ChatGPT

Background: Emacs-like Editor Architecture in Rust

Traditional Editor/IDE Architecture: Text editors and IDEs generally follow a Model–View–Controller (or MVP) design. The *model* is the text buffer (often implemented with a gap buffer, piece table, or rope data structure [1] [2]), the *view* renders the text on screen (handling fonts, scrolling, syntax highlighting, etc.), and the *controller* processes input events (keyboard/mouse) and updates the model. A cross-platform event loop (e.g. via `winit` [3] in Rust) polls OS events and dispatches them to the GUI. IDEs add layers on top (code-parsing, indexing, build systems, debuggers). For example, the Xi Editor (a Rust-based backend) uses a *persistent rope* and CRDT for efficient edits [4], separating the editing engine from GUI frontends.

Text Buffer Models: Text is often stored not as a single string but in a data structure optimized for insertions/deletions. Traditional editors used a *gap buffer* or *linked list of lines*. Modern editors often use a *rope* (a balanced tree of string chunks) for large files [2]. Rust's `ropey` crate, for example, is a UTF-8 rope designed for text editors [2]. The editor core must support operations like undo/redo, cut-and-paste, and maybe multi-file/project management. IDEs overlay additional architecture (background parsing, indexers, LSP or Language Server integration).

GUI and Event Loop: GUI toolkits manage windows, drawing, and input events. In Rust, common choices include *immediate-mode* libraries like **egui** and *declarative* libraries like **Iced** or **Druid**. `winit` provides the underlying window and event loop [3]. For instance, **egui** is "a simple, fast, highly portable immediate mode GUI library for Rust" [5]. Immediate-mode frameworks redraw the UI every frame based on state. **Iced** follows the Elm architecture: it separates *state (Model)*, *messages (events)*, *update logic*, and *view logic* [6]. Druid is a data-driven, retained-mode toolkit (using widget trees and property lenses). All rely on an event loop: each OS event (mouse/keyboard) is handled by the framework, which updates state and redraws as needed. Rust's memory safety means these libraries manage state without garbage collection (often using `Arc`/`Rc` for shared data).

Emacs Design Principles: Emacs is the archetype of a highly-extensible editor. Its key principles include:

- **Extensibility:** The core of Emacs is written in C (originally TECO, then Emacs Lisp), but almost all editing functionality is implemented in an embedded Lisp. "Extensibility means that the user can add new editing commands or change old ones…Emacs is written in a modular fashion, composed of many separate and independent functions. The user extends Emacs by adding or replacing functions, writing their definitions in the same language that was used to write the original Emacs system" [7]. In practice, Emacs Lisp libraries (loaded as needed) let users redefine behavior, create new modes, etc. Emacs supports *global and mode-specific hooks*, parameters, and keymaps that let users customize nearly every action.

- **Scripting & Dynamic Configuration:** Emacs uses an interpreted Lisp for scripting, so users can write code on-the-fly. Config files (`.emacs` or `init.el`) contain Lisp that runs at startup. This allows interactive tweaking and immediate testing. Emacs also provides "libraries" of common code that can be shared. Because Emacs Lisp is dynamic, users can redefine functions or variables at runtime (e.g. via `load` or `require`).

- **Self-Documentation:** Emacs has powerful help features. All commands, variables, and functions have documentation strings. The editor is "self-documenting": users can query what functions/commands are available, their keybindings, and even source code with commands like `describe-function` or `apropos`. As the Emacs paper notes, the system "provides powerful and complete interactive self-documentation facilities with which the user can find out what is available" [8]. This culture of documentation means new users can discover features without external manuals.

- **Modes & Units of Text:** Emacs is organized into *major modes* (editing modes for each language or file type) and *minor modes*. Each major mode typically defines syntax rules and language-specific commands (moving by words/sentences, indentation, etc.) [9]. The user or mode can redefine things like what constitutes a word or paragraph. The Emacs paper highlights that for each language a mode can "redefine any of the commands, and reset any parameters, so as to customize Emacs for that language" [9]. Units of text (characters, words, sexps, etc.) are first-class, enabling structural editing commands.

