Termail: A Secure, Fast, and Extensible Terminal Email Client

Kyle Trinh
kytrinh@ucsd.edu

Alexis Vergnet
avergnet@ucsd.edu

Nick Petrone
npetrone@ucsd.edu

Abstract

We introduce *termail*, a secure, keyboard-driven terminal email client built in Rust. Addressing the privacy risks associated with third-party extensions, *termail* implements a novel plugin architecture using the WebAssembly (WASM) Component Model. This approach enforces a strict capability-based security model, where plugins operate in isolated sandboxes and cannot access network or file system resources without explicit user consent. Leveraging Tokio for asynchronous performance and an incremental Maildir synchronization strategy, *termail* demonstrates how to combine the efficiency of a terminal interface with the safety guarantees required for sensitive communications.

1. Introduction

Email remains one of the most ubiquitous forms of digital communication, yet modern email clients have increasingly prioritized visual complexity over efficiency and security. For power users, developers, and privacy-conscious individuals who spend significant time in terminal environments, existing solutions often fall short.

We introduce *termail*, a security-first command line interface email client. We designed it primarily for users who prefer a keyboard-driven workflow and for users who wanted better extensibility.

To implement extensibility securely, we architected a plugin system using the WebAssembly (WASM) Component Model. Because email is highly sensitive, we could not trust external plugins by default. The WASM model allows us to enforce a strict capability-based security model. Each plugin is defined by a manifest (the WIT) that explicitly dictates what kinds of data they are allowed to see and modify. By default, each plugin is sandboxed. Unless the user explicitly grants network or file system capabilities, it is impossible for a foreign plugin to extract sensitive data and send it to a malicious party.

We built *termail* in Rust specifically to leverage its compile-time memory safety guarantees[1]. This ensures the core application is immune to common vulnerabilities like buffer overflows or race conditions, providing a stable and secure foundation for running untrusted WASM plugins.

Speed is achieved through Rust's zero-cost abstractions and an efficient asynchronous architecture built on Tokio[2]. *termail* employs local Maildir synchronization[3], enabling offline access to emails and reducing latency for common operations.

We demonstrate how the combination of Rust's memory safety guarantees, WebAssembly's sandboxing capabilities, and careful API design enables a terminal client that is simultaneously secure, performant, and highly customizable.

2. *termail*

2.1. Architecture Overview

termail follows a modular, event-driven architecture built on Tokio, Rust's asynchronous runtime. The system separates concerns across five primary components:

- **UI Layer**: A Ratatui-based terminal interface that renders views and processes user input. The UI operates on an event loop that unifies terminal events, tick events (30 FPS), and application events into a single `tokio::select!` loop.
- **Backend Layer**: Email provider implementations behind a common `Backend` trait. Currently supports Gmail (via OAuth2 and the Gmail API) and Greenmail (via IMAP for local testing). Backends execute commands such as `FetchInbox`, `SendEmail`, and `SyncFromCloud`.
- **Maildir Layer**: Local email storage using the Maildir format. A SQLite database caches message metadata (date, subject, sender) for fast sorting without parsing every email file.
- **Plugin Layer**: A WebAssembly runtime powered by Wasmtime that executes sandboxed plugins. Plugins register for hooks (`before_send`, `after_send`, `before_receive`, `after_receive`) and can modify email content during processing.
- **Config Layer**: TOML-based configuration that specifies backend credentials, plugin lists, editor preferences, and display options.

These components interact through async message passing. When the user initiates an action (e.g., syncing emails), the UI spawns an async task that calls the backend. The backend fetches data from the remote provider, writes to the Maildir, and sends an `AppEvent` back through the event channel. The UI receives the event and updates its state, triggering a re-render. Plugins intercept this flow at defined hook points, processing email content within their sandboxed WASM environment before returning control to the host.

2.2. User Interface

termail follows a modular, event-driven architecture designed to separate concerns between user interface, email

operations, local storage, and plugin execution. The system is built on Tokio, Rust's asynchronous runtime, enabling non-blocking I/O operations that keep the terminal interface responsive while performing network requests and disk operations in the background.

We originally designed it as a Text-based User Interface (TUI), but found that it was much easier to test the underlying backend using a Command Line Interface (CLI), allowing us to keep the UI decoupled from the data store and the "business logic".

2.2.1. TUI Framework Selection

We evaluated two Rust TUI frameworks: Ratatui and Cursive. Cursive provides a higher-level, widget-based API with built-in layout management and event routing, similar to traditional GUI toolkits[4]. Ratatui takes a lower-level approach, providing immediate-mode rendering primitives where the application explicitly controls every frame [5].

