ChatGPT

Rewriting Campfire Backend in Rust for Cost Efficiency

Context: *Campfire* is a web-based chat application originally built with Ruby on Rails (RoR). To reduce cloud infrastructure costs, we explore rewriting its backend in Rust. We evaluate two broad approaches:

1. **Native Rust server backends** – using frameworks like **Actix Web**, **Axum**, or **Tide**.
2. **WASM-based backends** – using WebAssembly runtimes like **Fermyon Spin**, **WasmEdge**, or **Lunatic** (an actor-based WASM runtime).

For each approach, we outline the architecture (frameworks, async model, routing, templating, DB, background jobs), how Rails concepts map to Rust, performance (cold starts, memory/CPU footprint), containerization, operational complexity, developer learning curve, and potential cloud cost savings (e.g. on Fly.io, Render, AWS Fargate, AWS Lambda/WASM). Finally, we recommend a cost-efficient, maintainable architecture and suggest innovative ideas for further cost or performance gains.

Option 1: Native Rust Server Backend (Actix Web, Axum, Tide)

Architecture & Frameworks: Modern Rust web frameworks are asynchronous and highly efficient, typically running as a single native executable. All these frameworks support an event-driven, non-blocking I/O model (using runtimes like Tokio or Async-Std) to handle many concurrent requests on a small thread pool. For example, **Actix Web** (built atop the Actix ecosystem) and **Axum** (from the Tokio project) use **Tokio**'s async runtime, whereas **Tide** is built on **Async-Std** [1]. They all provide HTTP/1.x and HTTP/2 support, routing, middleware, and can serve TLS (HTTPS) without an external proxy [2].

- **Actix Web:** A **pragmatic, high-performance** framework once built on an actor model. (Recent versions are largely decoupled from the actor crate – you only need the actor library for WebSockets now [3].) Actix Web uses a **trait-based system** under the hood for async handlers, with a powerful extractor system for request data. Routing is typically configured with macros (e.g. `#[get("/")]` on handler functions) or a builder DSL. It's known for **blazing speed** (Actix ranks among the fastest frameworks in benchmarks [4]) and can handle stateful services via its actor primitives if needed. It can run standalone (no Nginx needed) and supports HTTP/2 and TLS out of the box [2].

- **Axum:** A **modern, ergonomic** framework focused on modularity. It's built on **Hyper** (the low-level HTTP library in Rust) and **Tower** (for middleware), and fully embraces `async/await` for clarity [5]. Axum uses a **router-centric design**: you compose routes and handlers in code (no macros by default) for a clear, macro-free style [6]. Middleware (logging, auth, etc.) are added as *layers* using Tower's system [7]. Axum is considered very developer-friendly, with excellent documentation and community support [8]. It provides extractors for query params, JSON, etc., and easily ties into the broader Tokio ecosystem.

- **Tide:** A **minimalist, pragmatic** framework on Async-Std. It provides only a small API surface [9] – handlers are simple `async fn(Request) -> Result<Response>` with the developer manually extracting query/body data and crafting responses. This lower-level approach gives full control and simplicity at the cost of more manual work for conveniences like form parsing [10]. Tide uses a middleware system inspired by Tower, but leverages the new `async_trait` to make implementing middleware easier [11]. Because Tide is maintained by contributors involved in Rust's async ecosystem, it tends to adopt cutting-edge async features quickly (e.g. async functions in traits) [12]. Tide's design makes it a "playground" for async, but its performance, while respectable, may lag a bit behind Actix/Axum due to the Async-Std runtime's overhead.

Routing & Websockets: All these frameworks let you define RESTful routes easily. Actix uses attributes on handlers or a builder with `App.route()`, Axum uses a builder (`Router::route`) passing async handler functions [13], and Tide uses `app.at("/path").get(handler)` style. They also support **WebSockets** or server-sent events for real-time features (essential for a chat app). For example, Actix has an `actix-web-actors` crate to handle WebSocket connections as actors, Axum can integrate with WebSocket upgrades via Hyper/Tungstenite, and Tide (with some extra setup or the async-std ecosystem) can manage websockets. Rust's ability to handle thousands of socket connections on a single thread efficiently is a strong asset for chat. (For instance, Warp – another Tokio framework – demonstrates an echo WebSocket server with minimal code [14] [15], and similar patterns can be applied in Axum or Actix.)

Templating & Views: In Rails, views are rendered with ERB templates; in Rust, you would integrate a templating engine. Common choices are **Askama** or **Tera**. Askama is a compile-time templating engine (inspired by Jinja2/Django templates) that generates Rust code for templates, yielding very fast rendering and type safety. Tera is a runtime Jinja-like engine (more flexible for template loading/editing, but slightly more overhead). Both can fill the role of Rails ERB/Haml. For example, you might use Askama to create templates for chat room pages, translating Rails view helpers to Rust template functions. These frameworks don't impose a specific templating solution – you add it as a library and render templates in handlers (similar to calling `render` in Rails controllers). Static assets (images, JS, CSS) can be served either by the Rust app (e.g. Actix can serve files from a directory) or via an external CDN or web server.

