ChatGPT

Next Steps for a One-Shot Correct Campfire Rewrite (TDD-Driven Architecture)

After completing the high-level architecture (Option 5: *UI-Complete, Files-Disabled MVP*) and comprehensive specs, the next phase is to **bridge design to implementation** in a way that maximizes correctness. The guiding principles now are **test-first development, compile-time guarantees, and simplicity** to minimize bugs. Below is an opinionated 2–3 step plan outlining what to do next, with concrete examples of prototyping, stubbing, and module handling. A summary table of TDD checkpoints is included to clarify which components to **freeze** (fully implement and finalize) versus **stub** (temporary or minimal implementation) at each stage.

Step 1: Finalize the Technical Design and Scaffolding (Compile-First Correctness)

Solidify module interfaces and set up a "walking skeleton" of the app before writing any real logic. Use the completed architecture spec to create a minimal codebase that **compiles cleanly** and reflects your planned structure. This means generating all major modules/files (up to the ~50 file limit) with function signatures, type definitions, and struct stubs – but implementations can be `todo!()` or simple returns. The goal is to have the entire project's outline in place so that the compiler can catch integration issues upfront. This leverages Rust's *"compile-first"* philosophy: by designing strong types and boundaries now, many bugs become *impossible* to represent in code [1] [2]. For example, define domain types like `UserId`, `RoomId` as newtypes to prevent ID mix-ups at compile time [1], and use enums for states (e.g. a `MessageState` enum for pending/sent/failed) so that invalid states are unrepresentable [2].

Concrete actions in this step:

- **Generate the project structure and module files** according to your design. For instance, if your architecture doc outlines a `models/`, `handlers/`, `services/`, etc., scaffold those directories and files now [3] [4]. Each file should have the module declarations and function signatures that match the spec (e.g. a `MessageService::create_message(&self, ...)->Result<Message, Error>` in `message_service.rs`, a basic `handlers::messages` API function, etc.). Keep each file under 500 lines as mandated [5] – this naturally forces clear separation of concerns and lowers bug surface area by keeping code chunks small and digestible.

- **Stub out implementations with placeholders.** Inside each function, you can use `unimplemented!()` or `todo!()` so that it compiles but panic on use, or return dummy data where feasible. For example, a `Database::get_user(id)` might just return an `Err(Unimplemented)` or a fake user for now. Ensure that function signatures and return types are correct and fully thought out, because you want to *freeze the interface*. After this step, changing a function signature would ripple through the code and cause rework, so invest time now to get it right. This is where you double-check that the data flows make sense (e.g. does the message service

have access to a DB pool? does the WebSocket broadcaster know about room state?). Walk through a couple of use-case scenarios using your interfaces (on paper or pseudo-code) to verify no piece is missing or misdesigned.

- **Enforce anti-coordination constraints in design.** Make sure none of your scaffolding violates the strict simplicity rules from the spec. For instance, ensure you have **no coordinator or event bus objects** – your design should use direct calls and simple async tasks only [6]. The skeleton should reflect a straightforward 3-layer monolith (DB → API → WebSocket) [7]. If you find yourself creating a complex global state manager or multiple microservices, that's a red flag – revisit the spec's mandate that "if Rails doesn't do it, we don't" [8]. At this stage, also decide on any third-party crates and include them (for example, add Axum, Tokio, Sqlx, etc. in Cargo.toml) so that your project is technically ready to compile and run (even if it would do almost nothing). Including these now and testing a basic compile/link ensures you won't hit surprises later integrating a crate.

How to leverage Kiro in Step 1: Use the Kiro VSCode agent to automate this scaffolding. For example, you can **provide Kiro with your architecture spec** and have it generate the file structure and module templates. Kiro's spec-driven approach excels at turning structured designs into starter code [9]. You might instruct Kiro: *"Using the architecture document, create the Rust module files and include struct definitions and function signatures for each component (models, handlers, services, etc.), without implementing the logic."* Work iteratively – e.g., have Kiro generate one section (like all `models/*.rs`) and review them, then proceed to handlers, etc., to ensure it stays on track. Throughout this, keep *control*: verify names and APIs match your intentions (Kiro will generally follow your spec, but double-check things like naming conventions or lifetimes). Set up an **agent hook for "on save: run cargo check"** [10] so that every time Kiro writes files, the project is compiled to catch any errors immediately. By the end of Step 1, you should have a compiling (but non-functional) codebase that mirrors your intended architecture – essentially a blueprint in code form.

