ChatGPT

SOP v2: LLM-Guided Rewriting of the Campfire Codebase to Idiomatic Rust

Purpose and Scope

This Standard Operating Procedure (SOP) outlines a **team process for using AI coding agents (LLMs)** to rewrite Basecamp's **Campfire** (Ruby on Rails) codebase into idiomatic Rust. The goal is a **full-module, high-fidelity transformation** that preserves functionality while leveraging Rust's strengths: **compile-time safety, performance, and maintainability**. We emphasize a **layered approach** mapping Rails components to Rust's **L1 (core/no_std), L2 (std), L3 (ecosystem)** layers [1] [2]. This ensures that low-level logic uses Rust's core safely (L1), general functionality uses the standard library (L2), and high-level features leverage vetted crates (L3). By focusing on the community's **"vital 20%" idioms** at each layer, the team can achieve **99% of the code with minimal bugs and compile-first success** [3].

Scope: This SOP covers the end-to-end process – from planning the Rust architecture, through iterative LLM-assisted code generation for each module, to rigorous validation and integration testing. It addresses rewriting **web services** (Rails controllers -> Axum/Tokio), **database models** (ActiveRecord -> Rust ORM/SQL), **background jobs** (ActiveJob -> async tasks or services), and **CLI utilities** (Rake/bin scripts -> Rust CLI with Clap). The SOP assumes the team has access to the Campfire Ruby codebase and an internal **idiomatic-archive** – a knowledge base of Rust best practices – to guide the LLMs. All steps are designed to produce **idiomatic Rust** code that is **compile-first correct** (i.e. it compiles with minimal iterations) [4] and exhibits Rust's renowned reliability.

Background: Layered Idiomatic Approach

Rewriting a Rails application in Rust requires **rethinking architecture in layers**. We apply insights from our research on Rust idioms [5] [2]:

- **L1 (Core, `no_std`):** Focus on Rust's core language features independent of OS. Here idiomatic patterns include leveraging the type system aggressively (e.g. **Option/Result** instead of nil or exceptions, making invalid states unrepresentable) and avoiding unchecked behaviors. For example, **use `Result<T,E>` for operations that could fail** instead of Ruby's exceptions, and model optional data with `Option<T>` [6]. **No global mutable state** is allowed in this layer – thread local or passed-down state is preferred [7]. *Anti-patterns:* cloning data just to satisfy the borrow checker (instead, adjust lifetimes or use references) [6], or writing unbounded unsafe code. Aim for **0 unsafe code** at L1; if absolutely needed, isolate it and thoroughly document it [8].

- **L2 (Standard Library):** Use Rust's `std` for common functionality and data structures. Embrace idioms like **RAII for resource management**, iterators and closures instead of manual loops, borrowing (`&str`, `&[T]`) for function inputs instead of copying data, and thread-safe sharing via `Arc<Mutex<T>>` when needed [9]. Utilize `?` operator for error propagation and implement the

`From`/`Into` traits to streamline conversions (instead of ad-hoc converters) [10] [11]. *Anti-patterns:* using `.unwrap()` or `.expect()` in production code (Rust's equivalent of ignoring errors) – instead handle the error or propagate it. Do **not disable compiler warnings**; heed Clippy lints and warnings, as they often flag non-idiomatic code [12]. Avoid mutable global variables (if configuration or state must be global, use safe singletons like `once_cell` or `lazy_static` carefully) [7].

- **L3 (Ecosystem & Frameworks):** Leverage external crates for high-level needs. For web services, idiomatic Rust favors **async frameworks** like **Tokio** (for runtime) and **Axum** (for HTTP server) – these provide proven patterns for performance and reliability [5]. For example, **use Axum's extractor pattern** to handle request parsing and dependency injection via function parameters (mirroring Rails controllers but in a type-safe way). Use **Serde** for JSON serialization (instead of Rails's to_json) via `#[derive(Serialize, Deserialize)]`. For CLI tools, use **Clap** to handle argument parsing and subcommands idiomatically (replacing Rails' Thor or rake tasks). For database access, consider an ORM or query builder like **Diesel** (which offers compile-time query validation by checking SQL against the schema) [13] or **SQLx** (with compile-time checked queries [14]). For background jobs or scheduling, use async tasks with structured concurrency (Tokio's `JoinSet` for managing task lifecycles) and schedule using crates (e.g. `tokio_cron_scheduler`) if needed. A key idiom in async services: **never perform blocking IO or long CPU work on the async runtime** – use `tokio::task::spawn_blocking` for CPU-bound tasks to prevent event-loop hangs [15]. Also enforce timeouts on external calls and use backpressure strategies to handle load (e.g. bounded channels). *Anti-patterns:* mixing multiple async runtimes (stick to one, typically Tokio, to avoid compatibility issues) [16], reinventing what a well-tested crate already provides, or neglecting security best practices (e.g. use crates like `argon2` for password hashing instead of a custom implementation).

Idiomatic Patterns Emphasis: The team's idiomatic-archive contains the **20% of patterns** that yield the most benefit [3] – these will guide our rewrite. Examples include using **builder patterns** for complex config instead of giant parameter lists, leveraging **trait abstractions** to replace Ruby's duck-typing, and ensuring **"invalid states are unrepresentable"** by designing Rust types with strict invariants [17] [18]. By heeding these idioms and avoiding known anti-patterns, the new Rust codebase should compile with far fewer iterations and run with dramatically fewer runtime bugs [4].