Database Layer: Rails uses ActiveRecord (an ORM with migrations, schema, and a rich query API). In Rust, there are a few alternatives:

- **Diesel ORM:** A synchronous ORM with a robust DSL for building SQL queries at compile time. It has **compile-time schema enforcement** and prevents SQL injection by construction [16] [17]. Diesel feels similar to ActiveRecord in that you define models (as Rust structs) and can establish associations, but queries are built via methods instead of Ruby DSL. Diesel has built-in migration tooling and is one of the more mature Rust ORMs [18]. By default it is synchronous (blocking), but you can integrate it into async code by running queries in a thread pool or using the experimental `diesel-async` adapters [19]. Many Rust web apps successfully use Diesel with an async runtime by offloading DB ops to blocking threads.

- **SQLx:** A pure async **SQL toolkit** (not exactly an ORM). It lets you write raw SQL or use a lightweight query builder, and it performs **compile-time SQL checks** (it can verify your SQL against the database schema at build time) [20] [21]. SQLx has built-in connection pooling (e.g. `PgPool`) [22]. This is akin to writing raw SQL in Rails (via `ActiveRecord::Base.connection.execute`) but with Rust's safety. SQLx is great for performance and flexibility, especially if you're comfortable with SQL and want full async support.

- **SeaORM:** An async ORM in Rust that is more dynamic (runtime-driven) – somewhat closer to ActiveRecord's feel. It generates Entity structs and allows querying in an ActiveRecord-like fluent API. SeaORM sacrifices some compile-time checks (queries are built at runtime) [23] but can be easier to adopt for those who like ActiveRecord patterns (e.g. model struct methods for relations). It's newer but gaining popularity in 2024-2025.

Mapping ActiveRecord models to Rust: you would define Rust structs for your models (e.g. `User`, `Room`, `Message`), and use Diesel or SeaORM to handle relationships (for instance, a `Message` struct might have a `room_id` and you'd query the `Room` for it, similar to ActiveRecord's `belongs_to` but done explicitly in code). **Migrations** in Diesel are via migration files (SQL or Diesel's DSL) run with a CLI (like Rails' `db:migrate`). SeaORM offers a migration tool as well. Validations (presence, length, etc.) aren't built-in the same way as ActiveRecord, but can be handled with libraries (e.g. `validator` crate) or simply in application logic (Rust's type system can ensure certain invariants too).

Background Jobs & Async Tasks: In Rails, background jobs (e.g. sending notifications, processing uploads) are often done via ActiveJob + Sidekiq/Redis or similar. In a single-process Rust server, you don't have a built-in job queue, but you have options:

- **Async tasks in-process:** Because Rust's async runtime is very efficient, you can often spawn background **tasks (futures)** to handle jobs concurrently. For example, when a message is posted, you might spawn a tokio task to, say, generate an image thumbnail or send a web push notification, so the HTTP response isn't blocked. These tasks run in the same process (on the runtime's thread pool). This is analogous to kicking off a Sidekiq job, except it's in-process. The upside is simplicity (no external queue), but the downside is if the process dies, you lose the task. For critical jobs, an external queue is safer.

- **Dedicated job workers:** You can create a separate Rust service (or separate thread) that acts like Sidekiq. For instance, use a **message queue** or **broker** (Redis, RabbitMQ, NATS, etc.) – Rust has clients like `lapin` (RabbitMQ) or `redis` crate. The web app could enqueue tasks (just like Rails enqueues to Sidekiq), and a background worker binary (same codebase or separate) consumes and executes them. There are also higher-level crates evolving (e.g. `rustyscheduler` or others in the ecosystem) to manage job queues, but it's not as standardized as Sidekiq. Alternatively, one could use **Cron-like scheduling** (e.g. `tokio-cron-scheduler` crate) for periodic tasks inside the app process.

Mapping from Rails: if Campfire used Sidekiq for sending emails or cleaning up files, the Rust rewrite might use a combination of async tasks and perhaps an external queue for reliability. For example, Web Push notifications (which require VAPID keys) could be sent by a tokio task or by publishing a message to a NATS stream that a worker listens to. Rust's memory safety and concurrency allow handling many background jobs without memory bloat, so you might even get away with simpler designs (fewer external moving parts) for moderate load.

Cold Start Performance: A native Rust server is a **single compiled binary** that starts up very quickly – typically in milliseconds to a few hundred milliseconds, depending on initialization. There is no heavy VM or interpreter to load. In contrast, a Rails app (with an interpreter and dozens of gems) can take a significant time to boot. For serverless scenarios (like AWS Lambda), a Rust function's cold start is usually quite low (often ~50-100ms), whereas a Rails function cold start would be far longer (not typically used on Lambda without custom packaging). In a container environment, cold start matters less since the service is long-running, but it means scaling up new instances or deployments is fast with Rust. **Memory footprint on startup** is minimal – on the order of 1–2 MB for Axum or Actix in idle state [24] – and only grows as it caches things or handles load. (For example, an Axum service was measured using ~0.75 MiB at startup [25]!) This is **orders of magnitude** lighter than a Rails Puma worker (which can easily be 50–100 MB just loaded).

