Bringing Rust to work

Rust sits beside the PHP application you already run. The PHP app keeps the pages, the session, and the database it owns.

A command line tool is the usual first step. CLI and files is that tool: arguments, files, JSON, and a test you can run in CI. A report, a file conversion, or a check that already has a PHP script is enough.

A service behind the PHP app is the next step. PHP keeps the pages. Rust answers the requests you want in a separate process. Building an API is that service.

An in-process option is lighter than a full PHP extension. PHP 7.4+ ships FFI (`ext/ffi`). You build a Rust `cdylib` that exports a `#[unsafe(no_mangle)] pub extern "C" fn` , then call it with `FFI::cdef(...)` . No PHP headers and no libclang. Many PHP teams can try that in an afternoon. Edition 2024 wraps `no_mangle` in `unsafe(...)` because the attribute is an unsafe contract with the linker.

Cargo.toml

```
 [package]
name = "shop_ffi"
version = "0.1.0"
edition = "2024"

[lib]
crate-type = ["cdylib"]
```

src/lib.rs

```
 #[unsafe(no_mangle)]
pub extern "C" fn price_cents(unit_cents: u32, quantity: u32) -> u32 {
    unit_cents.saturating_mul(quantity)
}
```

crate root

```
 $ cargo build --release
```

price.php

```
 <?php

$ffi = FFI::cdef(
    'uint32_t price_cents(uint32_t unit_cents, uint32_t quantity);',
    __DIR__ . '/target/release/libshop_ffi.so', // .dylib on macOS, .dll on
Windows
);

echo $ffi->price_cents(1000, 2), PHP_EOL;
```

That prints `2000`. On macOS the library is `libshop_ffi.dylib`. On Windows it is `shop_ffi.dll`. Confirm the extension is loaded with `php -m | grep FFI`.

This runs from the CLI as-is. The default is `ffi.enable=preload`, so `FFI::cdef()` only works from the CLI and from `opcache.preload` scripts. Under PHP-FPM or Apache `mod_php` it throws `FFI API is restricted by "ffi.enable" configuration directive`. In a web request, declare the bindings in an opcache preload script and use `FFI::scope()`, or set `ffi.enable=true` (not recommended in production).

A proper PHP extension is still the heavier route when you need Zend types, classes, or a PECL-shaped install. ext-php-rs binds the Zend API so a Rust function can be called as a PHP function. The guide exports `hello_world(string $name): string`, prints a PHP stub with `cargo php stubs`, and installs the shared library with `cargo php install`. The build needs PHP headers and libclang on the machine that compiles it.

Shared logic in the browser is the fourth place. WebAssembly runs one pricing function in the page and on the server.

Start with one small, low-risk tool. Measure the PHP version and the Rust version on the same input before you talk about speed. Pair on the first crate. In review, `clippy` with `-D warnings` and `rustfmt` in CI are the bar this course used. A short GitHub Actions job:

.github/workflows/ci.yml

```
 name: ci
on: [push, pull_request]
jobs:
  rust:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v7
```

```
      - uses: dtolnay/rust-toolchain@stable
        with:
          components: rustfmt, clippy
      - run: cargo fmt --check
      - run: cargo clippy --all-targets -- -D warnings
      - run: cargo test --locked
```

The costs are part of the decision. A clean build takes longer than running a PHP script, because Rust compiles. Hiring is harder while fewer people have shipped Rust than PHP. The learning curve is the one you just went through: ownership, errors, and the async runtime are the price of the first useful tool.

Teams that have written up the choice include Firecracker at AWS, Pingora at Cloudflare, and Discord's move from Go. Those posts describe their systems. A Laravel app can stay a Laravel app.

> **Watch for:** A rewrite of the whole PHP application as the first Rust project. The first win is a tool you can delete if it does not help.

> **Idiomatic Rust:** One crate, measured on the same input, reviewed with clippy and rustfmt.

Side by side

Place	PHP keeps	Rust adds	Where this course taught it
CLI tool	the web app	a binary in CI	CLI and files
Service	pages and sessions	an HTTP API	Building an API
In-process (light)	the request	a `cdylib` via FFI	PHP `ext/ffi` here — CLI/preload-only unless configured

Place	PHP keeps	Rust adds	Where this course taught it
Extension	the request	a Zend extension	ext-php-rs, described here
Browser	the page	shared logic	WebAssembly

Key takeaways

- Start with one low-risk tool beside the PHP app, and measure it on the same input.
- PHP FFI plus a Rust `cdylib` is the light in-process try. ext-php-rs is the proper extension route.
- Review with `clippy -D warnings` and `rustfmt` in CI.
- Compile time, hiring, and the learning curve you just went through are part of the decision.

What's next: The Rust ecosystem.