Prerequisites and Tools

Team & Knowledge: Assemble a team with understanding of the Campfire domain (chat application features) and familiarity with Rust. Ensure everyone has read this SOP and the high-level system requirements (rooms, messages, file uploads, notifications, etc.). Have the **Campfire Rails codebase** accessible for reference. Also ensure the **Idiomatic-Archive** of Rust patterns is available (as a document or integrated tool) – this will be crucial for prompting LLMs and reviewing code.

Environment Setup:
- **Rust Toolchain:** Install latest stable Rust (via rustup) on all dev machines or a central CI. Ensure `cargo` is available. - **Project Initialization:** Create a new Rust workspace for Campfire rewrite. For example, a Cargo workspace with sub-crates: - `campfire-core` (L1/L2 logic, no external deps if possible), - `campfire-web` (Axum/Tokio web service for HTTP API and web UI, L3), - `campfire-cli` (for any CLI/admin tasks using Clap, L3), - possibly `campfire-db` (if separating data models with Diesel or similar).

Each crate can map to a layer or subsystem. This separation is optional but helps enforce layer boundaries (e.g., core logic crate does not depend on web framework). If the app remains small, a single crate with modules for each concern is also acceptable – but logically separate layers via module organization. - **Dependencies:** Add chosen crates to `Cargo.toml` for each crate. For example, in `campfire-web` add `axum`, `tokio`, `tower`, `serde`, `serde_json`, etc.; in `campfire-core` perhaps no external deps or only those that are no_std compatible if aiming for embedded usage; in `campfire-cli` add `clap` and `anyhow` for error handling if needed; in `campfire-db` add `diesel` or `sqlx` plus the database driver. Run `cargo fetch` to ensure all crates are downloaded. - **LLM Access:** Set up the environment for interacting with the LLM coding assistant. This could be via an IDE plugin (like Cursor, VSCode Copilot Labs), a command-line tool, or a web UI. Ensure the LLM is a capable model (GPT-4 or equivalent) that can handle code and follow complex instructions. **Configure a system prompt** (or few-shot examples) to imbue the LLM with our context: the importance of Rust idioms, the patterns from the idiomatic-archive, and a directive to produce **compilable, well-structured Rust code**. For example, a system prompt might include: *"You are a Rust expert assistant. Follow the organization's idiomatic-archive guidelines: no unsafe unless necessary, no* `.unwrap()` *in production code, use iterators, etc. You will be given Ruby code and must return equivalent Rust code using Axum/Tokio (for web) or appropriate crates, writing in idiomatic Rust style."* Keep this prompt ready. - **Automation Tools:** Install any supporting tools: - **Clippy:** `rustup component add clippy` for lint checks. - **Rustfmt:** `rustup component add rustfmt` to auto-format output code (the LLM's code should be formatted, but this ensures consistency). - **Testing Framework:** We can use Rust's built-in test harness. Also consider setting up basic test cases (perhaps port some Rails tests from the `test/` directory to Rust `#[test]` functions) to validate functionality as we proceed. - **Continuous Integration (CI):** (Optional but recommended) Set up a CI pipeline that automatically runs `cargo check`, `cargo fmt`, `cargo clippy`, and `cargo test`. This provides quick feedback on each module as it's rewritten. A nightly `no_std` check for core crate (if we target that) can enforce that layer's purity [19].

Idiomatic-Archive Integration: If the archive of idioms is available in a queryable format (documents or a Q&A bot), have it accessible. The team may use it in two ways: (1) **Upfront in prompts** – include relevant idiom guidelines when asking the LLM to generate code, and (2) **Post-generation audit** – use the archive (or a secondary LLM prompt) to review the output for any anti-patterns. Ensure the latest idioms (post-2021 developments) are included, such as using `anyhow::Result` or thiserror for error handling in application code, using `Arc<Notify>` or channels for notifying background tasks, etc., as relevant.

Architectural Mapping: Rails to Rust Equivalents

One of the first tasks is to **map Rails constructs to Rust constructs**. Use the following guide to decide how each part of Campfire's architecture will be implemented in Rust:

- **Models (ActiveRecord ORM) → Rust Data + ORM/DB:** Campfire's data models (e.g. User, Room, Message, Upload, etc.) should become Rust structs with fields. Use an ORM like **Diesel** or a query crate like **SQLx** for database interaction. With Diesel, define schema in Rust (via `table!` macros or Diesel's CLI) so that queries are checked at compile time [14]. Each model struct can `#[derive(Identifiable, Queryable, Insertable)]` etc., and methods on these structs (equivalent to Rails model methods) become Rust functions implementing business logic. *Rails to Rust tip:* Ruby's dynamic finders and scopes should be converted to explicit query functions or Diesel DSL chains. **Example:** a Rails scope `Message.recent(limit)` might become a Rust function `Message::recent(conn: &PgConnection, limit: i64) -> QueryResult<Vec<Message>>`

using Diesel's `.limit(limit)` and returning a Result. Also plan how to handle **ActiveRecord callbacks** (before_save, etc.): in Rust, you might call equivalent logic explicitly in service functions or use constructor functions that enforce those checks (Rust doesn't have AR-style lifecycle hooks by default). Ensure all model invariants are encoded (e.g. if Rails validates presence of a field, enforce in Rust by option type or constructor that returns Result if validation fails).