Under load, Rust's memory usage stays efficient. In one benchmark, Actix and Axum handling thousands of requests used around ~70 MB RAM at peak [24], then dropped back down, whereas a comparable Rails process might consume significantly more for far less throughput. **CPU usage** is likewise very efficient – Rust can leverage all CPU cores without the GIL limitations. For I/O-bound workloads (like chat, which waits on network and DB), Rust's async runtime handles tens of thousands of concurrent connections with minimal CPU overhead, waking threads only for active events.

Throughput & Efficiency: The performance gain from Rails to Rust is dramatic. As an illustrative example, one team ported a Rails service to Rust (using Actix) and saw a *94% reduction in CPU use and 97% reduction in memory use* for the same workload [26]. In their case, they went from needing 2 vCPUs and 4 GB RAM to handle traffic in Rails, down to **0.25 vCPU and 0.5 GB** in Rust for the same load, an **87% cost reduction** on AWS ECS [27] [28]. This aligns with general observations that moving from an interpreted GC language to optimized Rust can cut compute costs by **75–90%** [29]. Rust's top web frameworks (Actix, Axum) handle **10-12k requests/sec** in hello-world tests on one core, versus Rails which might do a few hundred per second on one core [4]. In real-world "business logic" tests (database, templating, etc.), Rust still outperforms by a wide margin. This means you can serve the same chat app traffic with far fewer servers or smaller instances. For example, one Rust instance might replace a cluster of 5–10 Rails instances, slashing hosting costs.

Containerization & Deployment: Packaging a Rust service is straightforward. You can statically compile the binary (e.g. using MUSL target) and produce a tiny Docker image (just the binary and maybe an empty base like `scratch` or Alpine). It's common to get images <50 MB or even <10 MB in size for a Rust web service, which improves startup and reduces attack surface. By comparison, a Rails image with the Ruby runtime, gems, and OS libraries often is several hundred MB. Running on platforms like **Fly.io** or **Render** is easy – you just deploy the single binary in a container, no special runtime required. Rust's binary runs anywhere (Linux x86_64, etc.), and doesn't need managed language runtime installed on the server.

Operational Complexity & DevEx: The trade-off for Rust's efficiency is a **higher developer learning curve** compared to Rails. Rust is a systems-level language with strict compile-time checks. Developers will need to get comfortable with concepts like ownership/borrowing and lifetimes, which can slow down initial development. Also, Rust web frameworks are less "batteries-included" than Rails. There is no direct equivalent of Rails' one-stop ecosystem (with ActiveRecord, ActionCable, ActiveJob, Devise for auth, etc. all integrated). Instead, you assemble components: choose a web framework, choose a DB crate, etc. This gives flexibility at the cost of writing more boilerplate and making more decisions.

However, Rust frameworks like Axum and Actix are generally well-documented and have growing communities. Many conveniences exist (for example, Axum has middleware and extractor libraries for sessions, auth, etc., similar to Rack middleware in Rails [5]). The lack of Rails-like "magic" means the code is very explicit – which can be positive for maintainability once the team learns Rust. Each request handling is just a Rust function that you can unit test like any other function. The **type safety** catches many bugs (mismatched types, missing fields) at compile time that in Rails might only show up at runtime.

From an operations perspective, a Rust service can be easier to manage: a single binary means fewer moving parts (no interpreter, no separate job runner if you integrate jobs, etc.). Crashes are rare (no null dereferences unless `unsafe` is misused), and memory usage is predictable (no GC pauses or massive heap growth over time). On the other hand, observability and profiling might be a bit more DIY (Rust has tools like `tokio-console`, but you won't have something identical to NewRelic for Rails out-of-the-box). You might incorporate libraries like `tracing` for structured logging and OpenTelemetry for monitoring, which is analogous to using Rails' log tags and APM.

Cloud Cost Estimates: By switching to a native Rust backend, **cloud costs can drop dramatically**. As noted, compute and RAM needs shrink ~5-10x for the same throughput [26]. Concretely, if Campfire on Rails required, say, a 2 CPU / 4 GB VM on Fly.io to handle peak load, the Rust version might need only a 0.5 CPU / 512 MB VM. Many hosting providers bill roughly linearly with resources, so this could cut costs by ~75-90%. On usage-based platforms like AWS Fargate or ECS, the savings are directly proportional to CPU/RAM reduced [27]. On AWS Lambda, a Rust function could be allocated less memory and finish faster than a Rails (Ruby) function, reducing per-invocation cost. There is also savings in **idle cost**: a Rails app often must run multiple worker processes (to leverage CPU cores or threads despite GIL) and keep them alive to handle bursts, whereas a single Rust process might handle everything, or fewer processes needed. Fewer containers/instances also mean lower overhead for orchestration.

