ChatGPT

Managing Optional vs Required Dependencies for Library and Binary Crates

One Package, One Set of Dependencies

In Cargo, **all targets in a package share the same dependency list**. If you include a dependency under `[dependencies]` in a crate's `Cargo.toml`, it applies to both the library and any binaries in that package [1] [2]. There is **no way to have completely separate `[dependencies]` for each binary target** within a single crate – the binaries inherently inherit the package's dependencies. (Dev-dependencies are only for tests, examples, etc., and do not apply to normal binaries.) This means that if you add a crate like `serde` for use in a binary, by default it would also be pulled in when your library is used by others [1].

To avoid forcing extra dependencies on library users, the typical approach is to **make those binary-only deps optional via Cargo features**. In other words, you declare `serde` (and similar heavy crates) as `optional = true` in `[dependencies]`, and gate their usage behind a feature flag in your code. This way, **library users don't pay the cost of unused binary dependencies** unless the feature is enabled [3] [4]. For example, in `Cargo.toml` you might have:

```
[dependencies]
serde = { version = "1", optional = true }
# ...other deps...

[features]
diagram = ["dep:serde"]  # feature enabling the optional serde dependency
```

However, making `serde` optional introduces a new challenge: your binary code under `src/bin/` (or `src/main.rs`) *requires* `serde` to compile. If the feature is off (as it will be by default), the binary's `use serde::...` imports will fail. The rest of this answer explores the idiomatic solutions to this problem.

Using Features to Gate Binary Dependencies

Rust's Cargo provides a mechanism to handle this via **feature flags and the `required-features` setting** on binary targets. The idea is to declare that your binary target **requires a certain feature** to be enabled, so that Cargo will only build the binary when that feature is active [5] [6].

In practice, you can introduce a Cargo feature (say `"cli"` or `"diagram"`) that enables all the binary-specific optional dependencies (like `serde`). Then, in the `Cargo.toml` `[[bin]]` entry for each binary, add a `required-features` field listing that feature. For example [7] [8]:

```
[features]
diagram = ["serde"]  # feature that pulls in serde (optional dep)

[[bin]]
name = "mytool"
required-features = ["diagram"]
```

With this setup, **if you try to build the binary without the `diagram` feature, Cargo will skip it** or emit an error saying the binary target requires an unavailable feature [5] [6]. This prevents the "unresolved import" errors – the binary simply won't compile unless you explicitly enable the feature. As of Rust 1.17, this is the intended way to mark that *"these optional deps are needed only for the binary"* [9] [8].

Important: Cargo will **not automatically enable** the feature for you; it just enforces that the feature must be on to build that target [10]. In practice, this means developers need to build/run the binary with `--features "diagram"` (or include that feature in their default set when working on the project) [11]. For example, running the tool would require:

```
cargo run --bin mytool --features diagram
```

If you forget the feature, Cargo will produce a clear error at compile time indicating the binary requires the feature [12]. This approach keeps the library side lightweight while allowing the binary to opt-in to heavier deps.

Drawbacks of the Single-Crate Feature Gating Approach

While using optional dependencies + required features is **canonical for single-crate packages**, it comes with some downsides in practice:

- **Manual feature toggling:** Developers (and CI scripts) must remember to enable the binary features when building or running those binaries. Forgetting to do so leads to build errors. This adds a bit of friction (e.g. always using `--features="diagram"` in commands) [11].

- `cargo install` **user experience:** If you publish this crate, a user doing `cargo install your_crate` will, by default, not get the binaries if they require a feature. Cargo will compile the library (since no features were enabled) and then **silently skip installing the binary**, only printing a warning [13] [14]. The warning looks like: *"none of the package's binaries are available for install using the selected features… bin requires the features:* `cli`*"*, which means the user needs to rerun the install with `--features="cli"` [14]. This is easy to miss and is considered poor UX – effectively, the binary doesn't install unless the user knows the magic incantation.

