Member-only story

How to Structure a Large-Scale Rust Application Without Losing Your Mind

4 min read · 2 days ago

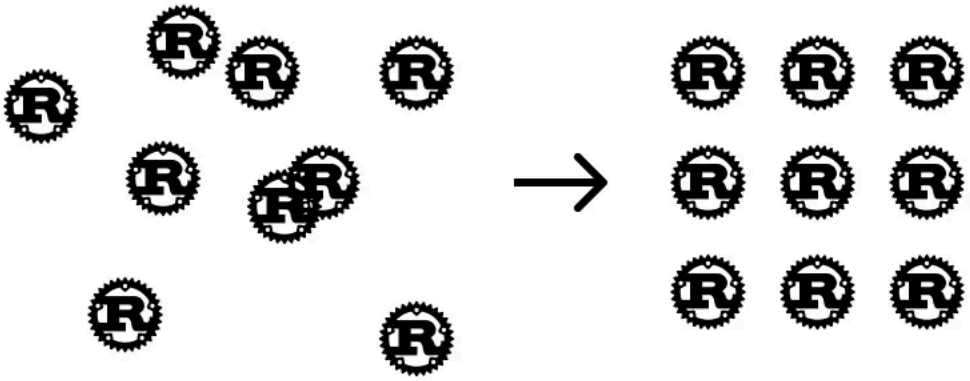
Harishsingh Following

Listen Share More

Rust gives you safety, performance, and control — but when your codebase starts to scale beyond a few thousand lines, you'll quickly face a different kind of complexity: **code organization**.

Whether you're building a service-oriented backend, a CLI tool, or a systems-level application, knowing how to **modularize**, **abstract**, and **scale** a Rust codebase is essential.

This post walks through real-world architecture techniques, crate layouts, and idiomatic patterns that help keep large Rust apps clean and maintainable.



Why Structure Matters in Rust

Unlike many dynamic or GC-based languages, Rust:

- **Forces early decisions** about ownership, lifetimes, and dependencies.
- Encourages a **functional-declarative style** that requires planning abstractions.
- Has **no inheritance**, but offers composition through traits, enums, and modules.

Without a good structure, your project can become a maze of `mod.rs`, cyclic imports, and monolithic logic.

Starting with a Strong Foundation

Let's begin with a base architecture for a large-scale Rust backend:

```
/my_app
├── Cargo.toml
└── src
```

```
    ├── main.rs               --> Entry point
    ├── lib.rs                --> App-wide exports and setup
    ├── config/               --> Config parsing and validation
    ├── api/                  --> HTTP handlers or CLI commands
    ├── services/             --> Business logic
    ├── domain/               --> Core types, enums, interfaces
    ├── db/                   --> Persistence layer (Postgres, Redis etc.)
    ├── errors/               --> Shared error types
    ├── utils/                --> Generic helpers
```

This layout follows the **Layered Architecture** pattern:

- Domain is at the center (pure logic)
- Services apply domain logic
- API and DB layers handle transport and storage

`main.rs` and `lib.rs`: Keeping the Entrypoint Clean

```rust
fn main() -> Result<(), Box<dyn std::error::Error>> {
    my_app::run()
}
```

In `lib.rs`, we wire everything together:

```rust
pub mod config;
pub mod api;
pub mod services;
pub mod db;
pub mod errors;

pub fn run() -> Result<(), Box<dyn std::error::Error>> {
    let cfg = config::load()?;
    let db_pool = db::init(&cfg.db)?;
    api::start_server(cfg, db_pool)
}
```

This separation lets you:

- Test `run()` without launching a binary
- Share `lib.rs` in integration tests or benchmarks

Config Module (with Serde and Envs)

```rust
// src/config/mod.rs
use serde::Deserialize;

#[derive(Debug, Deserialize)]
pub struct Config {
    pub db: Database,
    pub server: Server,
}
#[derive(Debug, Deserialize)]
pub struct Database {
    pub url: String,
}
#[derive(Debug, Deserialize)]
pub struct Server {
    pub port: u16,
}
pub fn load() -> Result<Config, config::ConfigError> {
    let cfg = config::Config::builder()
```

```
        .add_source(config::File::with_name("config/default"))
        .add_source(config::Environment::with_prefix("APP"))
        .build()?;
    cfg.try_deserialize()
}
```

Use `APP_DB__URL=postgres://...` to override nested values via env.