Plugin/Extension System Patterns: Designing a plugin system involves decoupling extensions from the core:

- **Plugin Architecture:** Core applications define *interfaces or APIs* that plugins implement [10] [11]. The host then *dynamically loads* or discovers plugins at runtime. For example, a plugin might be a shared library or separate process providing functions according to a known protocol. Key elements include versioning, interface stability, and a loading mechanism (e.g. scanning a plugins folder, RPC, or language bindings). Articles on plugin systems note that the core "dynamically load[s] and integrate[s] plugins at runtime" without modifying core code [12]. Plugins interact through well-defined interfaces (similar to OSGi services [13]) or base classes/traits.

- **Open/Closed Principle:** A well-known design guideline is that software should be *open for extension but closed for modification*. That is, we should be able to add new functionality (via plugins) without changing the host's source. SOLID principles (especially Open/Closed) often guide plugin-friendly design [11]. This usually means defining abstract traits or interfaces in the core, and having plugins register implementations.

- **Service/Module Lifecycle:** Systems like OSGi manage plugins as *services* with lifecycle (install/start/stop) and dependencies [13]. While an editor may not need full OSGi complexity, the pattern of *loading, unloading, and versioning* plugin modules is relevant. For instance, Vim and Atom allow plugins to be added/removed at runtime; a robust design detects version mismatches and isolates failures. Emulating this in Rust typically requires dynamic linking or inter-process isolation.

- **Safety and Isolation:** A crucial pattern (seen in modern editors) is isolating plugins so they can't crash the main editor. Xi-editor explicitly emphasizes *asynchronous, separate-process plugins*: "plugins should be asynchronous, and it should be possible to write them in any language…A slow plugin should not interfere with typing…a crashing plugin should not lead to data loss" [14]. Many editors use language servers (LSP) or RPC to sandbox plugins. In Rust, using separate processes or WASM modules (with no direct memory access) can provide similar isolation.

Rust as Host and Config Language: Using Rust as both the editor core language and the scripting/config language poses challenges:

- **Compiled vs. Interpreted:** Rust is statically compiled with no built-in interpreter. Unlike Emacs Lisp, you can't simply `eval` user-written Rust code at runtime. One approach is to compile user config files (e.g. with `rustc` or a tool like `cargo-script`) and load them. Another is dynamic loading: build plugins or config as DLLs or shared objects and load with `libloading` [15]. However, Rust's lack of a stable ABI makes this tricky; the `abi_stable` crate is one effort to enable stable Rust dynamic libraries [16].

- **Macros and Include!:** Rust's `macro_rules!` and procedural macros (`syn`/`quote`) work at compile-time, not runtime. They can't directly emulate a live scripting environment. One could use `include!(concat!(env!("HOME"), "/.config/myeditor.rs"))` to compile user code into the binary, but this only happens at build time.

- **Embedded Scripting:** Another pattern is embedding a scripting language. Rust has crates like `rhai` or `deno` (for JS/TS) or `mlua`/`rlua` (Lua) that allow runtime scripts. The NullDeref blog on Rust plugins notes Lua as a common choice for embeddable runtime scripting [17]. Embedding an interpreter allows truly dynamic user code, but sacrifices Rust's type-checked execution.

- **Configuration:** Even if the config is Rust code, one can use a hybrid: e.g. user writes config in a high-level file (TOML, or a small DSL) that the Rust core parses and acts on. Some editors use *domain-specific languages* for config (e.g. Dhall, JSON, or custom). For full flexibility, one might still call out to external scripts.