Summary (Native Rust): A native Rust rewrite (especially using a framework like Axum or Actix Web) would produce a **monolithic, high-performance backend**. Key Rails components would be replaced by Rust equivalents (Rails controllers → Rust handler functions, ERB → Askama templates, ActiveRecord → Diesel/SQLx, ActionCable → WebSockets handling via Tungstenite or framework support, ActiveJob → tokio tasks or separate job service, ActiveStorage → S3 via `rusoto` or `aws-sdk` crate, etc.). Cold start and baseline memory use would be minimal, and runtime throughput extremely high. This approach maximizes performance and likely yields the **highest raw cost savings** (in terms of resource usage reduction) while using relatively **mature, proven tech**. The main costs are the **development effort and learning curve** for the team, and ensuring the Rust codebase is well-structured (since it won't have Rails' conventions). In terms of maintainability, Rust's strong type system can make the code robust, but the smaller ecosystem means developers might implement some features manually. Overall, if the goal is **cost efficiency with reliable performance**, a Rust backend with a framework like Axum (for its combination of performance and clarity) is a strong choice – it's likely the most cost-efficient in steady-state usage.

Option 2: WebAssembly-Based Backend (Spin, WasmEdge, Lunatic)

WebAssembly on the server-side offers an innovative way to deploy backends with potential cost and scalability benefits. The idea is to compile the backend (or parts of it) to **WASM modules** and run them in a lightweight sandboxed runtime. This can enable **finer-grained scaling** (even per-request), superb isolation, and very fast startup times, often reducing overhead compared to full containers or VMs. We examine three scenarios: **Fermyon Spin**, **WasmEdge**, and **Lunatic** – each representing a different approach to using WASM for a backend.

Fermyon Spin (Serverless WebAssembly): Spin is a framework and runtime for building cloud **functions/ microservices in WebAssembly**. You write your code (e.g. in Rust, using the Spin SDK) and compile to a WASM module, and Spin executes these modules on incoming requests (HTTP triggers, timers, etc.). The architecture is **event-driven** and **stateless** per request – similar to how AWS Lambda functions work, but using WASM for isolation. Key features:

- **Routing/Triggers:** You define routes or triggers in a manifest (`spin.toml`), mapping URL endpoints to WASM module entry points. The Spin runtime uses **WASI (WebAssembly System Interface)** under the hood to allow limited system interactions. When an HTTP request comes in, Spin instantiates (or reuses) a WASM module instance to handle it. Each route can be handled by a separate module or the same module with internal routing logic, depending on design.

- **Async Model:** Spin functions are not exactly long-running async servers; each invocation is short-lived. However, you can perform async I/O inside (the Spin SDK provides async functions for things like making outbound HTTP calls, DB queries, etc., using host-provided futures). The runtime might schedule multiple WASM instances concurrently if requests come in parallel. Importantly, **cold start is extremely fast** – a Spin WASM component can start in **<1 millisecond** [30], which is essentially negligible. This means Spin can rapidly scale from 0 to thousands of instances to meet demand.

- **Templating & Response:** You can still render HTML or JSON in a Spin handler. For example, you might compile Askama templates to WASM and use them inside the Spin module – that's possible as long as the crate supports WASI (many pure Rust crates do). The response from a Spin handler is built and returned to the caller via the Spin runtime (no persistent connection unless using some workaround, because once the function returns, that instance can be freed).

- **Database Access:** Spin follows a **"bring your own database"** approach. It doesn't embed a database, but it provides interfaces to connect to external DBs like PostgreSQL or MySQL [31]. In practice, you use the Spin SDK's database plugin to query the DB (Spin can pool connections to improve efficiency as of Spin 3.4 [32]). So you might have your Campfire data still in Postgres, and each WASM request handler would open (or reuse) a DB connection to load/save chat messages. Because these are short-lived, Spin now even **pools DB connections** for you [32], mitigating overhead.

- **Background Jobs:** Since Spin modules are stateless and ephemeral, background processing needs a different approach. One method is to use **Spin's scheduled triggers** (Spin supports cron-like timers or you could ping a function on a schedule) to perform periodic tasks. Another is to offload long jobs to an external system (e.g. put a message in a queue for a separate worker service). Spin is oriented around quick request-response cycles, so something like broadcasting a chat message to all connected users would likely involve writing the message to the database or a cache, and possibly using WebSockets in another component (WebSockets aren't a native trigger in Spin yet, since maintaining a long-lived connection contradicts the spin-to-zero philosophy – though some creative solutions like server-sent events or polling could be used).

- **Performance & Resource Footprint:** The big advantage here is **scaling to zero** and high density. When no one is using the app, **there are 0 processes running** and effectively no memory used (maybe a tiny overhead for the runtime). When a user sends a chat message or loads a page, a WASM instance starts in ~0.5ms [33] and handles it. This is incredibly fast – Spin can literally start **thousands of instances in milliseconds** on modest hardware [33]. Because the WASM sandbox is lightweight, you can run tens of thousands of these isolated instances on a single VM if needed [33]. For *infrequently used apps*, this means you can pack many services on one server without paying for idle time [34] [35]. Spin's model ensures **memory is freed as soon as a request is done** [36] – so your peak memory usage is only what's needed for concurrent active requests. For *spiky high-traffic apps*, Spin can instantaneously spawn enough instances for each request, then kill them – essentially perfect autoscaling per request [37] [38]. Traditional autoscalers might take minutes to add VMs for spikes [39], but Spin does it in microseconds. This could handle viral traffic surges very cost-effectively (you don't pay for pre-provisioned capacity for the peak, only for actual invocations).