API Layer: Thin Controllers

```
// src/api/mod.rs
use axum::{routing::get, Router};
use crate::services::user::UserService;

pub fn start_server(cfg: crate::config::Config, db: sqlx::PgPool) -> Result<(), hyper::Error> {
    let user_service = UserService::new(db.clone());
    let app = Router::new()
        .route("/health", get(health))
        .route("/users/:id", get(move |id| get_user(id, user_service.clone())));
    axum::Server::bind(&([0, 0, 0, 0], cfg.server.port).into())
        .serve(app.into_make_service())
        .await?;
    Ok(())
}
async fn health() -> &'static str {
    "OK"
}
```

Each route delegates to a **service**, not directly to DB calls.

Domain Layer: Types and Traits

```
// src/domain/user.rs
#[derive(Debug, Clone)]
pub struct User {
    pub id: uuid::Uuid,
    pub email: String,
    pub is_active: bool,
}

pub trait UserRepo {
    fn find_by_id(&self, id: uuid::Uuid) -> Result<User, crate::errors::AppError>;
}
```

The domain layer defines **pure logic** and **contracts**, free from frameworks or database concerns.

Service Layer: Business Logic Lives Here

```
// src/services/user.rs
use crate::domain::user::{UserRepo, User};
use crate::errors::AppError;

#[derive(Clone)]
pub struct UserService<R: UserRepo> {
    repo: R,
}
impl<R: UserRepo> UserService<R> {
    pub fn new(repo: R) -> Self {
        Self { repo }
    }
    pub fn get_user(&self, id: uuid::Uuid) -> Result<User, AppError> {
        let user = self.repo.find_by_id(id)?;
        if !user.is_active {
            Err(AppError::InactiveUser)
        } else {
```

```
            Ok(user)
        }
    }
}
```

The service is generic over any `UserRepo`, making it easy to test.

Database Layer: The Adapter

```
// src/db/user_pg.rs
use crate::domain::user::{User, UserRepo};
use crate::errors::AppError;
use sqlx::PgPool;

pub struct PostgresUserRepo {
    pool: PgPool,
}
impl UserRepo for PostgresUserRepo {
    fn find_by_id(&self, id: uuid::Uuid) -> Result<User, AppError> {
        let rec = sqlx::query_as!(
            User,
            "SELECT id, email, is_active FROM users WHERE id = $1",
            id
        )
        .fetch_one(&self.pool)
        .await?;
        Ok(rec)
    }
}
```

If you later switch to Redis or mock the repo in tests, no service code changes.

Error Handling

Create your app-wide error types once:

```
// src/errors/mod.rs
#[derive(Debug)]
pub enum AppError {
    NotFound,
    InactiveUser,
    DbError(sqlx::Error),
}

impl From<sqlx::Error> for AppError {
    fn from(e: sqlx::Error) -> Self {
        AppError::DbError(e)
    }
}
```

Testing and Benchmarking

Use `#[cfg(test)]` modules inside each file for local tests:

```
#[cfg(test)]
mod tests {
use super::*;

struct MockRepo;
    impl UserRepo for MockRepo {
        fn find_by_id(&self, _: uuid::Uuid) -> Result<User, AppError> {
            Ok(User { id: uuid::Uuid::new_v4(), email: "test@example.com".into(), is_active: true })
        }
    }
    #[test]
    fn returns_user_if_active() {
        let service = UserService::new(MockRepo {});
```

```
        assert!(service.get_user(uuid::Uuid::new_v4()).is_ok());
    }
}
```