Performance, Safety, Concurrency (Rust): Rust offers C-like performance without garbage collection. Its ownership/borrow checker prevents entire classes of bugs (use-after-free, data races). As the Rust book notes, its type system "helps manage memory safety and concurrency problems" so that many concurrency bugs become compile-time errors [18]. This "fearless concurrency" model lets the editor use multiple threads or async tasks (e.g. a background syntax checker) without easily introducing race conditions [19]. Rust's zero-cost abstractions mean UI rendering and text operations can be fast (e.g. using SIMD in rope operations). The trade-off is that the code must satisfy strict borrow rules; for example, sharing a text buffer across threads requires synchronization types (`Arc<Mutex<…>>` or channels). In a long-running editor, Rust's memory safety reduces leaks/undefined behavior, but plugins loaded via `unsafe` (e.g. C or dynamic Rust code) could violate that, so care is needed.

Relevant Rust Crates and Libraries:

- **GUI:** *egui* (immediate mode) [5], *Iced* (Elm-like, declarative) [20], *Druid* (data-oriented, widget tree), *slint* (UI markup language), or using *wgpu/skia/glow* directly. These provide windowing, drawing, and widget support.
- **Window/Event Loop:** *winit* for cross-platform windows and events [3]. Many GUI toolkits build on top of winit.
- **Text Buffer:** *ropey* (rope for UTF-8 text) [2], *xi-rope* (from the xi editor), or *buffer-tree* crates. Ropes allow efficient edits on large text.
- **Concurrency:** *crossbeam* (threads, channels), *tokio* or *async-std* (async runtime) for background tasks.

- **Parsing & AST:** *syn* (Rust parser crate) and *quote* (quasi-quoting for code generation) allow writing procedural macros [21]. For general parsing, *tree-sitter* has Rust bindings [22] for incremental, grammar-based parsing. Editors often use Tree-sitter to get an AST for syntax highlighting or structural editing (Xi's docs show parsing and editing with Tree-sitter in Rust [22]).
- **Macros & Code Gen:** As above, `syn`/`quote` enable writing Rust code that generates Rust code (e.g. custom derive, DSLs). This could support compile-time plugin registration or config macros.
- **Plugin Loading:** *libloading* (DLL loading), *abi_stable* (stable Rust ABI helper) [16], or RPC frameworks (JSON-RPC, cap'n proto) for communicating with external plugin processes.
- **Configuration:** *serde* (for deserializing config files in formats like TOML/JSON), or embedded scripting engines (*rhai*, *Lua*, etc.) if choosing a dynamic config language.

Mapping Lisp-like Extensibility to Rust: Emacs's model (dynamic Lisp with runtime `eval` and macros) is fundamentally different from Rust's static model. To approximate it:

- **Callbacks and Traits:** Instead of defining a function at runtime, the Rust core can expose hooks (as traits or function pointers) that user code implements. For example, the editor could allow user code to register command functions or event handlers by calling a setup function during init. These would be Rust closures or function references – but they must be compiled into the binary (or loaded from a library) ahead of time. This is more static than Emacs's dynamic redefinition.

- **Procedural Macros:** Rust's macro system (compile-time) can simulate some extensibility. For instance, a user could write a custom attribute or derive that generates boilerplate for common editor actions. However, procedural macros run at *build time*, not at runtime, so they're more about easing the authoring of code than providing live extensibility.

- **Dynamic Loading or Scripting:** To get true runtime extensibility, one can use dynamic libraries or an embedded interpreter. The plugin blog notes that dynamic loading of Rust code requires heavy use of `unsafe` and careful design, possibly using crates like `abi_stable` [16] to handle versioning. An alternative is WebAssembly: compile plugins to WASM modules. WASM can be safely sandboxed (no unsafe, portable) [23], at the cost of some overhead. The plugin series suggests that WASM "wins against dynamic loading in terms of security by not needing `unsafe` at all" [23].

- **Emulating Lisp Macros:** Emacs Lisp macros and functions allow on-the-fly code generation. In Rust, one could instead rely on metaprogramming (serde-based config, code generation before compile, or embedding a small language with macro-like syntax). However, there is no direct runtime equivalent of Lisp macros in Rust.

In summary, one would likely design the editor so that extensibility happens either via Rust functions loaded at startup or via a separated extension layer (like external scripts/plug-ins). For example, Xi-editor's approach was to use an RPC plugin model, allowing any language and isolating failures [14], rather than embedding a static plugin system. Mapping Lisp's openness to Rust means leaning on static traits/macros or on external modules rather than trying to reinterpret Rust at runtime.