- **Operational Considerations:** Deploying Spin can be done via **Fermyon Cloud** (they offer a managed service with a generous free tier and usage-based pricing [40]) or self-hosted on Kubernetes with **SpinKube** [41]. Docker is integrating WASM support too; you could containerize a Spin runtime (there's a Docker shim to run Spin apps in containers [42] [43]). Containerizing Spin might involve running the `spin` CLI in a container that loads your .wasm modules. In terms of developer experience, Spin requires learning its manifest and the constraint of writing code that runs in a WASM environment. Debugging is improving (there are tools to run and test modules locally), but it's not as straightforward as running a local web server—though Spin's CLI does allow local testing of the HTTP routes. Also, since the state doesn't persist in-memory between requests, you'll lean on external state (DB, cache, etc.) more heavily, which can make the architecture more like a set of pure functions.

- **Cloud Cost:** If Campfire's usage is such that it has long periods of low activity with occasional bursts (e.g. certain hours of the day are quiet), Spin could save a **huge amount of cost by not running idle servers**. Instead of paying for a 24/7 server, you'd pay per request. For example, on Fermyon Cloud or another WASM PaaS, you might only incur costs when users actually send messages or fetch data. Additionally, thanks to high density, you could run many small services (or many tenants of Campfire if it were multi-tenant) on one machine. An article suggests as much as *83% of container costs are often for idle services* [44] – Spin aims to eliminate that waste by keeping services fully dormant until needed. Even for consistently busy workloads, Spin can be cost-efficient by maximizing resource usage efficiency and isolation (though if constantly busy, a native Rust service might perform similarly – Spin shines most when load varies or multi-tenancy is needed).

WasmEdge (WASM Microservice Runtime): WasmEdge is a high-performance WASM runtime designed for cloud native use (it's a CNCF project). The approach with WasmEdge would be to compile the Rust backend to a WASM module (likely a WASI module), and then run it using WasmEdge instead of a normal Linux process or container. In essence, **WasmEdge can run a WASM application as a microservice** with potentially lower overhead than a full container. Key points:

- **Architecture:** Unlike Spin's function-per-request model, with WasmEdge you might run the whole web server as a WASM module. For example, compile an Axum or Tide app to WASM (using WASI target). WasmEdge can then execute that module, managing its threads and I/O via the WASI layer. This is more analogous to just replacing the OS/VM with a WASM sandbox. You'd still have a long-running server inside the WASM module listening on a socket. WasmEdge supports networking, file I/O, and other system calls through WASI, so a Rust web framework can actually operate (with some configuration). In this scenario, you'd deploy perhaps a WasmEdge runtime per service. Think of it as container vs WASM: each Rust service runs in a WasmEdge sandbox instead of a Docker container.

- **Benefits:** The sandbox provides strong security isolation (if you wanted to run untrusted code or multiple tenant code on the same machine, a compromised service can't escape the WASM sandbox easily) [45] [46]. It's also lightweight – no full OS per service, and potentially faster startup than a container (WasmEdge can instantiate modules quickly). WasmEdge is optimized with JIT compilation to run WASM nearly at native speed [47]. It can integrate with container tools (Docker can run WasmEdge via a runtime shim [48] [49]), meaning ops teams could manage WASM services similar to containers. Memory usage is typically lower than a full Linux process because there's no duplicated OS overhead; multiple WASM modules can share the runtime and even memory pages (if using some advanced features).

- **Mapping Components:** If you go this route, you still keep your application structure similar to the native Rust approach (MVC style, same frameworks, etc.), but at compile-time you target WASM. Some library choices may be constrained by WASI support. For instance, Diesel (which uses some OS-specific code) might not run in WASM; you might use a pure Rust client like `surrealdb` or a HTTP-based DB access or Spin's DB interface. However, WasmEdge does support "Networking" so in principle a normal Postgres TCP connection from inside the WASM might work if WASI allows socket APIs. (There's evolving support for WASI sockets). Alternatively, you might pair it with an out-of-process DB proxy or use web APIs.

- **Performance:** WasmEdge and similar runtimes are optimized, but there is some overhead to running in a WASM VM vs native. Typically, code runs at ~near-native speed (within 5-15% of native in many cases) [47], which is still far better than interpreted Ruby. If using JIT, the first few requests might see JIT compilation delay, but then it's hot. **Cold starts** for a WasmEdge service are faster than starting a full OS process (WASM modules are smaller binaries and initialize quicker). If using a platform that can quickly spin up WASM instances (like an orchestrator that pools WasmEdge instances), scaling out can be very fast. Memory footprint per service can be lower – e.g., if the Rust binary was 50 MB RSS natively, the WASM version might be smaller because of shared runtime and possibly more efficient memory allocation for unused parts. There's also potential for **multitenancy**: multiple WASM modules can run in the same host process isolated, which could save overhead if you run multiple Campfire instances for different groups.