Benchmark Example

Using `criterion.rs`:

```
fn fibonacci(n: u64) -> u64 {
    match n {
        0 | 1 => n,
        _ => fibonacci(n - 1) + fibonacci(n - 2),
    }
}

fn bench(c: &mut Criterion) {
    c.bench_function("fib 20", |b| b.iter(|| fibonacci(20)));
}
```

Dependency Management: Workspaces and Crates

If your project has multiple domains or shared libraries, split it into a **workspace**:

```
/project-root
├── Cargo.toml (workspace)
├── core/
│   └── Cargo.toml
├── web/
│   └── Cargo.toml
└── db/
    └── Cargo.toml
```

Workspace-level `Cargo.toml`:

```
[workspace]
members = ["core", "web", "db"]
```

This keeps domain logic, persistence, and delivery isolated.

Diagram: Layered Ownership and Abstraction

```
       [ main.rs ]         --> Application entrypoint
            |
       [ lib.rs ]          --> Wire dependencies, start server
            |
+-----------------------+
|    Application Layer  |
|  (services, handlers) |
+-----------------------+
            |
+-----------------------+
|     Domain Layer      |
|  (types, traits, logic)|
+-----------------------+
            |
+-----------------------+
|  Infrastructure Layer |
| (DB, HTTP, Redis etc.)|
+-----------------------+
```

Closing Thoughts

Rust gives you the tools to write high-performance, safe systems — but with power comes complexity. If you're building something non-trivial:

- Keep your **domain pure and interface-driven**
- Push side effects and infra concerns **to the edges**
- Build around **traits, enums, and modules**, not inheritance
- Start small — but **structure like you're building big**

This structure has helped us scale codebases to **50k+ lines**, onboard new contributors quickly, and maintain rock-solid reliability in production.

Rust | Rust Programming Language | Microservices | Scaling | Tokio

Following

Written by Harishsingh

26 followers · 4 following

I'm a software developer in backend and I love to read and share technological content.

No responses yet

Michaelf

What are your thoughts?

More from the list: "OhRustMyRust"

Curated by Michaelf

In Stackad... by mikelepp...
Track Your Finances with Leptos & Rust — Part 1:...
Jun 17

