

- Support multi-line input via unmatched quotes and line-continuation (trailing `\`); treat it as a single logical command.
- Continuations show `>` prefix; transcript preserves exact typed content (excluding continuation backslashes).
- Continuation with unmatched quotes and `\` line-joins retains logical command structure.

4 Feature: Persistent History and Line Editing

4.1 History Implementation

(`History.hpp/.cpp`) Persistent ring with:

- Append, clear, save/load to `~/.myterm_history`: History is stored in a plain text file in the user's home directory, with each command on a new line. The file is loaded at startup into a vector for quick access. Commands are appended after successful execution, and the file is saved periodically or on exit. The `clear` subcommand resets the in-memory history and truncates the file.
- Search with exact and substring strategies: Exact search matches commands that start with the query. Substring search uses longest common subsequence (LCS) to find commands containing the query characters in order, prioritizing higher LCS lengths.
- Best-match suggestions for autocomplete: When autocompleting, history provides suggestions based on prefix matches.

4.2 Line Editing and Search

- Line editing supports basic input; Ctrl+A (start of line) and Ctrl+E (end of line): The input line is a string buffer with a cursor position. Characters are inserted at the cursor, and Ctrl+A moves the cursor to index 0, Ctrl+E to the end of the string.
- Ctrl+R searches backward in command history: Initiates reverse search mode, where typing narrows down matches from history. Up/down arrows cycle through matches, and Enter selects one to fill the input line.
- Backspace/Delete edit text: Backspace removes the character before the cursor and shifts the rest left; Delete removes the character at the cursor.

5 Feature: Autocomplete Functionality

5.1 Design

- When the user presses the Tab key during input, the shell attempts to complete the current word.
- If the word matches the prefix of a built-in command, external command, or file/directory in the current working directory, possible completions are suggested or inserted.
- Directory-prefix-aware replacements are supported: typing a partial directory or file name and pressing Tab will complete or list matches.
- For file name completion:
 - If one file matches the prefix, complete the input with the full file name.
 - If multiple files match, complete to the longest common prefix if it extends the current input.
 - If multiple matches remain after prefix completion, display numbered options (e.g., 1. file1.txt 2. file2.txt) and prompt the user to select by number.

5.2 Implementation

- The input handler scans the current input buffer for the word under the cursor.
- Built-in commands are matched from a static list; external commands are matched from the system PATH.
- File and directory matches are found using globbing in the current working directory.
- The transcript is updated with suggestions if multiple matches are found; otherwise, the input buffer is updated with the completed word.

5.3 Limitations

- Autocomplete does not support advanced shell grammar (e.g., variable expansion, command substitution).
- Only single-word completion is supported; multi-stage pipeline or quoted arguments may not autocomplete as expected.
- No fuzzy matching; only prefix matches are considered.

5.4 Usage

- Press Tab while typing a command or file path to trigger autocomplete.
- If multiple completions are possible, they will be listed in the transcript area.
- Example: typing `ec` and pressing Tab will complete to `echo`.
- Example: typing `test_fi` and pressing Tab will complete to `test_file.txt` if present in the directory.
- Example (file completion): In a directory with files `"abc.txt"`, `"def.txt"`, `"abcd.txt"`:
 - Typing `./myprog de` and pressing Tab completes to `./myprog def.txt`.
 - Typing `./myprog abc` and pressing Tab displays `"1. abc.txt 2. abcd.txt"` and waits for user input (e.g., pressing 1 completes to `./myprog abc.txt`).

6 Feature: Pipelines and I/O Redirection

6.1 Parsing and Wiring

1. Parse stages, and within each stage, extract redirections for `stdin/stdout/stderr`.
2. Create N-1 pipes for an N-stage pipeline.
3. For single-stage with no redirection, attempt PTY for interactive behavior; else use pipes.
4. Child i: `dup2` appropriate fds (`stdin` from prev pipe or file; `stdout` to next pipe or parent `outPipe`; `stderr` to `errPipe` unless redirected).
5. Close unused fds in children and parent; parent sets nonblocking on read ends.

6.2 PTY Usage

- PTY mode is used for single-stage interactive commands (TTY-friendly behavior); otherwise, pipes are used.

7 Feature: Background Jobs and Signal Handling

7.1 Design

- Background jobs are processes detached from the foreground, allowing continued execution while the shell accepts new commands.
- `Ctrl+Z` detaches the current foreground job, moving it to background; output is asynchronously drained and displayed.
- Jobs are tracked by PID/PGID and file descriptors for output monitoring.
- Built-in commands `bgpids` list active background jobs, and `killprocess` sends signals to terminate them.
- Signal handling: `Ctrl+C` interrupts foreground process groups; `multiWatch` sets its own PGID for targeted interruption.