- **Operational Complexity:** Adopting WasmEdge means introducing a relatively new layer into your stack. Instead of running a Docker container directly, you either use Docker with WASM support or run WasmEdge manually. Tools are emerging (Docker, containerd support it as mentioned [43]) so it's not too bad. The developer workflow would involve cross-compiling to WASM, which can have its own challenges (ensuring dependencies compile to WASI, etc.). Debugging a WASM module might be slightly trickier (though WasmEdge and Wasmtime have some debugging support). If everything works, the deployment and scaling could be similar to containers – you scale instances of the WASM module as needed, possibly with finer granularity and density. **Ease of containerization**: you might create a Docker image that includes the WasmEdge runtime and your .wasm file, and entrypoint runs WasmEdge with the module. This image could be extremely small (since WasmEdge itself is small and your .wasm might be a few MB). Docker Desktop's experimental WASM support even allows running the .wasm without a heavy base image [50] [51].

- **Cloud Cost:** Running a backend on WasmEdge would save cost by **improving resource density**. You could pack more services per VM (since each uses less memory). If using something like AWS Fargate, you might be able to run more lightweight tasks per dollar. WasmEdge itself doesn't inherently scale to zero like Spin, since it's more akin to a continuous service, but you could combine it with fast startup to scale in/out quickly. For example, you might run it on **AWS Lambda with a custom runtime** – AWS has announced better support for WASM in serverless – which could combine the scale-to-zero benefit with the fast startup of WASM. On Kubernetes, you might run WasmEdge with an operator to dynamically spawn modules. We're essentially cutting out OS overhead, which may yield maybe a 30% or more efficiency gain in memory usage for the same application logic. Also, security isolation from WASM could reduce the need for separate VMs for isolation, potentially consolidating workloads (lower infrastructure footprint).

Lunatic (WASM Actor Model runtime): Lunatic is an **Erlang-inspired actor runtime** that uses WebAssembly under the hood. Its architecture is quite unique and potentially well-suited for a chat application if leveraged correctly:

- **Architecture & Model:** In Lunatic, each lightweight **actor is a separate WebAssembly instance** with its own memory [52]. Actors communicate by message passing (like Erlang processes) and are scheduled by the Lunatic runtime (which uses Tokio and a custom async scheduler) [53]. Crucially, Lunatic can spawn **huge numbers of actors very quickly** – on the order of **hundreds of thousands per second** on modest hardware [52]. Each actor is isolated (memory-safe sandbox) and if it crashes, it doesn't take down others (fault isolation like Erlang's "let it crash" philosophy).

- **Async and Simplicity:** Lunatic provides a **synchronous programming style** to the developer, even though it's asynchronous underneath. You can write blocking calls (e.g., sleep, socket read) in an actor, and Lunatic will transparently turn those into async operations by yielding the actor's execution when blocking occurs [54]. This means you **don't need `async/await` in your code**; it feels like writing normal blocking Rust code, but the runtime ensures no thread is actually blocked (actors get scheduled cooperatively). This can simplify development – no need to propagate async semantics everywhere – while still handling massive concurrency. For example, you could write a function that handles a single chat client connection with straightforward loops and blocking reads/writes, and spawn that as an actor for each client. Lunatic will manage all those actors under the hood, potentially handling millions of connections easily [55].

- **Mapping Campfire to Actors:** You could design the system such that each **user connection** or each **chat room** is an actor. For instance, when a WebSocket connection comes in, spawn an actor to handle that socket; that actor can communicate with a "room actor" representing the chat room to broadcast messages. This is analogous to ActionCable channels but using Lunatic's message passing. Because actors are so cheap, you can have one per user or even one per message if needed. Lunatic also allows setting permissions and limits per actor (e.g., an actor can be forbidden from making outbound network calls, etc., enhancing security for multi-tenant logic) [56]. It's a very powerful model for isolation – even more granular than threads.

- **Performance:** Lunatic's ability to spawn ~300k actors/second and handle millions of concurrent actors means it can scale well beyond typical thread-based systems [52] [55]. The overhead per actor is minimal (much lower than a thread or OS process). For chat, this means you could keep an actor alive for each user's session without worry, and even if you had 100k online users, that might be fine. The memory overhead per actor is the size of a small WASM instance's linear memory (which can be kept small unless the actor needs a lot of data). Also, Lunatic's scheduling ensures that no actor can starve the system – even if an actor runs a tight loop, the runtime will interrupt it to maintain low latency across all actors [57] (since all code is instrumented via WASM, it can force yields in long loops, something not trivial in native threads). This means **latency can be kept low** even with many tasks, a critical factor for chat responsiveness.

- **Integration & State:** Lunatic is still evolving, but it has support for networking (actors can open TCP listeners, etc.), so you could implement an HTTP server actor or integrate with an existing one. It might not have a full-fledged web framework integration out-of-the-box – you may need to use lower-level networking and parse HTTP, or perhaps use a simplified framework that can run in WASM. (Potentially, you could compile something like a tiny HTTP server to WASM and let each request spawn an actor, though this is unorthodox.) More likely, one would build a custom server loop under Lunatic. The **database** access could be tricky: you'd either have to use a WASM-friendly client or perhaps use Lunatic's outbound TCP to talk to the DB. If each actor opened DB connections naively, that'd be too heavy – instead, you might have a **DB connection pool actor** that handles queries for others, or use an approach like that.