TheOptimizationKing
10 Rust Async Patterns That Solve Concurrency...
Jun 13

Abhinav
Choose This Architecture, and Rust...
Jun 16

TheOptimizationKing
10 Rust Mistakes I See in Every Codebase...
Jun 16

View list

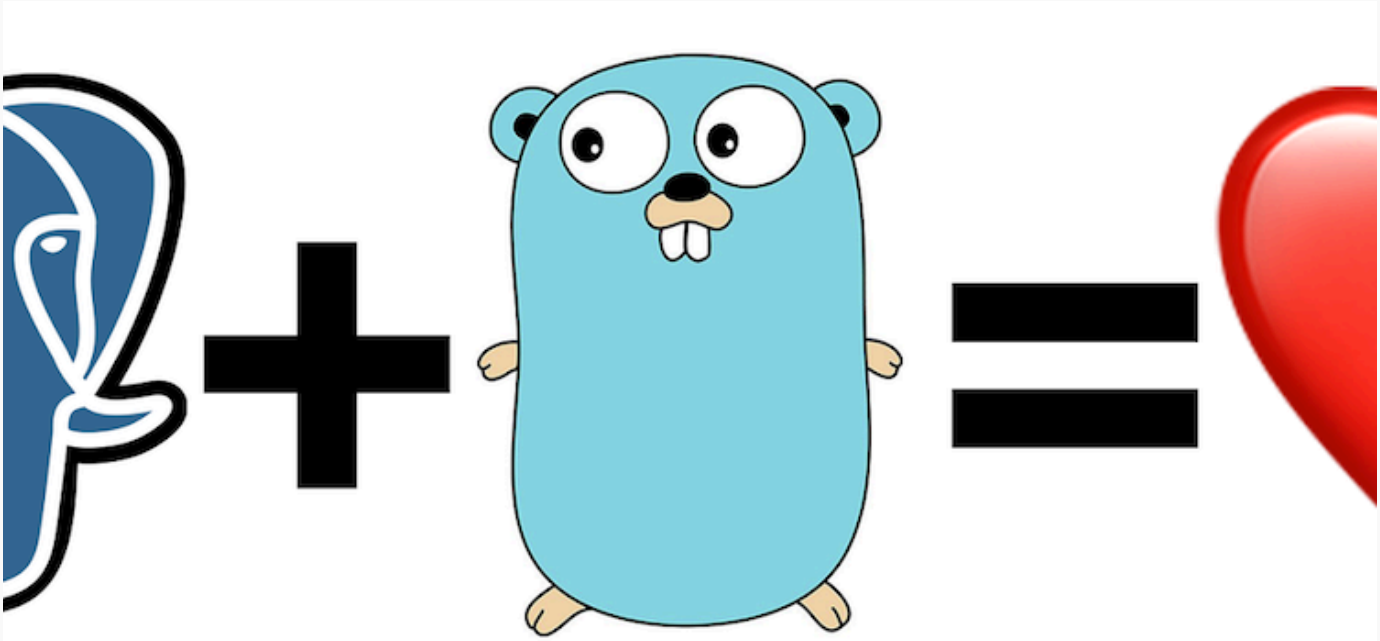
More from Harishsingh

Harishsingh

Zig vs Rust: The Silent War for the Future of Low-Level Programming

For decades, C and C++ have held the crown in low-level systems programming. But in the shadows, a new rivalry brews—one that will shape...