- **Controllers & Routes → Axum Services:** Rails controllers and routes map to Axum route handlers and endpoints. In Rails, routes (config/routes.rb) direct HTTP verbs + URLs to controller actions. In Axum, define a router (using `axum::Router`) with `.route("/rooms", get(list_rooms).post(create_room))` etc., mapping to async handler functions. Each **controller action becomes an async Rust function** that receives `axum::extract::State<AppState>` (for shared state like DB pool), request parameters (as `Path<...>` or `Query<...>` extractors), and possibly JSON bodies (`Json<T>` extractor), and returns `Result<impl IntoResponse, HttpErrorType>`. Use Axum's response types to return either HTML (for web UI, maybe via a template engine) or JSON (for API responses). If Campfire has WebSocket or live update features (e.g. chat messages streaming), use **Tokio + Axum's WebSocket support** or Server-Sent Events as appropriate. *Rails to Rust tip:* Rails filters (before_action) can be modeled via Axum **middleware** (e.g. require authentication on certain routes) [20]. Use Tower HTTP middleware for cross-cutting concerns (logging, auth checks). Also, where Rails might use concerns or helpers, Rust can use traits or plain functions imported where needed.

- **Views (ERB templates) → Templating or Frontend:** If Campfire's UI is server-rendered (ERB/Haml templates), you have two choices:

- **Server-Side Rendering in Rust:** Use a templating crate (like **Askama** or **Tera**) to generate HTML on the server, similar to Rails views. This keeps the project closer to the Rails approach. For instance, an ERB template for a chat room could become an Askama template struct implementing `Template` trait, populated in the handler and rendered to HTML.

- **Client-Side Application:** Alternatively, you could separate the frontend (e.g. a Single Page App or static files) and have Rust provide a JSON API. Given Campfire is a full product with web UI, the simpler path is to use server-side templates in Rust to avoid rewriting the entire front-end. Decide this up front. If using templating, set up the chosen crate and ensure templates compile (Askama templates are checked at compile time, fitting our compile-first goal). If static files (the `public/` folder in Rails), those can be served with Axum's static file service (e.g. `tower_http::services::ServeDir`).

- **Background Jobs → Async Tasks / Services:** Campfire's background jobs (e.g. for sending notifications, processing uploads, clearing caches) in Rails might be using ActiveJob with a queue (maybe Sidekiq/Redis or asynchronous job threads). In Rust, for a single-instance deployment, you can handle these as internal **asynchronous tasks** spawned via Tokio. For example, upon certain events, spawn a task to send a Web Push notification in the background so the HTTP response isn't delayed. Use `tokio::spawn` for lightweight tasks or `spawn_blocking` for heavy CPU tasks (like image thumbnail generation) [15]. If scheduling recurring tasks (cron jobs like daily cleanup), use a scheduler crate or OS cron invoking a CLI command. Rust doesn't have a built-in background job runner like Sidekiq, but you can use work-stealing via Tokio or a library like `async-channel` with worker tasks. Ensure proper error handling and logging for these tasks, since they run detached from any request (consider using `tracing` crate for logging). If needed for scale, these could also be split into separate microservices later, but initially a single binary is fine (Tokio can run web server and background tasks together). We will incorporate these in the `campfire-web` Axum service (spawn tasks on startup or on demand).

- **Caching Layer:** Rails caching (likely used for fragments, or Russian-doll caching of views, etc.) might have utilized an in-memory store or Redis. For Rust, identify what needs caching: e.g., recently active rooms list, or rendered message HTML, etc. If needed, use an in-memory cache (e.g. `dashmap` or `cached` crate) protected by a mutex or built on an atomic structure. This can reside in the application state (AppState struct) shared via Axum's `State`. For distributed caching or persistent caches, integrate with Redis using a crate (like `redis`) similar to how Rails might use Redis. This is an L3 concern – prefer standard crates rather than writing cache logic from scratch. Ensure thread-safe usage and avoid global mutable state (put caches inside Arc<Mutex<_>> if global, or use lock-free structures carefully).

- **File Storage (ActiveStorage) → Rust File I/O or S3:** Campfire supports file attachments. In the Docker deploy it mounts a volume for `/rails/storage` [21], implying files are stored on disk. In Rust, static file serving can be done via Axum routing (e.g. serve `/attachments/*filepath` from a directory). Use `std::fs` or `tokio::fs` for file operations. If images need thumbnails, use an image processing crate (e.g. `image`). This code will mostly be L2 (std I/O) and L3 (external image crate). Ensure to propagate errors (no silent failures). If the design should later support cloud storage, abstract file storage behind a trait so that a local filesystem implementation can be swapped with an S3-backed one (using `rusoto` or `aws-sdk-s3` crate).

- **Notifications (Web Push & Email):** Campfire uses Web Push for notifications (with VAPID keys) [22]. In Rust, use a crate like **web-push** or **vapid** to send Web Push notifications by constructing the payload and VAPID signature. This likely sits in a background task (triggered when a message mentions a user, etc.). Similarly, if any email notifications (password reset, etc.), use an email crate (e.g. `lettre`). These external integrations are L3 – utilize crates and follow their idioms (e.g. reuse an HTTP client for push if needed, respect async). Secure sensitive data: e.g. if storing private keys in memory, consider using `zeroize` crate to zero them out after use for security [23].