Step 2: Define a Test Strategy and Write High-Level Tests First

With the skeleton in place, the next step is to **turn your requirements into a comprehensive test suite** *before* filling in the implementations. This test-first approach will act as an executable specification, giving you immediate feedback as you start coding and preventing regressions. Focus on covering the critical behavior of each module and the integration of the whole system, guided by your requirements document's acceptance criteria.

Concrete actions in this step:

- **Write happy-path integration tests for core user flows.** Start with a few high-level tests that simulate the main use cases of the system from end to end. For example, you might write a test that *launches the server (in a test mode)*, then does: user signup → login → create a chat room → post a message → verify the message is broadcast to another user in the room. This can be done with something like spinning up an Axum server on a random port (or calling handler functions directly), using WebSocket client and HTTP client to simulate user actions. These tests will initially fail (since nothing is implemented), but they define "what it means for the app to work correctly." They also force you to consider integration details (session cookies, message formats, etc.) now, so you can adjust the design if needed. Ensure that these tests adhere to the MVP scope – e.g., they should *not* attempt file uploads or image previews since those are disabled in the MVP [7] (instead, a message

with an attachment in the test should result in a placeholder response per your spec). Essentially, these tests should align with the "graceful degradation" behavior for files: e.g. asserting that uploading an avatar returns a "feature not available" message rather than success [7].

- **Write unit tests for critical logic in each module.** For each service or model module, identify the key behaviors and write tests for them in isolation. For instance, in `message_service` you might have a test for `create_message` ensuring that an empty message or oversized message is rejected (this ties back to your validation rules), or that a normal message gets saved to the DB and returned properly [11] [12]. For the `auth_service`, test that a correct password login yields a session token and a wrong password yields an error. Use stubs or in-memory substitutes for dependencies to keep these unit tests focused: e.g., use a dummy in-memory database or a **mock database** trait implementation to simulate DB responses [13]. Rust has libraries like `mockall` for this [14], or you can define a simple trait for the DB layer and implement it with a test double. The point is to be able to test the business logic without needing the full infrastructure for every unit test. Also write tests for things like WebSocket broadcaster logic (e.g., if you have a function that takes a message and active connections and produces broadcast events, test that a user who is not in the room doesn't get the message, etc.). By outlining these expected behaviors now, you again might catch design issues early – for example, if writing a test for "presence tracking removes a user on disconnect" is awkward, it might mean your presence module's interface needs tweaking. Adjust the design *now*, before implementations, to make testing straightforward.

- **Prioritize tests for edge cases and constraints.** It's not just the happy paths – consider the tricky scenarios that could hide bugs. Think of cases like: two users join/leave a room rapidly (does presence remain accurate?), a message with special characters or very long content (is it handled or validated correctly?), or hitting the rate-limit (does the middleware block after 10 requests as expected?). Write tests for these according to the spec's requirements (your requirements doc lists many such details like rate limits, content sanitization, etc.). For example, create a test that calls your rate-limiting middleware 11 times in 3 minutes and expect an error on the 11th – this ensures your implementation meets the **security constraint (10 attempts per 3 minutes)** [15]. Similarly, test the "no coordination" constraint in code: e.g., if you have a global state, ensure it's only updated in simple ways. While you can't directly unit-test "absence of complex coordination," you *can* enforce it via code structure (which you did in Step 1) and by code reviews. You might even add a sanity test or script that counts files or checks for forbidden patterns (like searching the code for uses of a message queue library) to assert the architecture stays within bounds (<= 50 files, no Kafka, etc.) [6] [5].