3d ago 11 1

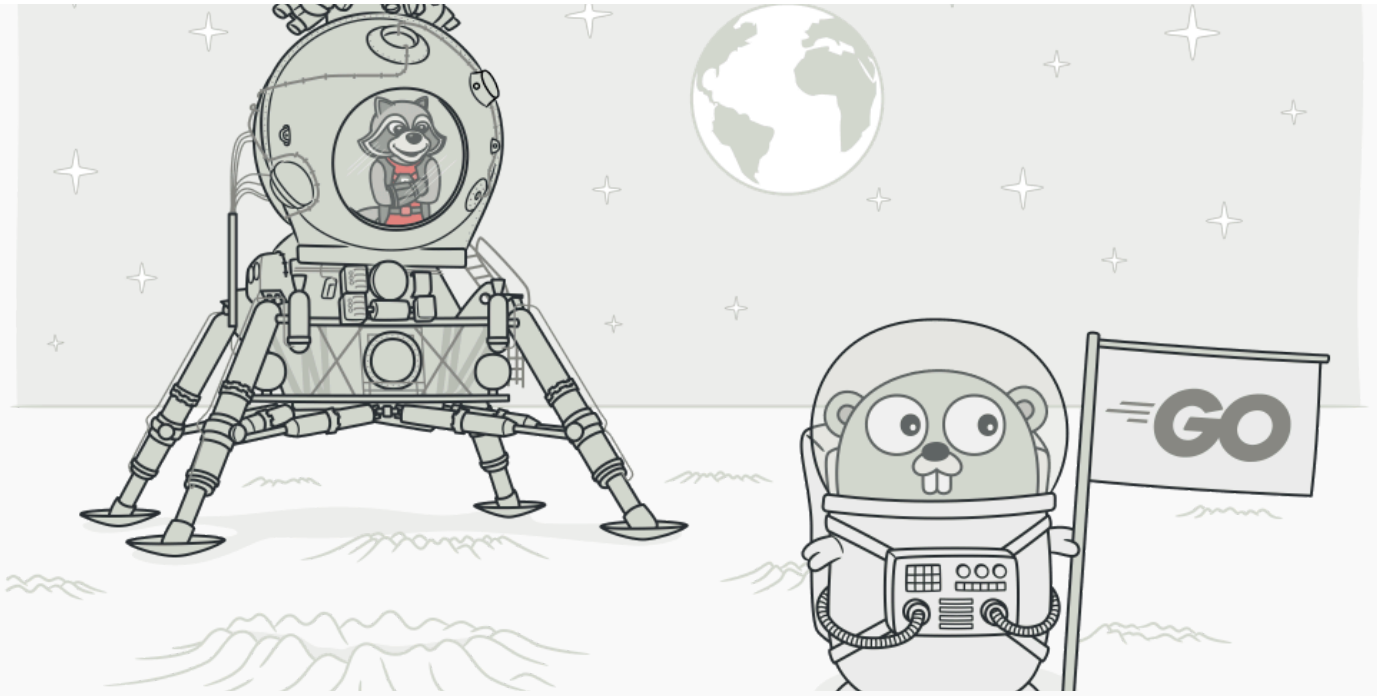


Harishsingh

Go + PostgreSQL: Scaling from 1 Query/Second to 100K Without Breaking a Sweat

When you start with Go and PostgreSQL, everything feels fast. But as usage grows, even the simplest query can become a bottleneck. This...

4d ago 5



Harishsingh

The Only Golang Design Patterns You'll Actually Use in Production

Most design pattern articles are bloated with academic examples and patterns you'll never use beyond a system design interview. This one's...

4d ago 5

Harishsingh

Deep Dive Into Fearless Concurrency in Rust: Async, Channels, and Parallelism

Introduction

Jun 17 10

See all from Harishsingh

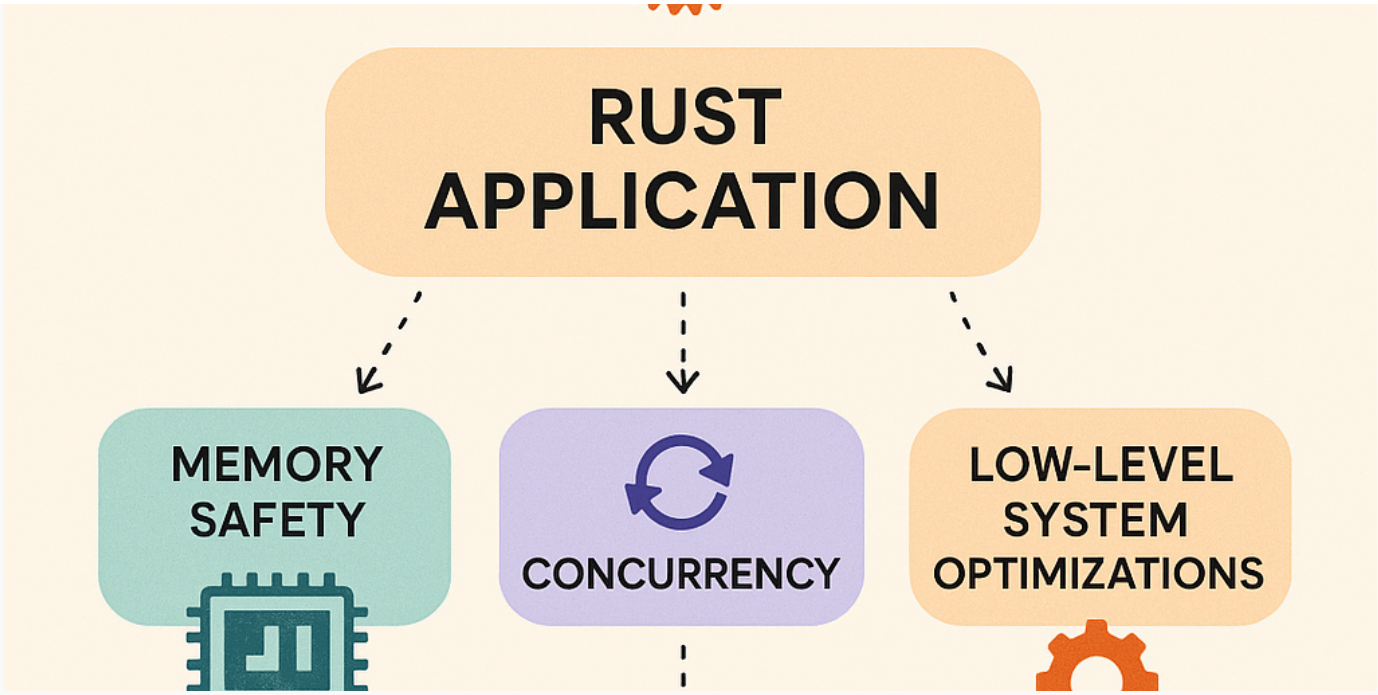
Recommended from Medium

Loïc Labeye

My opinion on Rust's web frameworks evolution over the past years

Alright, let's start with a little on my background, I have been using Rust since early 2020s and it wasn't quite as popular as it is...