- **Command-Line Utilities (bin/ & Rake tasks) → Rust CLI:** Rails apps often have scripts in `bin/` (setup, server, etc.) and Rake tasks for maintenance (e.g. `create-vapid-key` as mentioned [22]). These should become Rust CLI tools using **Clap** (derive or builder API) for argument parsing and help messages. We might combine multiple admin functions into one binary with subcommands (similar to how `rails` command has subcommands). For example, a `campfire-admin` CLI with subcommands: `create-vapid-key`, `migrate-db`, etc. Clap's derive macros make this straightforward and idiomatic (e.g. define an enum for subcommands and annotate with `#[derive(Parser)]` [24]). Ensure these tools share code with the main application where appropriate (e.g. database connection setup code, config loading) – don't duplicate logic. Use Rust's structured error handling for CLI (e.g. return `Result<(), anyhow::Error>` from main and let the library print errors nicely).

- **Configuration & Environment:** Rails uses YAML and ENV variables for config. In Rust, use environment variables (accessible via `std::env`) for secrets and options like `SSL_DOMAIN`, `SECRET_KEY_BASE` [25]. You can use the **config** crate or **dotenv** to load `.env` files if convenient. Maintain .env.example similar to Rails. For secrets like SECRET_KEY_BASE, use Rust's `ring` or `rand` to generate cryptographically secure random values if needed, and store them similarly. The goal is to have parity in configuration flexibility.

Decision Checklist – Rails to Rust Mapping: (Use this when designing each feature)
- [ ] **Data Model**: For each Rails model, decided on Rust struct + DB mapping (Diesel vs SQLx? Or even in-memory if small scale?).
- [ ] **Associations**: If Rails has associations (has_many, etc.), map to Rust: either use Diesel's associations feature or handle via queries returning child objects. Ensure foreign keys are represented as Rust types (e.g. use `uuid` or `i64` for ID types consistently).
- [ ] **Controller Action**: Mapped to an Axum handler (noted what input/outputs it will have). Prepared any Axum extension (like extracting user auth from cookies – maybe use axum_extra for sessions or JWTs).
- [ ] **View/Template**: If needed, decided which templating approach. If API only, decided JSON output structure (and created corresponding response structs).
- [ ] **Background Task**: Identified any background jobs and how to implement (Tokio task, separate service, etc.).
- [ ] **External Service**: Mapped any integration (push, email, etc.) to a Rust crate and checked how to use it idiomatically.
- [ ] **CLI**: Listed any script or rake task to reimplement. Combined them logically into one or few binaries with Clap.
- [ ] **Config**: Ensured we know how each Rails config or initializer translates (e.g. Rails secrets -> .env, Rails configuration (like caching engine, mailer settings) -> constants or config files in Rust).

By completing the above mapping, you create a **blueprint** so that when using LLMs for code, you know what target pattern to instruct for each piece (e.g. if a controller deals with a model, the LLM should use Diesel queries inside an Axum handler function returning `Result<Json<T>, ApiError>`).

Procedure: LLM-Assisted Module Rewriting

1. Planning and Setup (Project Kick-off)

1.1 Initialize Rust Workspace: Using the above mapping, create the Rust project structure (as per *Prerequisites*). For each major component (core, web, etc.), create scaffolding code. For example, create stub structs for models, stub Axum routes and handlers returning e.g. `todo!()`, and stub out main binaries. This scaffolding will serve as a target for LLM to fill in. Commit this initial structure to version control. Ensure the project builds (even if functionality is stubbed) – e.g. `cargo check` passes with empty functions. This verifies your Cargo setup and crate boundaries are okay.

1.2 Align Team on Idioms: Before coding, review the key idioms from the archive that will govern this rewrite: - Use of `Result` and `Option` for error handling and absence of data (no exceptions or nulls) [26]. - Avoid global mutable state; use dependency injection (passing references) or concurrency primitives [7]. - Favor iterators (`iter()/map/filter`) over indexed loops for clarity and safety. - Prefer **safe code**: no `unsafe` blocks unless absolutely required (and none expected for a web app). - Leverage Rust's ownership to prevent data races (e.g. using `Send + Sync` bounds for shared state, using `Arc` for shared ownership). - Ensure *compile-first correctness*: if something can be checked at compile time (types, lifetimes), use that approach rather than runtime checks [4]. (E.g., if a function should only take valid room IDs, consider using a newtype `RoomId` that is guaranteed to be valid, instead of a plain integer that you constantly validate at runtime.)

All team members and the LLM agent should follow these principles. It may help to create a **"Rust Idiom Checklist"** document (one-pager) summarizing do's and don'ts, drawn from the idiomatic-archive, to use during code reviews and to feed into the LLM.