We selected Ratatui for the following reasons:

- **Expressiveness**: Ratatui's immediate-mode rendering model provides direct control over layout and drawing.
- **Async compatibility**: Ratatui integrates cleanly with Tokio's async runtime. The framework does not impose its own event loop, allowing us to unify terminal events, tick events, and application events into a single `tokio::select!` loop.
- **Ecosystem**: Ratatui has active development and a growing ecosystem of companion crates, including ratatui-image for terminal graphics protocol support (Kitty, Sixel).

The tradeoff is increased implementation complexity. Ratatui requires manual management of layout constraints, state, and rendering, but this cost is acceptable for an application that requires fine-grained control over the terminal interface. In particular, image rendering is not possible in cursive.

2.2.2. External Editor Integration

While implementing the `ComposeView` component, we recognized that building a standalone text editor within *termail* would be infeasible. A full-featured editor like `Vim` represents decades of development effort. Instead, *termail* delegates email composition to the user's preferred external editor.

When the user enters the body field in `ComposeView` and presses Enter, *termail* executes the follow Rust code:

```
use std::io::stdout();

// 1. Stop event polling
self.events.stop_events();

// 2.  Suspend TUI
execute!(stdout(), LeaveAlternateScreen);
disable_raw_mode();

// 3. Run editor (blocks until user saves and
exits)
let result = Editor::open(&editor_cmd,
current_draft);

// 4. Restore TUI
enable_raw_mode();
execute!(stdout(), EnterAlternateScreen);
terminal.clear()?;
self.events.start_events();

// 5. Update state with edited draft
composer.draft = result?;
```

The user is then able to send their email through their selected backend.

2.2.3. Image Rendering

Terminal-based applications have historically been limited to text output. However, modern terminal emulators such as Kitty and Ghostty support graphics protocols that enable inline image display. *termail* leverages these protocols to render image attachments directly within the message view.

Multiple competing standards exist for terminal image rendering. However, *termail* uses the `ratatui-image` crate to abstract over these protocols.

2.2.4. Loading and Limitations

To avoid blocking the UI, *termail* uses optimistic view transitions: when a user opens an email, the message view displays immediately with text content while image attachments load asynchronously in the background. Once loaded, an `EmailLoaded` event triggers image decoding and rendering.

The current implementation only renders the first PNG attachment per email. Terminals without graphics protocol support display no images.

2.3. Synchronization Strategy

Ingesting incoming emails into *termail* presented two architectural options:

1. Implementing a custom Mail Delivery Agent (MDA) to write directly to a *Maildir* structure.
2. Treating a cloud mail provider as the source of truth and synchronizing the local state.

We selected the latter. The resulting architecture couples a *Maildir* directory for each remote client with a SQLite database for metadata tracking. *Maildir* is a filesystem-based format where each email is stored as an individual file within three subdirectories: `/tmp`, `/cur`, and `/new`. These files adhere to the RFC 822[6] standard (superseded by RFC 2822[7] and 5322[8]). Adopting *Maildir* over strictly API-based access enables offline capabilities and interoperability with external Unix mail tools.

To maintain consistency between *termail* and the source of truth (Gmail), we implemented a three-tier synchronization strategy: Full Sync, Smart Sync, and Incremental Sync.

- **Full Sync**: This process executes when the local database is empty. It performs an initial, comprehensive

download of all messages from Gmail, populating the *Maildir* directories.

- **Smart Sync**: This mode handles state reconciliation, particularly for resuming interrupted downloads or managing large mailboxes. By computing the set difference between remote `gmail_ids` and local database records, the system identifies messages requiring download or deletion. For messages present in both sets, only metadata is fetched to update state (e.g., label changes), avoiding redundant body downloads.
- **Incremental Sync**: This mode leverages Gmail API history records, which persist for approximately one week. The system tracks a `last_sync_id`. If this ID remains valid remotely, the system retrieves only the history records generated since the last checkpoint. If the history data is expired or invalid, the system automatically falls back to a Smart Sync to ensure integrity.

2.4. Plugin System

termail's plugin system enables third-party extensions while maintaining strict security boundaries. Plugins run as sandboxed WebAssembly modules, isolated from the host system and each other. This section describes the design decisions, interface definitions, and runtime architecture that make this possible.