7.2 Implementation

- Ctrl+Z handler: Sends SIGTSTP to foreground PGID, adds job to `backgroundJobs` vector with PID/PGID/FDs.
- `drainBackgroundJobs()`: Nonblocking read from background FDs, appends output to scroll-back.
- `bgpids`: Iterates `backgroundJobs`, prints PID and command name.
- `killprocess`: Parses PID and signal (default SIGTERM, -9 for SIGKILL), sends via `kill`.
- Process groups: Foreground jobs set PGID; Ctrl+C uses `killpg` to signal group.

7.3 Limitations

- No job control commands like `fg` or `jobs`.
- No `'&'` operator for backgrounding at command end.

7.4 Usage

- Ctrl+Z: Detach running command to background.
- `bgpids`: List background jobs with PIDs and commands.
- `killprocess` PID: Terminate a background job (use -9 for force kill).

7.5 Background Detaching (Ctrl+Z)

Ctrl+Z detaches the current foreground job; output is drained and appended to the transcript. The `&` operator is not supported. Jobs are tracked with PID/PGID and FDs.

7.6 Signal and Process Group Management

- Foreground jobs form a process group; Ctrl+C sends SIGINT via `killpg`.
- `multiWatch` worker sets its own PGID so Ctrl+C targets the whole watch job.

8 Feature: Custom Command `multiWatch`

8.1 Goals

- Run multiple commands *in parallel* each period
- Stream outputs intermixed as data is available, not after completion
- Tag each stream with command and UNIX timestamp
- Clean up cleanly on Ctrl+C and when the shell exits

8.2 Overview

1. The shell forks a **multiWatch worker** process.
2. Each period, the worker:
 - (a) Forks N children; each child writes to a FIFO at `temp/.temp.<PID>.txt` (one per command).
 - (b) The worker opens nonblocking read-ends for all FIFOs and uses `poll()` to multiplex.
 - (c) As data arrives, it prints:
 - Header: `"cmd" , current_time: <unix_timestamp> :`
 - Separator line: 56 dashes (as configured)
 - Raw command output (may be empty)
 - Trailing separator line
 - (d) On EOF/HUP for a FIFO, the worker closes and unlinks that FIFO.
 - (e) After all commands finish, the worker sleeps for the configured interval and repeats.
3. Ctrl+C kills the worker's process group; the worker's signal handler then:
 - Kills all child PIDs for the current cycle
 - Unlinks any FIFOs it knows about
 - Exits
4. The parent (GUI) reads the worker's stdout and paints the transcript; upon worker exit, it restores the scrollbar that was present before multiWatch started.

8.3 multiWatch Worker Loop Implementation

Rationale for FIFOs over regular files. Regular files are always readable by `poll()`, making readiness semantics unhelpful. FIFOs support readiness and EOF/HUP, enabling true streaming.

Per period algorithm.

```
// At worker startup: sweep temp/.temp.*.txt (orphan cleanup)
loop forever:
  clear mw_pids, mw_tempfiles
  mkdir("temp", 0755)
  // Spawn N children
  for i in 0..N-1:
    p = fork()
    if child:
      tf = "temp/.temp." + getpid() + ".txt"
      // Nonblocking open loop until reader is up
      while ( (wfd=open(tf, O_WRONLY|O_NONBLOCK))<0 ) sleep 10ms
      dup2(wfd, 1); dup2(wfd, 2); close(wfd)
      execlp("sh", "sh", "-c", cmd[i], NULL)
      _exit(127)
    else:
      tf = "temp/.temp." + p + ".txt"; unlink(tf); mkfifo(tf, 0644)
      rfd = retry open(tf, O_RDONLY|O_NONBLOCK)
      pfds.push({fd:rfd, events:POLLIN|POLLHUP|POLLERR}); map pfds->cmdIndex

  // Stream with poll
  headerPrinted[j]=false; trailerPrinted[j]=false
```

```

while openCount>0:
    poll(pfds, 200ms)
    for each j with revents:
        if POLLIN: read; on first bytes print header+separator; write chunk
        if read==0 or POLLHUP/ERR: drain; ensure trailing separator; close+unlink fifo;
    openCount--

// Reap children and record exit codes (optional); sleep interval seconds

```

8.4 File/Descriptor Strategy

The design uses **FIFOs** for each child invocation:

- Satisfies requirement to create hidden temp files named with the real PID
- Enables true nonblocking multiplexing with `poll()` over read descriptors
- Files are short-lived: unlinked as soon as the corresponding child completes

A sweep is performed at worker startup and at shell exit (via `atexit`) to remove stale entries in `temp/`.