1.3 Prepare LLM Prompts: Formulate how you will prompt the LLM for each task. Typically, for each Rails module or feature: - Provide a **concise summary** of what the code does (if it's lengthy or if domain context is needed). E.g., "This is the controller for chat rooms, it lists messages and allows posting new messages with optional image attachments." This gives the LLM context beyond just code text. - Provide the **source Ruby code** (or the relevant excerpt) in the prompt, if it's not too large. For bigger files, you might prompt iteratively (e.g., model first, then each controller action separately). - Include any **mapping guidance** from the earlier step. For example: "Rewrite the above in Rust using Axum. Use Diesel for database calls. The equivalent of `Room.find(params[:id])` is a Diesel query like `rooms::table.find(id)` returning a `Room` struct. Use `Result` for error handling instead of exceptions. The function should return an Axum `Json` response or appropriate error." Essentially, *you give the LLM the recipe* for how to map this piece into Rust. - Include **idiomatic hints**: e.g. "Follow Rust best practices: no .unwrap(), use `?` to propagate errors, ensure the code compiles without modifications. Use meaningful Rust naming conventions and include comments where logic might be non-obvious." The idiomatic-archive can be a source for these hints. For instance, if the Ruby code uses a lot of mutable state, remind the LLM to "minimize mutability – use immutable bindings and functional style where possible, only use mut when needed (idiom of temporary mutability) [27]."

For complex modules, consider a **step-by-step prompting**: first ask the LLM to outline the Rust structure (types and functions needed), then ask it to write the code. This ensures it has a correct high-level design before filling in details.

2. Module-by-Module Conversion (Iterative Process)

Repeat the following sub-steps for each module or logical unit of the Campfire application (e.g. a pair of model + controller for a feature, or a standalone component):

2.1 Feed and Instruct the LLM: Provide the prepared prompt for the current module. For example, for the **Room controller & model**: - System prompt: *"You are a Rust expert assisting in code migration. Adhere to our idioms (no unsafe, no unwrap, use `Result`, etc.) and target Axum for web logic and Diesel for DB."* - User prompt: include Ruby code (model and controller code, truncated if needed) and instructions: *"Convert this to an idiomatic Rust module. Create a Rust `Room` struct (with fields corresponding to ActiveRecord fields) and impl block for any methods. Create Axum handler functions for each controller action (index, show, create, etc.), using Diesel to load/save data. Ensure proper error handling (if record not found, return 404), and use `serde` to serialize to JSON. Include any necessary struct definitions (e.g. input DTOs for creating a room)."*

Be as specific as possible so the LLM knows the expectations (it should output complete Rust functions/structs ready to compile). If the Ruby code uses Rails helpers or concerns, explain their effect so the LLM can incorporate that logic (for instance, if there's a `before_action :authorize` in Rails, instruct the

LLM to assume a middleware or a guard check is needed in Rust, possibly just note a TODO or call an auth check function).

2.2 LLM Generates Rust Code: Allow the LLM to produce the Rust code. It should ideally produce an entire module (one or multiple functions and struct definitions). For a web controller, this might be multiple functions; for a model, the struct and impl; for a background job, perhaps a function or a `tokio::spawn` call in a setup function. **Ensure the output includes necessary `use` statements or module definitions** – sometimes the LLM might omit use declarations if not prompted. You can preempt this by including a snippet: *"Remember to include* `use axum::{Json, extract::State, http::StatusCode}; use diesel::prelude::*;` *etc."* in the system prompt or editing the output after.

2.3 Review for Completeness: Manually or with another LLM pass, review the generated code for any obvious missing pieces or misunderstandings: - Does every piece of functionality in the Ruby code have a counterpart in the Rust output? (E.g., if Rails code sends an email in a callback, did the Rust code account for that, perhaps by calling a notification function?) - Are all data fields handled? (If Ruby had `@room.name`, ensure Rust struct has `name` and it's used). - Are edge cases considered? (If Ruby code had a branch for when something is nil or empty, Rust code should handle Option or empty list accordingly). - Check that the code respects layering: low-level logic should not directly call high-level constructs. For example, the `campfire-core` crate (if you have one) shouldn't be spawning tokio tasks or doing HTTP – that belongs in `campfire-web`. If the LLM intermixes concerns, plan to refactor accordingly (possibly guiding the LLM to separate concerns).

If issues are found, **prompt the LLM again with feedback**. For instance: *"The code is missing the error handling for when a room is not found. Please update the* `get_room` *handler to return* `StatusCode::NOT_FOUND` *if the ID is invalid."* The iterative nature of LLM use means you refine prompts until the module code is solid.

2.4 Compile and Test Immediately: Save the LLM output into the Rust project (into the appropriate module file). Run `cargo check` (or `cargo build`) on the project. **Resolve compile errors**: - If there are errors (very likely on first try), read them and decide whether to fix manually or loop back to the LLM: - *Minor fixes* (e.g. a wrong function signature or a missing import) can be quicker to do by hand. However, consider feeding the error back to the LLM to correct its model of the code. For instance: *"Compiler error: no method* `execute` *on Option – it seems the code tried to use* `execute` *on an Option. Fix this by unwrapping the Option or handling None properly."* This teaches the LLM about Rust's actual compiler feedback, reinforcing compile-first correctness. - If errors are numerous or indicate a misunderstanding, supply the LLM with the error list plus context and ask for corrections. The goal is to achieve **zero compile errors**. This might take a few cycles, but each fix should be faster than writing from scratch. - Once it compiles, run `cargo clippy --all -- -D warnings`. This will catch warnings and lint issues (treat warnings as errors to force addressing them). Typical clippy suggestions might be to remove unused imports, or use `&str` instead of `String` in certain places, etc. Apply these suggestions. You can again use LLM to fix clippy warnings: *"Refactor the code to resolve these Clippy warnings: [paste warnings]."* This ensures the code is not just correct but idiomatic (Clippy is effectively encoding a lot of Rust idioms and anti-pattern checks).