2.4.1. Design Requirements

Before selecting an implementation approach, we identified four requirements for *termail*'s plugin system:

1. **Security without implementation burden**: The plugin runtime should provide isolation guarantees without requiring us to implement sandboxing ourselves. Email is highly sensitive data, and plugins must not be able to exfiltrate content or compromise the host.
2. **No recompilation of *termail***: Users should be able to install, update, and remove plugins without rebuilding the host application.
3. **Language agnosticism**: Plugin authors should not be restricted to a single language. Systems like VS Code (JavaScript), Hyprland (C++) [9], and Neovim (Lua) [10] limit plugins to one language, excluding developers unfamiliar with that ecosystem.
4. **Richer than stdin/stdout pipes**: The plugin interface should support structured data and bidirectional communication, not just text streams.

2.4.2. Alternatives Considered

We evaluated several plugin architectures before selecting WebAssembly:

Native shared libraries (e.g., Hyprland's approach): Plugins compile to `.so` files loaded via `dlopen`. This offers high performance and direct access to host APIs, but provides no isolation—a malicious plugin has full access to process memory. This is unacceptable for an email client.

IPC-based plugins: Each plugin runs as a separate process, communicating with the host via pipes, sockets, or D-Bus. This provides process-level isolation but introduces serialization overhead and complexity. Security still depends on correctly sandboxing each process which we would need to implement ourselves.

Embedded scripting language (e.g., Neovim's Lua): The host embeds an interpreter and exposes a scripting API. This is simpler than IPC but restricts plugins to a single language.

WebAssembly: Plugins compile to WASM modules executed by a runtime like Wasmtime [11]. WASM provides memory isolation by design—modules cannot access host memory or syscalls unless explicitly granted. The Component Model adds language-agnostic typed interfaces.

2.4.3. WebAssembly Component Model

We selected the WebAssembly Component Model over traditional WASM modules for several reasons:

- **Language independence**: Plugins can be written in any language that compiles to WASM components, including Rust and Python. The Component Model defines a standard ABI, eliminating the need for language-specific FFI bindings.
- **Capability-based security**: Components have no ambient authority. They cannot access the filesystem, network, or environment variables unless the host explicitly provides those capabilities through WASI.
- **Typed interfaces**: WIT (WebAssembly Interface Types) provides a language for defining strongly-typed contracts between host and guest, catching interface mismatches at compile time rather than runtime.

2.4.4. WIT Interface Definition

The plugin interface is defined in a single WIT file that specifies two APIs:

```
package tm: plugin-system;

interface event-api {
    variant event {
        before-send(string),
        after-send(string),
        before-receive(string),
        after-receive(string),
    }
}

interface host-api {
    call-host:  func(invocation-id: string,
request: string)
        -> result<string, string>;
}

world plugin {
    import host-api;
    use event-api. {event};
    export on-notify: func(invocation-id: string,
event: event) -> event;
}
```

The `event-api` defines the four hook points as variants of an `event` enum, each carrying a string payload. The `host-`

`api` allows plugins to call back to the host for operations they cannot perform themselves. The `plugin` world ties these together: plugins import host capabilities and export the `on-notify` function that *termail* invokes.

2.4.5. Hook-Based Architecture

Plugins register for specific hooks in their manifest file:

```
name = "signature-rs"
description = "Appends a signature to outgoing
emails"
backends = ["gmail"]
hooks = ["before_send"]
```

The four available hooks correspond to stages in the email lifecycle:

Hook	Description
`before_send`	Modify email content before transmission
`after_send`	Trigger actions after successful send
`before_receive`	Process incoming emails before display
`after_receive`	Post-processing on received emails

When multiple plugins register for the same hook, they execute in sequence. Each plugin receives the output of the previous plugin, forming a processing pipeline.

2.4.6. Plugin Lifecycle

Plugin loading follows a four-stage process:

1. **Discovery**: *termail* scans `~/.config/termail/plugins/` and `./plugins/` for directories containing a `manifest.toml` file.
2. **Manifest validation**: The manifest is parsed and validated. Plugins specifying backends that do not match the active backend are skipped.
3. **Component instantiation**: The WASM component is loaded, a per-plugin `Store` is created with its own WASI context, and the component is instantiated against the linker.
4. **Hook registration**: The plugin instance is added to a `HashMap<Hook, Vec<LoadedPlugin>>` for dispatch.

2.4.7. Host Callbacks and Invocation IDs

Plugins may need to request information or actions from the host that they cannot perform within their sandbox. *termail* supports this through the `call-host` import, secured by invocation IDs.

When *termail* dispatches an event to a plugin, it generates a UUID and stores the event in a map keyed by that ID. The plugin receives the ID and can use it to authenticate callbacks:

A plugin attempting to call `call-host` with an invalid or expired invocation ID receives an error, preventing replay attacks or unauthorized access.