ASTs and Macro Systems in Editor Customization: Abstract Syntax Trees (ASTs) are central to many advanced editor features. Parsing code into an AST (using a library like Tree-sitter [22]) enables structural editing (e.g. select a function or expression), smart indentation, refactoring, and syntax-aware search/replace. Emacs modes traditionally use regex or simple parsing, but modern editors integrate actual parsers. Rust's `syn` crate can parse Rust code, and Tree-sitter supports many languages with incremental reparse.

Macro systems relate in two ways: first, user customization often involves small macros or DSLs. In Emacs, users write Lisp macros or scripts for new commands. In Rust, customization might use procedural macros (`syn` + `quote`) to generate repetitive code or to embed DSL fragments. For example, one could use a macro to define a new editor command: inside `quote!{ ... }` you generate the Rust code for that command. The `quote` crate is specifically intended for this: "procedural macros receive a stream of tokens, execute code to manipulate them, and produce tokens back" [21]. Thus Rust's macros are a compile-time analog of Lisp macros, though they operate on Rust code syntax rather than being an interactive REPL.

Overall, AST-based customization means the editor can introspect or transform code based on its syntax tree. For instance, one might allow a user to write a small Rust-like script (or DSL) to define code refactoring patterns, which the editor compiles with `syn`/`quote` and runs on the AST. Alternatively, plugins (in any language) could use Tree-sitter ASTs to apply edits. The theory here is that syntactic structure and generative macros enable advanced, language-aware editor features – but in Rust they largely occur at compile time or via external tools, rather than as live Evalable Lisp.

Key References: Numerous projects and papers inform this design. The Emacs design paper by Stallman lays out the core principles (extensibility, self-documentation, Lisp macros) [7] [8]. Plugin system patterns are discussed in texts like *The Architecture of Open Source Applications* (e.g. Eclipse's plugin model) and blogs (e.g. Mario Ortiz's "Plugins in Rust" series [24]). Xi Editor's docs illustrate a Rust text engine with ropes and async plugins [4] [14]. Crate documentation (e.g. *ropey* [2], *egui* [5], *Iced* book [20]) and the Rust book (concurrency chapter [19]) provide further grounding in the concepts above.

---

[1] Building a Text Editor: the Gap Buffer // Work Dad Dev
https://workdad.dev/posts/building-a-text-editor-the-gap-buffer/

[2] ropey - Rust
https://docs.rs/ropey/latest/ropey/

[3] winit - Rust
https://docs.rs/winit/latest/winit/

[4] Home
https://xi-editor.io/

[5] GitHub - emilk/egui: egui: an easy-to-use immediate mode GUI in Rust that runs on both web and native
https://github.com/emilk/egui

[6] [20] Architecture - iced — A Cross-Platform GUI Library for Rust
https://book.iced.rs/architecture.html

[7] [8] [9] EMACS: The Extensible, Customizable Display Editor - GNU Project - Free Software Foundation
https://www.gnu.org/software/emacs/emacs-paper.html

[10] [12] Building a plugin architecture with Managed Extensibility Framework - Part 3 | Elements of computer science
https://www.elementsofcomputerscience.com/posts/building-plugin-architecture-with-mef-03/

[11] [13] language agnostic - How to design extensible software (plugin architecture)? - Stack Overflow
https://stackoverflow.com/questions/323202/how-to-design-extensible-software-plugin-architecture

[14] Plugin architecture
https://xi-editor.io/docs/plugin.html

[15] [16] [17] [23] [24] Plugins in Rust: The Technologies | NullDeref
https://nullderef.com/blog/plugin-tech/

[18] [19] Fearless Concurrency - The Rust Programming Language
https://doc.rust-lang.org/book/ch16-00-concurrency.html

[21] quote - Rust
https://docs.rs/quote/latest/quote/

[22] tree_sitter - Rust
https://docs.rs/tree-sitter/latest/tree_sitter/