2.5 Validate Behavior (Basic Tests): If possible, run any available tests for that module. If the original project had tests (Campfire has a `test/` directory [28]), consider porting a couple of critical tests to Rust or at least doing an ad-hoc test: - For example, if you converted the authentication logic, try starting the Rust server (if feasible) and hitting an endpoint, or write a quick `#[tokio::test]` that calls the handler function to see if it returns expected output. - Use known sample data (you might have seed data from Rails `db/seeds.rb` or fixtures) to test the function of the code. This manual testing can be as simple as calling a function in `cargo run` or a quick unit test. - If any test fails or the behavior seems off (logic bug), fix it. Prefer fixing via LLM by describing the bug and desired behavior, unless it's a trivial one-liner.

2.6 Code Quality & Idiomatic Audit: With the module now compiling and functionally correct, do a final quality sweep: - **Idiomatic style check:** Cross-check the code against the idiomatic-archive or the checklist. Ensure it uses appropriate idioms: e.g., if a function can return a `Result` rather than panicking on error, it should; if some logic could be expressed more cleanly (using iterator chain vs. manual loop with push), consider refactoring it now. The team's Rust experts or a "review" LLM prompt can help: *"Review the following Rust code for idiomatic style and suggest improvements."* - Remove any leftover `TODO` or unimplemented sections by implementing them or creating follow-up tasks. If something was marked to revisit (like complex business logic you temporarily simplified), address it before moving on. - Check for any uses of **anti-patterns** noted in archive: e.g., unnecessary clones (use Clippy's `needless_clone` lint as guide – if any `.clone()` can be removed by reborrowing, do it), usage of `Mutex` where an atomic or channel might be better, etc. One common thing: ensure error handling is done via `Result` and not by simply logging and ignoring errors (Rust encourages bubbling up errors to a central handler, e.g., Axum can use a custom error type that implements `IntoResponse`). If the LLM output just logs and continues (mimicking a rescue in Rails), refactor to proper error returns. - **Performance check:** Though full optimization is out of scope, glance for obvious issues. For instance, cloning large structs repeatedly or doing O(n^2) operations that could be O(n). Use efficient data types from std (Vec, HashMap, etc., which are akin to Ruby arrays and hashes, but ensure they are used appropriately with capacity reserved if size is known, etc.). If the LLM wrote something weird (like using recursion where loop would be clearer), consider adjusting to a more idiomatic approach.

After this audit, run `cargo test` and `cargo clippy` one more time. The module is now considered **migrated and verified**.

2.7 Commit the Module: Check the diff and commit the changes to version control. Write a commit message referencing the module and noting any deviations (e.g. "Rewrite Rooms controller and model in Rust (using Axum & Diesel) – passes basic tests, uses compile-time checks for DB lookups. Note: implemented simple auth guard as middleware (will refine later)."). This documentation will help later reviewers understand how things were addressed.

Repeat **2.1 – 2.7** for each module or feature in a logical order. A recommended order is: **Models first** (so you have structs ready), then **controllers/services**, and **auxiliary tasks** (background jobs, CLI) last. This way, the model types are available when writing controller handlers, etc. You might do it feature-by-feature (model + controller together) which can also work since the LLM can use them in one prompt. Ensure you update any shared code (like if multiple controllers use the same utility function, maybe factor that into `campfire-core` and generate it once).

3. Integration and System Testing

After all major pieces are rewritten and compiled individually, it's time to test the **whole system integration**:

3.1 Wiring Everything Together: Make sure your `main.rs` (for the Axum web server) is set up to initialize the app: - Set up global state (e.g. database connection pool via `sqlx::PgPool` or Diesel's ConnectionManager and r2d2 pool, caches, config from env). - Build the Axum `Router` combining all routes. This might have been partially done during module conversion; ensure routes from all controllers are mounted (e.g. `let app = Router::new().nest("/rooms", rooms_routes).nest("/users", users_routes)...` etc.). - Start the server (`axum::Server::bind(...).serve(app.into_make_service())`) on the configured port, possibly with TLS if implementing SSL directly (or you run behind a proxy; for local dev just use HTTP). - If background tasks are internal, ensure they are spawned (for example, if there's a job to purge old messages every hour, spawn a Tokio task with an interval). - For CLI tools, ensure they compile to separate binaries (`cargo build --bin campfire-admin`, etc.) and test one by one (e.g. run `campfire-admin create-vapid-key` to see if it generates keys properly).