Jun 15 14

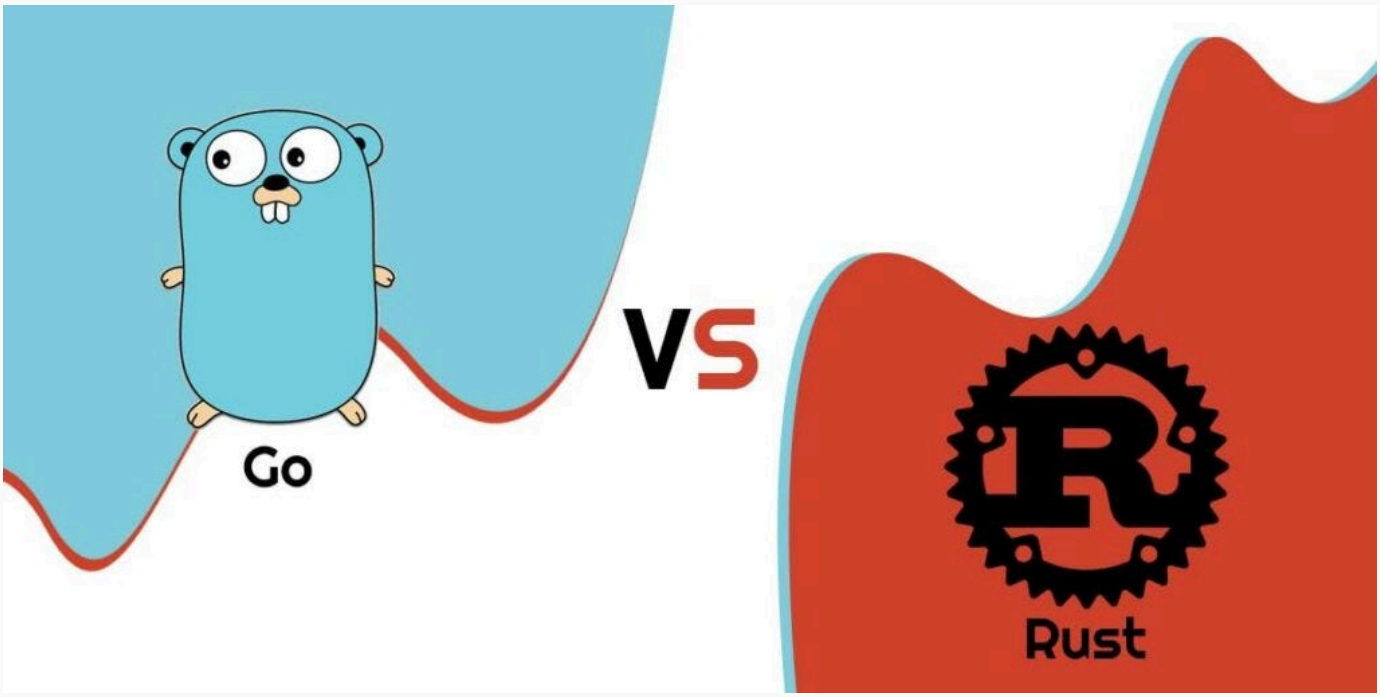


Borelli Fotso

Rust Performance Optimizations Compared to Other Programming Languages

Rust is a modern programming language celebrated for its blend of performance, safety, and concurrency. As a systems programming language...