- **Table-driven and documentation-based testing.** Given you have a detailed requirements doc, you can cross-reference each requirement with at least one test. It may help to maintain a checklist mapping each "Acceptance Criteria" item to a test name. That way, you ensure full coverage of the spec and can be confident that if all tests pass, the system meets the documented behavior. This practice will maximize your confidence before writing the production code.

How to leverage Kiro in Step 2: Kiro can significantly accelerate test development. You can **feed Kiro your requirements** (or specific user stories) and ask it to draft test cases. For example, provide an acceptance criterion (like "WHEN file attachments are encountered THEN a placeholder message is created…") and prompt Kiro to generate a Rust integration test for that scenario. Kiro's agent, given the spec context, may produce a good initial version of the test which you can refine [9]. Use Kiro to enumerate edge cases too:

"List some edge conditions for the messaging service based on the spec," then turn those into tests. Additionally, set up an **agent hook to run tests on file save** (similar to Step 1's compile check) [10] – this will give instant feedback as you write tests (initially they will fail) and later as you implement code. Essentially, you're using Kiro as a partner to ensure no requirement is forgotten in your test suite. However, be sure to review and *anchor* each test to the spec – Kiro might need guidance on exact assertions or might oversimplify, so validate that each auto-generated test indeed checks the right thing. Once the critical tests are in place, you have effectively "frozen" the expected behavior of the system in code form. Now you are ready to implement with high confidence.

Step 3: Incremental Implementation with TDD and Agent Assistance

With a robust test suite specifying *what the code should do*, you can now implement the application in small, test-driven increments. The strategy here is to **tackle one feature at a time**, write just enough code to pass the tests for that feature, and lock in each module as "done" (frozen) before moving to the next. This minimizes coordination bugs and rework – by focusing on one area, you avoid entangling unfinished parts and can be sure new code doesn't break existing tests. The end result should be a codebase that passes all tests (hence meeting all spec requirements) in one go, with minimal debugging needed.

Concrete actions in this step:

- **Implement the simplest vertical slice first.** Pick a core vertical slice such as "text message sending in a room" – this involves the database (to save the message), the message service, and the WebSocket broadcaster, plus a bit of the API handler. Start by implementing the bare minimum in each of those modules to get the main message-flow test (from Step 2) passing. For instance, implement `MessageService::create_message`: make it insert a record via your DB layer (you'll implement a stub of the DB layer here too) and then call the broadcaster to fan it out. At first, you might keep things very simple: e.g., no pagination, no @mention logic, just the core send. Run the tests: if the integration test for "send message" now passes, you've confirmed your end-to-end path is correct. Now that these components work together, you can **freeze** their interface and basic behavior. Freezing means you shouldn't need to dramatically change the function signatures or data models anymore – they've been validated in practice. Going forward, any enhancements (like adding optional features or optimizations) should not break the existing contract. Freezing also implies you might do a code review for that segment now: ensure it adheres to idiomatic patterns and has no obvious bugs (e.g. check that you handled errors properly instead of unwrapping [16], used correct lifetimes, etc., as guided by your Rust patterns doc).

- **Progress module by module, guided by tests.** After the initial slice, pick the next set of features – for example, **user authentication** might be next (so that login tests pass), or **presence tracking** (so that your presence tests pass). For each, follow the TDD loop: run the relevant test (it fails), implement or fix the code in the simplest way, run tests again until green. Leverage Rust's compiler heavily during this process: often you can write the function skeleton, and the compiler will tell you where you need to handle results or clone data, etc. Aim to uphold *compile-time guarantees* – e.g., use the type system to prevent misuse as you fill in logic. For authentication, maybe introduce a newtype `HashedPassword(String)` to avoid mixing plain strings by accident (the compiler will then enforce you don't call the wrong function with an unhashed password type). Each time you

finish a feature and tests pass, consider that module **frozen**. For example, once the `auth_service` passes all auth-related tests (valid login, invalid login, session cookie setting, etc.), you lock it in: avoid revisiting it unless a bug is discovered or a later test contradicts it. This containment keeps you from constantly tweaking early modules when you add later ones – a common source of bugs. It also builds confidence: frozen modules have a passing test suite and can be trusted as you build other parts.