3.2 Comprehensive Testing: Now test the application end-to-end: - Run `cargo test` if you wrote integration tests. Otherwise, do manual testing: - Start the server (`cargo run`) and exercise key functionalities. Create a few users, create chat rooms, send messages, upload files, search, etc. This may involve using an HTTP client (like `curl` or a REST client) if there's no UI, or load the webpage in a browser if you implemented templates. You might use the Rails app's API as a reference for expected behaviors and output. - If the app has an API (Campfire's README mentions a bot API [29]), test those endpoints as well (maybe with sample requests). - Check background processes: e.g., if you send a message that should trigger a notification, verify that the notification sending function was called (maybe by logs or by using a stub push service in dev). - Performance sanity check: with the server running, perhaps simulate a quick load (e.g. post 100 messages) to ensure the Rust code handles it without panics or slowdowns. Rust should be quite fast, but this can reveal any remaining heavy operations on the wrong thread, etc.

Any issues discovered (panics, incorrect logic, crashes) need fixes: - If it's a minor code fix, do it and add a test if possible. - If it's a design flaw (e.g. you realize a certain Rails feature wasn't ported), decide if it's in scope or should be a follow-up. Since our goal is a faithful rewrite, try to cover everything.

3.3 Security and Robustness Check: Before considering the rewrite complete, do a pass for security: - Ensure input validation is done (Rust makes buffer overflows unlikely, but logic-level validation is needed: e.g. check string lengths, ensure proper authorization checks on each route, etc.). If Rails had validations or filters, make sure none were skipped. - Use Rust's type system to our advantage: e.g. if certain actions required an admin user, represent that as a different type or an enum so that you can't call admin-only logic without an admin flag at compile time if possible. This might be a future improvement, but note it. - Memory safety should be fine if no unsafe, but if any unsafe code exists (perhaps for FFI or special cases), audit that it upholds Rust's safety contracts.

Finally, update documentation: Write a new README or update it to explain how to run the Rust version, any differences in configuration, etc. Document known differences or limitations if any.

4. Governance and Continuous Improvement

With the codebase now in Rust, establish practices to **govern the idiomatic quality** going forward: - **Code Review:** All team members should review the migrated code, not just for correctness but for idiomatic style adherence. Use the idiomatic-archive as a reference during reviews – e.g., if someone notices a piece of code could use a known pattern from the archive, refactor it. This keeps the code aligned with evolving best practices. - **Add to Idiomatic-Archive:** Any new patterns or lessons from this rewrite should be fed back into the archive. For instance, if the team devised a neat pattern to handle user sessions in Axum or a trait to abstract permissions (something not directly copied from elsewhere), record it as an idiom for future projects. Also note any pitfalls encountered (e.g. “LLM tended to use `.clone()` too liberally, we should add a guideline to avoid needless clones”). - **Automated Checks:** Consider adding automated idiomatic checks – Clippy covers a lot, but custom linters or even an LLM-based code auditor could be integrated. For example, run a daily or CI job that uses an LLM prompt to scan the code for any anti-pattern (a bit experimental, but possible given the archive). - **Performance Monitoring:** Once in production, monitor performance and memory. Rust likely improves performance, but watch out for any pathological cases (like blocking calls that were missed or large memory use if data isn’t freed). Optimize using idiomatic techniques (e.g. use `async` properly, use streaming responders for large data, etc.) as needed. - **Team Upskilling:** Ensure that all team members understand the new Rust codebase. Host internal sessions to explain how Rails concepts map to Rust code now, using this rewrite as a reference architecture. This will also help in future work where LLMs might be used – the more the team knows idiomatic Rust, the better they can prompt and evaluate LLM output.

Finally, treat this SOP as a living document. Collect feedback from the team on what worked well or what was challenging in using the LLM for code conversion. Update the SOP for a future “v3” as needed, possibly including more refined prompt strategies or tools.

Checklists and Decision Flows

Pre-Migration Checklist: *(Before writing any Rust code)*
- [ ] **Frameworks Chosen:** Decided on Rust equivalents (Axum, Tokio, Diesel/SQLx, Clap, etc.) and added to `Cargo.toml`.
- [ ] **Project Scaffolding:** Rust workspace and crates initialized; basic module files created.
- [ ] **Idiomatic Archive Ready:** Key idioms and anti-patterns compiled for quick reference (cheat sheet for team and for LLM prompts).
- [ ] **LLM Configured:** Access to a capable LLM, with a system prompt loaded with our guidelines. Few-shot examples prepared if needed (e.g., show it a simple Ruby method and an idiomatic Rust translation).
- [ ] **Team Alignment:** All developers understand the layering approach and target architecture (perhaps a brief kickoff meeting done).

Module Conversion Checklist: *(For each module/feature migrated)*
- [ ] **Ruby Code Understood:** Read and understand the purpose of the module (and relationships to others).
- [ ] **Mapping Decided:** Knew what Rust constructs to use (structs, functions, crates) for this module.
- [ ] **LLM Prompt Prepared:** Included code and clear instructions, with relevant idiom reminders.
- [ ] **Rust Code Generated:** LLM output obtained and saved.
- [ ] **Compile Check:** Code compiles (`cargo check` passes).
- [ ] **Clippy Clean:** No Clippy warnings (or they have been addressed).
- [ ] **Tests/Basic Run:** Module logic tested (via unit test or manual call) and matches expected behavior.
- [ ] **Idiomatic Review:** Code reviewed for idioms (no anti-patterns like naked unwraps, unnecessary mutability, etc., per archive).
- [ ] **Documentation:** Module/function comments updated (especially if behavior differs from the Ruby version or requires explaining tricky logic).
- [ ] **Commit Done:** Changes committed with clear message.