- **Enabling features by default – a tradeoff:** One might attempt to solve the above by marking the binary feature as part of the default features of the crate (so it's on by default, ensuring the binary is built and installed). However, that reintroduces the original problem: now **library users will unwittingly pull in those extra deps by default** [15] [16]. For example, if `lit-bit-core` made `diagram` a default feature (enabling `serde`), then anyone who does `cargo add lit-bit-core` will get `serde` and other optional crates unless they opt-out with `default-features = false` [15] [16]. This is considered undesirable because it violates the expectation of a "lean" core library.

- **Complexity in CI:** In a single-crate solution, your CI needs to test multiple configurations: the library in minimal form (feature off, possibly `no_std` on embedded target) and the binary with features on (on a host target). It's doable (e.g., running `cargo check --no-default-features --target <embedded>` and `cargo check --features diagram --bin mytool --target <host>`), but it requires careful setup. Tools like `cargo hack` or matrix builds are often used to ensure all feature combinations are tested. The process is manageable but not as straightforward as having separate crates.

Given these drawbacks, the Rust community often recommends a more robust solution: **split the package into multiple crates**.

Splitting into Library and Binary Crates (Workspace Approach)

The idiomatic long-term solution is to **use a Cargo workspace with two crates**: one for the core library and one for the binary (CLI/tool). This avoids all the feature flag gymnastics by giving each crate its own dependency list [17] [18]:

- **Core library crate (`lit-bit-core`):** Contains your reusable library code, minimal dependencies, and no required `serde` unless behind optional features. This crate can be `no_std` for embedded targets by default, with optional features (e.g. `"std"` or `"serde"`) to extend functionality on hosts. Library users would depend on this crate and only pull in what they need (e.g., they might never enable the `serde` feature).

- **Binary crate (e.g. `lit-bit-cli` or `lit-bit-tool`):** Contains the `main.rs` (and any binaries), depends on `lit-bit-core` (and *always* enables whatever features on `lit-bit-core` are needed for the binary's functionality). It also directly includes any binary-only dependencies like `serde`, `clap`, etc. in its own `[dependencies]`. This crate is a normal executable crate targeting the host (e.g., for CI tooling or diagram generation). It can use `std` freely since it's not for embedded. You can give this crate a name like `lit-bit-cli` but have it produce a binary named `lit-bit-core` (or `lit-bit`) if you want the installed tool name to match – Cargo allows specifying a different binary name in `Cargo.toml` [19].

How other projects do it: It's very common to see a `-core` and `-cli` (or `-tool`) split in Rust projects [20] [21]. For example, one might have `mycrate-core` and `mycrate-cli`, where `mycrate-cli` depends on `mycrate-core`. This way, library users add `mycrate-core = "X.Y"` to their `Cargo.toml` and **never even compile the CLI**; CLI users do `cargo install mycrate-cli`. This approach is explicitly recommended as the "clean" solution in many discussions [17] [22]. It aligns with the Rust package design philosophy: each crate has a clear responsibility and does not force unwanted dependencies on users who don't need them.

Maintaining a workspace: Using a workspace keeps the crates in one repository for easy coordination. You can have a top-level `Cargo.toml` listing the members (core and cli). The core crate remains lean, and the CLI crate can be as heavy as needed. Integration between the two is just via the normal dependency relationship (e.g., `lit-bit-cli` depends on `lit-bit-core` = { path = "../lit-bit-core", features = ["diagram"] } if it needs to enable that feature). This structure greatly simplifies conditional compilation: the core crate can use `#[cfg(feature = "serde")]` as before, but the binary crate will always enable it when needed, instead of relying on the end-user's feature selection.

Are There Other Workarounds or Cargo Features?