2.4.8. Capability-Based Security

By default, plugins have no capabilities. This means plugins cannot:

- Read or write files
- Access environment variables
- Make network requests
- Spawn processes

without explicit user approval. The only channel for plugin-host communication is the explicitly defined `host-api` interface.

2.4.9. Limitations of the WebAssembly Interface Types (WIT) format

WebAssembly Interface Types[12] allows us to define a "world" in which we provide host function calls to WebAssembly plugins that take protected actions in *termail* that cannot be done from within the Wasmtime sandbox. Worlds also export a set of functions which *must* be implemented by a plugin, otherwise it will not be allowed to load.

With WIT, we can use the `wit-bindgen` crate to generate static bindings for this world. This gives us static types for both plugin-provided functions as well as host functions *termail* provides to plugins. Adherence to the WIT world specification is checked at plugin-load time; if any of the exported plugin function signatures don't match the expected types, a panic is automatically generated and the plugin is not loaded.

Given this, WIT is still a developing specification that shows its limitations when trying to express several kinds of useful patterns. WIT does not yet support passing closures as arguments (useful for asynchronous operations) and does not support optional function exports, meaning that every plugin must implement every function we export in the WIT file. We wanted to implement an event handler pattern in which there are a variety of different events with different associated data, and each event has an associated response. Strongly linking only one type of event with only one type of response was not possible.

3. Security Considerations

Security is the primary design constraint of *termail*. This section details the specific mechanisms used to secure user data.

3.1. Rust Memory Safety

The foundation of *termail*'s security is the Rust programming language. *termail* leverages Rust's ownership and borrowing model to guarantee memory safety at compile time. By strictly enforcing rules about data access and lifetime management, the Rust compiler prevents entire classes of vulnerabilities.

3.2. WebAssembly Sandboxing

To safely execute third-party code, *termail* relies on the **WebAssembly (WASM) Component Model**. This pro-

vides a significantly stronger isolation boundary than shared libraries (`.so`/`.dll`) or interpreted scripts (Python/Lua).

Plugins in *termail* do not share address space with the host process. Instead, they run inside a virtualized environment managed by **Wasmtime**. This architecture provides memory isolation and control flow integrity.

3.3. Capability-Based Access Control

Isolation alone is insufficient; plugins need to interact with the world to be useful. *termail* solves this using a **Capability-Based Security** model via the WebAssembly System Interface (WASI).

A plugin has zero privileges by default—it cannot open files, access the network, or read environment variables. Access must be explicitly granted by the user via the plugin manifest (`manifest.toml`) and is enforced by the runtime.

4. Conclusion

In this paper, we presented *termail*, a terminal-based email client that reconciles the tension between extensibility and security. By leveraging **Rust** for its memory safety guarantees and **Tokio** for asynchronous performance, we demonstrated that a keyboard-driven interface can be both responsive and reliable without the legacy debt of older clients.

Our primary contribution is the implementation of a **WebAssembly-based plugin architecture**. By adopting the WASM Component Model, *termail* successfully isolates third-party extensions, strictly enforcing a capability-based security model through WebAssembly Interface Types (WIT). This approach proves that power users do not need to grant full inbox access to untrusted code to achieve a customizable workflow.

While the current implementation focuses on Gmail integration and core email operations, the modular design lays a stable foundation for future expansions, such as multi-account support and full-text search. Ultimately, *termail* illustrates that modern systems programming paradigms can revitalize traditional terminal tools, providing a viable, privacy-respecting alternative to commercial web-based clients.

5. Future Work

While *termail* provides a functional terminal email client with secure plugin extensibility, several areas remain for future development. The following list represents features and improvements we identified but did not implement within the scope of this project:

- **Full-text search**: Integrate a local search engine to enable fast querying across downloaded emails.
- **Additional email providers**: Extend backend support beyond Gmail to include Outlook, generic IMAP/SMTP (although this is partially implemented with Greenmail), and other common email providers.
- **Email deletion and archiving** Add TUI commands for deleting, archiving, and moving emails between folders, with bidirectional synchronization to the remote backend.
- **Reply and forward functionality**: Implement reply, reply-all, and forward actions with proper quoting of original message content and header preservation.
- **Status bar notifications**: Add a persistent status bar to display operation feedback such as "Email sent successfully" or "Sync in progress," rather than relying solely on log output.
- **Label navigation in TUI**: Implement scrolling and selection within the labels sidebar, allowing users to filter emails by Gmail labels or folders.
- **Expanded attachment support**: Support viewing and saving additional attachment types beyond PNG images, including PDFs, documents, and other common formats.
- **Plugin capability grants**: Extend the plugin manifest format to allow explicit capability declarations (e.g., network access, filesystem access) that users can review and approve before plugin execution.
- **Performance optimization for image resizing**: The current image resize request handling through the event channel may introduce latency. Alternative approaches such as dedicated image processing threads could improve responsiveness.
- **Secure plugin installation from remote repositories**: Implement a CLI command to install plugins directly from GitHub repositories. This would involve cloning the repository, validating the plugin manifest, compiling the source to WASM, and pinning the installed version to a specific commit hash. The primary challenge is secure compilation: compiling untrusted code on the user's machine introduces risk, so a potential solution would involve remote verified builds in a secure enclave or trusted build service that produces signed WASM artifacts.
- **HTML rendering in the terminal**: Many modern emails are HTML-only or HTML-primary. Integrating a terminal HTML renderer would improve readability for these messages while maintaining the terminal-native experience. However, we found that Gmail will output a `text/plain` MIMEType which is ideal for terminal environments.
- **Secure network access for plugins**: Expose network capabilities to plugins through host-mediated calls, with URL allowlists defined in the plugin manifest. This raises non-trivial access control questions: should `google.com` permit requests to `google.com/arbitrary/path`, or only to the domain root? Should wildcard patterns be allowed? Wildcards combined with email read access would enable a malicious plugin to exfiltrate entire mailboxes to an attacker-controlled subdomain. A careful

capability design must balance plugin utility against the risk of data exfiltration.

6. Related Work

Mutt is a terminal email client first released in 1995 and still actively maintained. Mutt provides keyboard-driven navigation and supports extensive configuration through its macro system. Its extension model relies on shell scripts and external programs, which execute with full system privileges [13].

Notmuch focuses on email indexing and search rather than acting as a standalone email client[14]. Built around a Xapian-based search engine, *notmuch* provides tagging and query capabilities designed to scale to large mailboxes. Users typically pair *notmuch* with a frontend such as Emacs, Alot, or neomutt. This modular architecture requires users to configure and maintain multiple components. Customization of the Emacs interface requires familiarity with Emacs Lisp.

Aerc is a more recent terminal email client written in Go [15]. Aerc provides built-in support for multiple accounts, a tabbed interface, and an embedded terminal for composing messages. Aerc handles OAuth2 authentication natively for some providers. Its extension model uses filter commands and external scripts, which execute with full system privileges.

7. References

[1] The Rust Team, “Rust: A language empowering everyone to build reliable and efficient software.” [Online]. Available: https://www.rust-lang.org/

[2] Carl Lerche, “An asynchronous Rust runtime.” [Online]. Available: https://docs.rs/tokio/latest/tokio/

[3] D. J. Bernstein, “Maildir Format.” [Online]. Available: https://cr.yp.to/proto/maildir.html

[4] A. Bury, “Cursive: A TUI library for Rust.” [Online]. Available: https://github.com/gyscos/cursive

[5] J. McKinney, “Ratatui.” [Online]. Available: https://github.com/ratatui/ratatui

[6] David H. Crocker, “STANDARD FOR THE FORMAT OF ARPA INTERNET TEXT MESSAGES.” [Online]. Available: https://www.rfc-editor.org/info/rfc822

[7] E. Resnick P., “Internet Message Format.” [Online]. Available: https://www.rfc-editor.org/info/rfc2822

[8] E. Resnick P., “Internet Message Format.” [Online]. Available: https://www.rfc-editor.org/info/rfc5322

[9] The Hyprland Contributors, “Hyprland Plugins.” [Online]. Available: https://wiki.hypr.land/Plugins/Using-Plugins/

[10] The Neovim Contributors, “Nvim Lua Guide.” [Online]. Available: https://neovim.io/doc/user/lua-guide.html

[11] The Bytecode Alliance, “Wasmtime: A fast and secure runtime for WebAssembly.” [Online]. Available: https://wasmtime.dev/

[12] WebAssembly Alliance, “The WebAssembly Component Model.” [Online]. Available: https://component-model.bytecodealliance.org/introduction.html

[13] Michael Elkins, “The Mutt E-Mail Client "All mail clients suck. This one just sucks less."." [Online]. Available: http://www.mutt.org/

[14] Carl Worth, “Notmuch -- Just an email system.” [Online]. Available: https://notmuchmail.org/

[15] Robin Jarry, “aerc - a pretty good email client.” [Online]. Available: https://aerc-mail.org/