- **Operational Complexity:** Lunatic is relatively **bleeding-edge**. It's not as battle-tested as Actix or even Spin. Using it in production would require careful consideration. The Lunatic runtime itself would run as a process (likely you'd deploy it as a Docker container running `lunatic` with your compiled modules). That one process could handle enormous load, but it's a single point of failure (though you could run a cluster of them for redundancy, similar to an Erlang cluster). Developer-wise, Lunatic requires learning a new paradigm (actor model, message passing, supervision trees possibly in the future). It's similar to learning Elixir/Erlang in concept. This could be a big shift for a team used to Rails request-response logic.

- **Cloud Cost:** If Lunatic fulfills its promise, it could let you run your entire application with **unprecedented efficiency**. One Lunatic process might handle what previously required many instances. For example, where Rails might need 10 servers to handle 50k long-polling connections, and Rust might need 2 servers to handle 50k WebSockets, Lunatic might handle 50k (or even 2 million) connections in 1 process [55]. That minimizes infrastructure – potentially one beefy VM instead of a cluster (though for high availability, at least two for failover). CPU usage is kept high but efficient, and memory is carefully managed per actor. With Lunatic's fine scheduling, you squeeze every ounce of performance from the hardware, which translates to cost savings. The model is also **serverless-like in development convenience** – you write simple synchronous code and the runtime scales it – but you still manage the server process yourself. If multi-tenancy is needed, Lunatic's isolation could let you run multiple customer "apps" in one process safely, again saving cost.

Innovative Rewrite Ideas & Hybrids: In addition to the above, one could consider **hybrid architectures** to maximize cost savings and performance:

- **WASM Microservices** – Break the application into small pieces that run as separate WASM functions. For example, one could isolate the search functionality or notification sending into its own Spin function that only runs when needed, while the core chat runs as a Rust server. This way, rarely-used components scale to zero independently. This microservice approach could reduce costs by not having every feature loaded in memory at all times. However, it adds complexity (you'd need to orchestrate calls between services, perhaps via HTTP or a message bus).

- **Edge Computing with WASM** – Deploy parts of the app to edge runtimes like Cloudflare Workers (which use WASM under the hood). For instance, static file serving or simple API calls could be moved to edge functions close to users, reducing load on the core server. Cloudflare Workers have a free tier and extremely low per-request cost, which might reduce cloud costs for global access. The main chat server could remain centralized, but auxiliary functions could be at the edge.

- **Actor-model within native Rust** – Even if not using Lunatic, one can adopt an actor approach in native Rust (using something like Actix actors or the `xtra` crate) to manage chat rooms and connections. This could improve scalability design-wise (easier to reason about state per room), though it doesn't directly cut costs beyond what Rust already does. It can, however, help structure the application for fault tolerance and potentially allow spawning/dropping parts of the system on demand.

- **Serverless DB or Storage** – To pair with a scaled-to-zero compute (like Spin or Lambda), using serverless databases (like DynamoDB or Fauna) or on-demand Postgres (Neon, etc.) could ensure you only pay for storage when needed. This is more about re-architecting the data layer but can contribute to overall cost efficiency if usage is low/variable.

- **Language Interop** – If a full rewrite is daunting, an intermediate step could be to rewrite performance-critical parts (like message broadcasting, or search indexing) as Rust libraries or services and call them from Rails (via FFI or HTTP). This isn't a greenfield rebuild (so slightly off the scenario), but it can yield partial cost and performance improvements. However, since the prompt allows greenfield, a full Rust or WASM solution likely yields a cleaner, more maintainable result than a polyglot system.

Recommendation and Conclusion

Considering **cost efficiency** and **maintainability**, the **native Rust server approach** with a framework like **Axum** (or Actix Web) is likely the best overall solution for a Campfire rewrite. It offers a **drastic reduction in resource usage** (potentially 5-10x less CPU and memory than Rails [26]) which translates directly to cloud cost savings. This approach leverages well-established tooling – the Rust ecosystem in 2025 has mature libraries for web, DB, auth, etc. – making it easier to build and maintain in the long term. Containerization and deployment are straightforward, and the application remains relatively simple to reason about (monolithic, like the Rails app, but more efficient). While the development team must overcome Rust's learning curve, the payoff is a robust, high-performance system that can run on much smaller infrastructure. For example, running the Rust Campfire on a single low-cost VM on Fly.io or a small container on Render could suffice where multiple larger Rails dynos were needed before. This is **cost-efficient and not overly complex to operate**.

That said, **if maximizing cost savings under low or spiky load is the top priority**, a case can be made for a **WASM-based solution like Fermyon Spin**. Spin would shine for a chat app that's not constantly active: it would nearly eliminate idle costs by scaling to zero when no one is chatting, and handle traffic spikes effortlessly by instant-on functions [37]. The complexity here is ensuring the app can be made stateless-per-request or finding a way to manage WebSocket-like needs. Perhaps a hybrid: use Spin for the HTTP API (login, fetching messages, sending messages as HTTP calls) and a lightweight WebSocket gateway (maybe a small Rust service or a durable object concept) for push. This is more **innovative** and would require more engineering effort and risk, but could potentially allow **serverless pricing** (you pay per use, which for intermittent use is unbeatable). If the user base of Campfire is very intermittent or if deploying many isolated instances for different client organizations is needed, Spin or a similar WASM approach could let one machine host dozens of isolated Campfire instances cheaply.