Jun 20 32 3

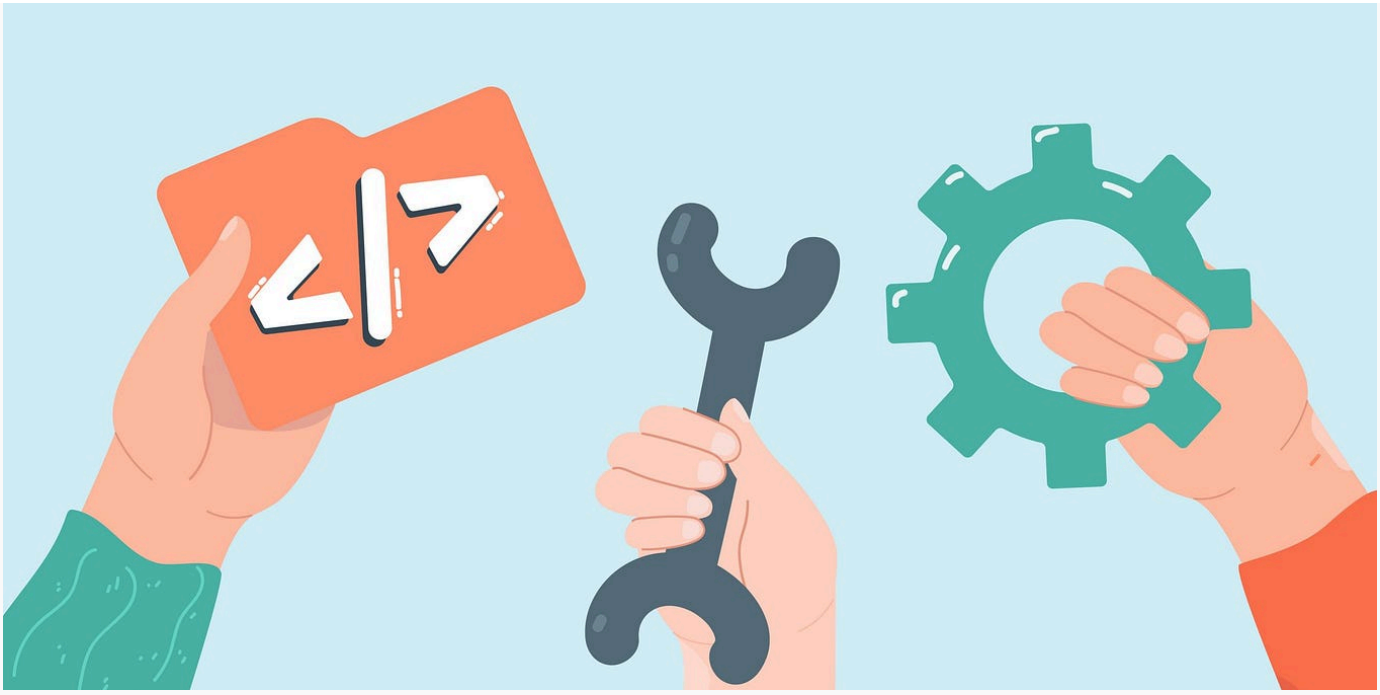


The Quantum Yogi

Go vs. Rust on AWS: The Unexpected Winner When You Care About Money

When we first built our microservices architecture, we reached for Rust. The promise of blazing-fast performance, memory safety, and...

5d ago 80 4



Yash Batra

From Java to Rust: The Great Backend Migration in 6 High-Impact Use Cases

The software world is buzzing with migrations. One of the most controversial yet increasingly adopted shifts is moving backend systems from...

Jun 20 13 3

DevTrendsDaily

Here's What Happened When I Replaced RabbitMQ Consumers with Tokio Streams

I didn't plan to replace our RabbitMQ consumer layer—until it broke under backpressure during a Friday night deployment.

Jun 12 14

AIAlchemist_Ab1r

Rust Debugging Cheatsheet

Debugging Rust applications can be a challenging task, especially when dealing with complex data structures and memory-related issues...

Dec 29, 2024 66

See more recommendations