8.5 Cleanup Strategy and Interrupts

- **Per-stream:** On EOF/HUP, close the FIFO and immediately `unlink()` it.
- **On signals:** SIGINT/TERM/HUP/QUIT handled by the worker: kill cycle PIDs, unlink all `mw.tempfiles`, exit.
- **On worker startup:** Sweep `temp/.temp.*.txt` via glob and unlink.
- **On app exit:** `atexit` sweep in `main.cpp` unlinks lingering `temp/.temp.*.txt`.

8.6 Formatting

Header/Separator Formatting. On first data (or EOF without data) for a given command in the period, print:

```

"<cmd>" , current_time: <unix_timestamp> :
-----
<data if any>
-----

```

The dashed line count is constant in code and can be tuned; a separate, longer dashed line is used between queued commands from pasted input.

Per command per cycle:

```

"cmd" , current_time: 1690000000 :
-----
<Output>
-----

```

Repeat for each command, and then each cycle.

9 Feature: Rendering, Unicode, and UI/UX

9.1 Multiline Unicode Input and Rendering

The requirement specifies X11/Xlib for rendering, but X11 alone is insufficient for robust handling of multiline Unicode input, particularly when dealing with characters from various languages that require complex shaping and layout. To address this, MyTerminal optionally uses Pango/Cairo for UTF-8 text rendering, which provides proper glyph shaping, bidirectional text support, and accurate positioning for multilingual content.

For example, if the user types a command like `echo "Hello "`, the Unicode characters (e.g., Chinese characters) are correctly shaped and displayed in the terminal output, preserving encoding and layout. This is achieved by integrating Pango's text layout engine with Cairo's rendering backend, ensuring that multiline inputs and outputs with mixed scripts are rendered faithfully without corruption.

The build system supports this via the `USE_PANGO_CAIRO` flag in CMake, or by using `Makefile` (which includes Pango) versus `Makefile.nopango` (which disables it for minimal builds).

- Xlib text drawing with ANSI color approximation; optional Pango/Cairo rendering for true UTF-8 shaping
- Custom scrollback and scrollbar; 60 Hz redraw loop for smoothness
- Prompt coloring (user/host/cwd) with theme-able palette
- ANSI parsing tracks ESC/CSI states and applies color/intensity where feasible.

9.2 UI/UX Details

- Prompt format: `user@host:cwd$` ; colored segments for readability.
- Between commands in a multi-command submission, a long separator line is printed: -----

10 Supporting Details

10.1 Commands and Shortcuts

10.1.1 Built-in Commands

The shell supports the following built-in commands, which are handled internally without forking external processes:

multiWatch

Syntax: `multiWatch [interval] ["cmd1", "cmd2", ...] or multiWatch [interval] cmd1 cmd2 ...`

Runs multiple commands in parallel each period, streaming outputs with headers and timestamps. Interval defaults to 5 seconds if omitted.

Example: `multiWatch 10 ["date", "uptime"]`

Feature: multiWatch (parallel monitoring)

bgpids

Syntax: `bgpids`

Lists all active background job PIDs with their command names.

Feature: Background Jobs

killprocess

Syntax: `killprocess [-9] PID [PID ...]`

Sends SIGTERM (or SIGKILL if -9) to the specified PIDs.

Example: `killprocess 1234`

Feature: Background Jobs (use Ctrl+Z to detach a running foreground job; " is not supported)

echo

Syntax: `echo [args ...]`

Prints arguments to stdout, separated by spaces.

Feature: Command Execution

history

Syntax: `history` or `history clear`

Shows command history; `clear` removes all history.

Feature: History

cd

Syntax: `cd [directory]`

Changes current working directory; defaults to home if no argument.

Feature: Command Execution

clear

Syntax: `clear`

Clears the terminal screen.

Feature: UI/UX

pwd

Syntax: `pwd`

Prints the current working directory.

Feature: Command Execution (implied in prompt)

10.1.2 Keyboard Shortcuts

The following keyboard shortcuts are supported for input editing and control:

- **Ctrl+C**: Interrupt the current foreground command (sends SIGINT to process group).
- **Ctrl+Z**: Send the current foreground command to background (detaches job).
- **Ctrl+A**: Move cursor to the beginning of the line.
- **Ctrl+E**: Move cursor to the end of the line.
- **Ctrl+R**: Search backward in command history.
- **Arrow Keys**: Navigate cursor left/right.
- **Backspace/Delete**: Edit text.

These shortcuts enhance usability for line editing, history access, and job control.