Post-Migration Integration Checklist:
- [ ] **All Routes Registered:** Axum router includes all endpoints (compare with Rails routes to ensure none missing).
- [ ] **Startup OK:** Application starts without panic, connects to DB, etc. (tested `cargo run`).
- [ ] **Basic UI/API Test:** Key endpoints manually tested (or via test scripts) – create user, login, send message, etc., all working as expected.
- [ ] **Background Jobs Running:** Any periodic or async tasks are observed to run (for testing, perhaps include logs to confirm execution).
- [ ] **Resource Cleanup:** No obvious resource leaks (file descriptors closing, tasks finishing, etc.). Use `tokio::task::JoinSet` or drop handlers if needed to manage long-running tasks gracefully [30].
- [ ] **Security Checks:** Authentication & authorization enforced where needed (no route left unprotected that was protected in Rails), sensitive data handling (encryption, not logging secrets) is in place.
- [ ] **Performance Checks:** The app handles expected load for a single instance (if feasible to test; otherwise plan a staging test).
- [ ] **Docs Updated:** README or deployment docs updated for Rust version (how to build, env vars needed, differences from Rails if any).

Decision Flow Highlights: (Use these rules when making design choices)

- **Database Access:** *If* needing to choose between direct SQL vs ORM – choose ORM (Diesel) for compile-time safety [13], unless performance testing dictates otherwise. *If* complex queries not easily expressed in Diesel, consider SQLx with compile-time SQL checking or raw SQL with caution.
- **Web Framework:** *If* an endpoint is simple (JSON in/out), implement as an Axum handler returning `Result<Json<_>, _>`. *If* the endpoint requires streaming (e.g. sending large files), use Axum's streaming response (e.g. `BytesStream`). *If* template rendering is needed, integrate the template engine and return `Html<String>` or similar.
- **Async vs Blocking:** *If* some logic is CPU-bound or does blocking I/O (e.g. image processing, large file operations), do not run in the core async task – wrap it in `spawn_blocking` [15] to keep the reactor thread free.
- **State Management:** *If* multiple handlers need common state (DB pool, cache), use Axum's `State<AppState>` extractor to pass an Arc of state. *Avoid* global statics for this (Rust makes it hard to use them for good reason).
- **Error Handling:** *If* an operation can fail (DB query, file IO), have the function return a `Result` with a domain-specific error (implement `From` for upstream errors to convert, or use anyhow for quick prototypes) [6]. *If* the error should be user-facing (e.g. "room not found"), map it to an HTTP response with appropriate status (404). *If* an error is truly unrecoverable, it's okay to panic, but those should be extremely rare (Rust idiom: panic only on irrecoverable bugs, not on expected error cases).

- **Testing any tricky logic:** *If* a piece of logic was complex in Ruby (say a method with lots of conditional branches), make sure to write a couple of unit tests for the Rust version to ensure equivalence. LLM translations might subtly miss a branch, so tests guard against that.

- **Performance vs Idioms:** *If* you encounter a situation where idiomatic code seems to cause a performance issue (e.g. using a certain crate leads to slow compile or runtime), evaluate trade-offs. In rare cases, a less idiomatic approach might be justified for performance, but measure and document this. Generally, trust that idiomatic Rust is designed for both safety and speed; premature optimization by deviating from idioms is discouraged unless backed by data.

By following this SOP's structured approach – planning with architecture mapping, using LLMs in a controlled loop (prompt → generate → validate → refine), and enforcing idiomatic patterns at every step – the team can successfully transform the Campfire application into a robust, idiomatic Rust codebase. This process not only yields a safer and faster product but also enriches the team's own **idiomatic-archive** for future projects, creating a virtuous cycle of learning and improvement [31] [32]. The end result should be a Rust implementation of Campfire that **maintains feature parity with the Rails version** while demonstrating the benefits of Rust's paradigm: fewer runtime errors (many classes of bugs caught at compile time) and efficient use of resources, all implemented with clean, idiomatic code that will be easier to maintain in the long run.

---

[1] [3] [4] [5] [6] [7] [8] [9] [10] [11] [12] [13] [16] [17] [18] [26] [27] Exploring Rust in Layers_ Language Core to Idiomatic Patterns.docx.md
file://file-BcBe8Lhob4CYq3wtD8V1YM

[2] [15] [19] [23] [30] [31] Unlocking _Compile-First Success__ A Layered Blueprint for Building and Governing Rust's Idiomatic-Archive.md
file://file-L7NV6EuY11nPSwUeZ2zP3r

[14] diesel - Rust
https://docs.diesel.rs/1.4.x/diesel/index.html

[20] Axum Hello World - Fullstack Rust - Rust Basel
https://rust-basel.ch/fullstack-rust-workshop/04-axum_hello_world.html

[21] [22] [25] [28] [29] GitHub - basecamp/once-campfire
https://github.com/basecamp/once-campfire

[24] Clap - unable to map input - help - Rust Users Forum
https://users.rust-lang.org/t/clap-unable-to-map-input/72673

[32] AI Coding Agents Are Removing Programming Language Barriers | Rails at Scale
https://railsatscale.com/2025-07-19-ai-coding-agents-are-removing-programming-language-barriers/