- **Stub or defer non-MVP and complex features until last.** Some parts of the system are explicitly out-of-scope or less critical for MVP (recall file uploads, avatars, advanced search scaling, etc., are either disabled or simplified [7]). Keep those as stubs until the end – or even beyond initial release. For instance, your code might already have a `file_storage` module (because you planned for it), but it can remain a stub that always returns "feature not supported" for now. The tests for file-related behavior should expect the "graceful degradation" message, which you can satisfy with a trivial implementation (essentially, those tests passing confirms the stubs work as intended). By stubbing heavy features, you reduce the surface area for bugs in this phase – you're not worrying about image processing or external APIs while you get the core chat working. Similarly, for complex optimizations (like the performance tuning or multi-threading for WebPush notifications), consider stubbing with a simplified approach first. For example, implement push notifications by just logging or printing an info message instead of actually sending – enough to satisfy the test that "a notification attempt is made," but not a full-fledged implementation. You can mark these with TODOs to flesh out later. The key is to avoid getting bogged down in non-essential complexity when validating overall architecture.

- **Continuously run the full test suite and fix issues early.** After each small implementation, run **all tests**, not just the one you're targeting, to catch any unintended side effects. Ideally, if you truly isolated modules and followed the anti-coordination rules, a change in one module (like message logic) shouldn't break another (like auth) – if it does, that's a sign of unexpected coupling that you should address (e.g., too much shared state). Keep an eye on Rust compiler warnings and Clippy suggestions as well; they often flag potential bugs or misuse early (treat warnings as errors to maintain discipline). By the time you get to the last unimplemented feature, most of the code has already been exercised by tests. The final pieces (perhaps things like the search feature or a particular admin function) can then be implemented and tested in the same way.

How to leverage Kiro in Step 3: Now Kiro becomes your pair-programmer for implementation. For each failing test or unimplemented function, you can prompt Kiro to generate the code to make it pass. For example: *"Implement the* `create_message` *function so that it saves the message to the database and broadcasts it to relevant users. Use the existing* `Database` *trait and* `WebSocketBroadcaster`*."* Because Kiro has the context of your spec and the code written so far, it should produce a solution consistent with your architecture (and without disallowed patterns) [17]. Always review Kiro's output – ensure it doesn't introduce anything against your constraints (like spinning up extra threads or using a global state unwisely). If it does, correct the course by reminding it of the constraints (e.g. "avoid using global mutable state, use the passed-in connection manager instead"). Kiro's strength is speed and coverage: it can write boilerplate (like model field getters/setters, error enum implementations, etc.) quickly and suggest edge-case handling you might miss. Use it to also improve code after it's passing tests: you can ask Kiro to refactor a function for clarity or optimize it, and since you have tests, you'll know if the changes break anything. Another effective use is generating **additional tests** on the fly for any new bug that appears – if a test fails or you discover a case you missed, have Kiro draft a new test to capture it, then fix the code. Finally, as you freeze

modules, you can inform Kiro (mark tasks as done in the Kiro task list) so it doesn't keep revisiting them. By integrating Kiro at each micro-iteration (write test -> have Kiro implement -> run tests), you combine AI speed with human judgment, all under the safety net of TDD. The outcome should be very close to "one-shot correct" code: each piece is correct by construction (thanks to the spec-driven tests and compile checks) and the overall system is coherent.

TDD Checkpoints and Module Status

To visualize the plan, here's a breakdown of key TDD milestones and how different modules should be handled (frozen vs. stubbed) at each point:

Checkpoint	Focus/Goal	Modules Frozen (fully implemented)	Modules Stubbed (placeholder or deferred)
1. Skeleton Compiling (After Step 1)	*Project scaffolding set up; compiles without logic.* *(Minimal "hello world" test passes, e.g., health check returns 200).*	**None yet** – no real logic implemented. *(All modules exist with signatures only.)*	**All modules stubbed** – e.g., every service function is `todo!()`. Core structures defined but no behaviors.
2. Core Chat Flow (After Step 3 initial slice)	*Basic chat operations work end-to-end.* *(Users can auth, join room, send/receive text messages.)*	**Core domain frozen:** User/Auth model & service, Room & Message models, MessageService (text only), AuthService (login/logout), minimal RoomService, WebSocket broadcaster, basic HTTP handlers. *(Interfaces and behavior for these are complete and tested.)*	**Ancillary features stubbed:** Notifications (Push/Webhook calls do nothing or log), Search (return basic results or use simple LIKE query), File/Avatar endpoints (return "Not supported" response [7]), any advanced validation or formatting (e.g. emoji parsing stubbed or simplified).
3. MVP Feature Parity (Final before release)	*All MVP features implemented per spec (minus disabled features).* *(All tests green; system meets requirements.)*	**All MVP modules frozen:** All services (Message, Room, Auth, Notification, Webhook) implemented according to spec; Database layer fully working with SQLite; Security middleware (rate limiting, etc.) in place; Frontend integrated (serving React SPA, etc.). *(System is feature-complete for Option 5 MVP [18].)*	**Non-MVP remains stubbed:** File storage & processing remain disabled (modules present but no-ops or upgrade message), Avatar handling disabled, OpenGraph previews off. *(These stubs are acceptable as MVP per spec and can be tackled in future versions.)*

Each checkpoint builds on the previous one. By **Checkpoint 2**, you have a demonstrable product slice (you could actually deploy a test version that allows basic chatting). By **Checkpoint 3**, you achieve the full MVP scope as defined, with high confidence in reliability due to the test suite. At that point, the codebase should be very stable; any further work (like enabling file uploads in a v2) can be done in new branches or phases, using the same TDD + spec-driven method.

Using the Kiro agent at each checkpoint: Kiro can help move from one checkpoint to the next by automating repetitive work and ensuring consistency. In Checkpoint 1, Kiro generates the skeleton. In Checkpoint 2, Kiro can rapidly flesh out the core flow (it might even suggest code for the chat flow based on known patterns) – you still run tests to verify. By Checkpoint 3, as things get more complex, you might use Kiro's *autopilot* or multi-step capabilities to implement several smaller tasks in one go, but always keep it aligned with your spec (use **steering files** or reminders of the anti-coordination rules so it doesn't introduce disallowed patterns [6]). The agent's assistance combined with your test suite creates a tight feedback loop where mistakes rarely survive more than a few minutes – leading to a virtually bug-free first complete build.

Conclusion

In summary, the optimal next steps are to **cement your design in code form, write tests as an executable specification, and then let those tests drive a disciplined implementation**. By scaffolding the architecture upfront and using Rust's compile-time checks, you eliminate whole classes of integration errors before they happen. By writing tests first (and leveraging your thorough spec to do so), you make sure every requirement is accounted for and prevent yourself from coding beyond what's needed. And by implementing in small TDD steps with the help of the Kiro AI agent, you combine human architectural insight with AI's speed, while respecting the project's strict constraints. This approach maximizes reliability and confidence: when the last test turns green, you can be reasonably sure the Campfire Rust rewrite is correct by construction, with minimal debugging needed. You haven't just avoided "premature coding" – you've essentially treated **design, testing, and coding as one integrated process**, ensuring that by the time you write "real" production code, it's already backed by design validation and test proof. Following this plan, you will have a high-quality, one-shot-correct MVP ready to launch, and a solid foundation to build on for future enhancements. Good luck, and enjoy the process of seeing your well-laid plans come to life in code! [17] [19]

---

[1] [2] [11] [12] [13] [14] [16] [19] comprehensive-rust-patterns-guidance.md
file://file-SUTSdM5NZSSeFYARwzUvey

[3] [4] architecture.md
file://file-8b13B8Yz9MUHzn96SvZW3n

[5] [6] [7] [8] requirements.md
file://file-V5YBq256b2Yigq4xJSeVGw

[9] [10] [17] Kiro: The AI IDE for prototype to production
https://kiro.dev/

[15] analysis-progress.md
file://file-BtbPw5hzjg4D3pphNgEV8i

18 architecture-options.md
file://file-MjTficR3BJCcLqe735givJ