10.1.3 Feature-to-Commands Mapping

For reference, here is a mapping of features to their associated commands/shortcuts:

- **Command Execution (Pipelines, Redirections):** External commands (e.g., `ls | grep foo > out.txt`), `echo`, `cd`, `pwd`.
- **Interactive Mode & PTY:** Automatic for single-stage commands like `vim`.
- **Background Jobs:** `bgpids`, `killprocess`, `Ctrl+Z` (the `”` operator is not supported).
- **multiWatch:** `multiWatch`, `Ctrl+C` (interrupt).
- **Input Handling:** Multi-line with quotes/backslashes, semicolon/newline separation.
- **History:** `history`.
- **Signal Handling & Cleanup:** `Ctrl+C`, `Ctrl+Z`.
- **Rendering & Output:** ANSI colors, `clear`, `Ctrl+L`.
- **Temp Files & Resource Management:** Automatic (FIFOs for `multiWatch`).
- **Build & Configuration:** `make`, `cmake`.
- **UX Enhancements:** Separators, prompts, shortcuts.

10.2 Error Handling and Edge Cases

- **Command not found:** prints *command not found* with status 127
- **Globbing:** expanded via `glob(3)`; tokens not matching become literals
- **UTF-8 input:** Accepted via XIM/XIC; unrecognized control bytes are dropped
- **PTY fallback:** If PTY allocation fails, falls back to pipe mode
- **multiWatch FIFO races:** Reader open retries and nonblocking write open in child avoid `ENXIO/ENOENT` races

10.3 Performance Considerations

- Nonblocking I/O and short poll timeouts keep UI responsive.
- `multiWatch` scales roughly $O(N)$ in number of polled FIFOs; each poll loop is bounded and data is chunked.
- PTY path is used for single-stage interactive commands to avoid line-buffering surprises and to support TTY-aware programs.

10.4 Build and Configuration

Two build routes:

- **Make (with or without Pango/Cairo):**

```
# With Pango/Cairo (default Makefile uses Pango)
make -f Makefile
make -f Makefile clean

# Without Pango/Cairo
make -f Makefile.nopango
make -f Makefile.nopango clean
```

- **CMake:**

```
cmake -S . -B build -DUSE_PANGO_CAIRO=ON    # or OFF to disable Pango/Cairo
cmake --build build
```

For CMake builds, toggle Pango/Cairo via the cache option `-DUSE_PANGO_CAIRO=ON|OFF`.

10.4.1 Runtime Paths and Permissions

- Temp FIFOs live under `temp/` with filenames `.temp.<PID>.txt`. Default mode 0644 is adequate for single-user project work; production-grade terminals should place `temp` under `/tmp/myterm-<uid>` with directory mode 0700 to avoid cross-user interference.
- `umask` may affect FIFO file modes; code assumes a permissive default.

10.5 Security Considerations

- No shell escaping is performed beyond glob expansion; commands are executed via `execvp` or `sh -c` (in `multiWatch`). Treat input as untrusted.
- Temp FIFOs are created under project `temp/`; production usage should prefer `/tmp/myterm-<uid>` with 0700 to avoid cross-user interference.
- Signal handling avoids partial cleanup by unlinking known FIFOs and killing child PIDs on exit paths.

10.6 Testing Strategy

- Unit test core string parsing where feasible (history, splitting helpers)
- Integration tests:
 - Pipes/redirections correctness
 - PTY execution for interactive programs
 - `multiWatch` with short/long commands; verify headers, timestamps, and cleanup
 - Signal handling on `Ctrl+C` and shell exit: ensure no `temp/.temp.*.txt` remains

10.7 Limitations and Future Work

- Full POSIX job control is limited; background output is supported but lacks job control commands like `fg/jobs`.
- No shell scripting language—this is a runner with simple parsing, not a full shell grammar.
- `multiWatch` currently uses `sh -c`; direct `argv` variants could avoid shell interpolation.
- Optional enhancements:

- `--keep-temp`: keep FIFOs/files for debugging instead of unlinking
- Persistent per-command logs via parent-side `tee`
- Configurable separators and header formats
- More robust ANSI parsing and colors
- Better prompt customization and themes

11 Appendix: Key APIs

11.1 TerminalWindow

```
void run();
void executeLine(const std::string& line);
void executeLineInternal(const std::string& line, bool echoPromptAndCmd);
void spawnProcess(const std::vector<std::string>& argv);
void pumpChildOutput();
void drainBackgroundJobs();
void submitInputLine(Tab& t, bool triggerRedraw = true);
void runNextCommand(Tab& t);
static std::vector<std::string> splitArgs(const std::string& s);
```

11.2 Tab

```
std::deque<std::pair<std::string, bool>> pendingCmds;
pid_t childPid, childPgid;
int outFd, errFd, inFdWrite;
bool watchActive;
std::string savedScrollbarBeforeWatch;
```

11.3 multiWatch (worker inner loop)

```
// For each period:
- Fork command children
- Parent mkfifo temp/.temp.<PID>.txt and opens read end (nonblocking)
- Child opens write end and dup2s stdout/stderr to it
- Parent poll()s across all FIFO read fds:
  - Print header, then separator, then stream data, then trailing separator
- Close and unlink FIFO when done
- Sleep interval seconds and repeat
```