The **Lunatic actor model** is an exciting prospect for maximum performance and scalability (imagine handling millions of concurrent chat connections in one process). However, given its relative youth, it might be a risky choice for a full rewrite unless the team has expertise in that area. It could be worth prototyping – e.g., a Lunatic-based chat server might handle concurrency even better than a traditional async server. But maintainability could suffer if the ecosystem isn't mature (fewer examples, smaller community). If successful, it would yield a system with Erlang-like resilience and Rust-like speed, which is very attractive for real-time applications. In terms of cost, it could minimize the number of servers needed to just one or two, fully utilizing their capacity (which is cost-effective).

Conclusion: For a greenfield rewrite of Campfire prioritizing cost savings, I would **recommend a native Rust backend (Axum or Actix)** as the primary choice, due to its **proven efficiency gains** and relatively straightforward path (translating Rails concepts to Rust with established libraries) for developers. This alone could cut infrastructure costs by around **75% or more** while reliably supporting Campfire's features. Then, as a forward-looking consideration, the team could incrementally explore **WASM-based microservices** for specific components or deploy the Rust service in WASM for additional sandboxing and density benefits. If the usage pattern fits, deploying on a serverless WASM platform (like Spin on Fermyon Cloud) could further reduce costs by eliminating idle resource usage, though with added architectural complexity.

In summary, **Rust's efficiency directly translates to cost savings** – fewer or smaller servers, less memory, and less CPU time [27]. A well-architected Rust backend (with prudent use of async and background tasks) can serve the same workload as Rails at a fraction of the cost, as evidenced by real-world migrations [26]. WebAssembly offers the next level of optimization by optimizing deployment and multi-tenancy, which could be layered in as needed. The recommended approach is to start with the solid foundation of a Rust web service (to immediately gain the bulk of cost reduction), and keep an eye on WASM technologies to gradually incorporate for scaling and cost efficiency in specific areas. This balanced architecture would yield a **cost-efficient, high-performance, and reasonably maintainable** solution for Campfire, positioning it to handle future load with minimal cloud spend.

Sources:

- Basecamp's open-sourced Campfire (Rails) README [58]
- Rust web framework performance and memory benchmarks [24] [59]

- Actix Web documentation (design and capabilities) [3] [2]
- Axum framework overview and features [6] [5]
- Tide framework summary (async-std and minimalist design) [9] [60]
- Diesel vs SQLx (Rust database libraries) comparison [19] [20]
- Ruby to Rust migration cost savings (87% cost reduction example) [26] [28]
- Fermyon Spin and WebAssembly fast startup and density [33] [44]
- Lunatic runtime internals (actors as WASM instances, concurrency) [52] [54]

---

[1] [9] [10] [11] [12] [14] [15] [60] Best Rust Web Frameworks to Use in 2023 | Shuttle
https://www.shuttle.dev/blog/2023/08/23/rust-web-framework-comparison

[2] [3] What is Actix Web | Actix Web
https://actix.rs/docs/whatis/

[4] [26] [27] [28] [29] I saved 87% on compute costs by switching languages
https://worldwithouteng.com/articles/i-saved-87-percent-on-compute-costs-by-switching-languages/

[5] [6] [7] [13] Rust Web Frameworks Compared: Actix vs Axum vs Rocket - DEV Community
https://dev.to/leapcell/rust-web-frameworks-compared-actix-vs-axum-vs-rocket-4bad

[8] [24] [25] [59] Axum vs Actix vs Rocket
https://eternal-search.com/en/axum-vs-actix-vs-rocket

[16] [17] [18] [19] [20] [21] [22] Choosing a Rust Database Crate in 2023: Diesel, SQLx, or Tokio-Postgres?
https://rust-trends.com/posts/database-crates-diesel-sqlx-tokio-postgress/

[23] Compare with Diesel | SeaORM An async & dynamic ORM for Rust
https://www.sea-ql.org/SeaORM/docs/0.5.x/internal-design/diesel/

[30] [33] [34] [35] [36] [37] [38] [39] [41] [44] Lightweight Kubernetes and Wasm is a Perfect Combo
https://www.fermyon.com/blog/lightweight-kubernetes-and-wasm

[31] Relational Database Storage | Spin Docs
https://spinframework.dev/v2/rdbms-storage

[32] Announcing Spin 3.4 - Fermyon
https://www.fermyon.com/blog/announcing-spin-3-4

[40] Fermyon Cloud Pricing
https://www.fermyon.com/pricing

[42] [43] [45] [46] [47] [48] [49] [50] [51] Wasm vs. Docker | Docker
https://www.docker.com/blog/wasm-vs-docker/

[52] [53] [54] [55] [56] [57] Lunatic: Actor based WebAssembly runtime for the backend - DEV Community
https://dev.to/bkolobara/lunatic-actor-based-webassembly-runtime-for-the-backend-36oj

[58] GitHub - basecamp/once-campfire
https://github.com/basecamp/once-campfire