Binary-specific dependencies are a long-standing pain point, and there have been RFCs and discussions to improve it (e.g., an RFC to support per-target dependencies) [23] . As of 2025, there is *no stable Cargo feature* like `[target.'cfg(bin)'.dependencies]` to declare "bin-only" deps – target-specific dependency tables apply to OS/architecture cfgs, not to distinguishing binary vs library targets. All the clean solutions boil down to the two approaches above: **feature gating within one package, or splitting into multiple packages** [8] [17] .

Other less common approaches include: - **Using examples or dev-dependencies:** If the binaries are purely for internal testing or tooling (not meant to be installed by users), one trick is to put them under `examples/` or `tests/` so that they use `[dev-dependencies]` [24] [25] . This way, the library crate stays clean. You'd run them with `cargo run --example ...`. However, this is not suitable if you actually want to distribute those as real binaries (e.g., via `cargo install`) [26] . - **Symlink or subcrate hacks:** In some cases, people have a separate binary crate but keep its code in a subdirectory of the main repo (or even symlink `src/main.rs` into the library for convenience). While you *could* symlink files or use a workspace member that lives in a subfolder, it's usually simpler to just create two proper crates and share code by having the binary depend on the library. Symlinking source is considered a hack and can confuse tooling, so it's not an idiomatic solution.

- **Workspace-wide features:** If you have a workspace, you can define "virtual" features at the workspace level that simultaneously enable features in member crates (by naming them in each member's `[features]`). This can be used to make it convenient to turn on, say, `serde` support across several crates with one flag. In your case, though, since the binary crate will *always* use `serde`, you may not need a workspace feature – you'd just enable the library's feature in the binary's `Cargo.toml` dependency. Workspace features are more useful if you had multiple optional components spread across crates that you want to toggle together.

In summary, there isn't a magical hidden Cargo trick beyond **features or splitting crates**. The Rust RFCs acknowledge this is a gap, and the community consensus has gravitated to the two-package pattern as the "true path" in most cases [27] [28] .

Best Practices for CI in Dual-Target Workspaces

When supporting both embedded (no/std) and host (std) targets in a project, you should set up CI to **exercise all relevant feature combinations and targets**:

- **Test the core library in no_std mode on an embedded target:** For example, compile `lit-bit-core` for a Cortex-M or RISC-V target (using `cargo check/build --target thumbv7m-none-eabi --no-default-features`) to ensure it doesn't pull in `std` or optional deps by default. Also run its unit tests with no_std if possible (embedded testing can be tricky, but at least ensure it compiles for those targets).

- **Test the core library with features enabled on a host target:** e.g., `cargo test --features diagram` on `lit-bit-core` for a desktop target. This ensures that the optional `serde`-powered code actually works when enabled (and can catch any cfg errors).

- **Test the binary crate on the host:** Simply build and run tests for `lit-bit-cli` on a normal PC target (like x86_64). Since this crate always includes `serde` (no feature flags needed in this crate if it's separate), it will just compile with its full dependency set. This double-checks that the binary and library integrate correctly (the binary will pull in the core with the needed features).

- **Run linters/analysis on both**: Tools like `cargo geiger` (for detecting unsafe code) or others should be run on the library crate *without* features (to see the bare minimum unsafe usage) **and** with features (to catch any issues introduced by optional code), as well as on the binary crate. Similarly, if you use `cargo audit` or `cargo clippy`, run them for both crates. In a workspace, you can script this with Cargo's `-p` (package) flag or use a matrix in GitHub Actions to cover each package with appropriate feature flags.

- **Automate feature combinations**: Consider using `cargo hack` (a tool to test all feature combinations) or at least ensure that `--no-default-features` and `--all-features` builds are part of CI. This is important if your library has many feature flags (e.g., `"serde"`, `"std"`, etc.). This way, you won't accidentally break the no-feature (embedded) build when adding changes that use `std` or `serde`.

By structuring the project as a workspace, these CI tasks become clearer: you treat the core and CLI as separate units to test, each with their own set of features. This separation is **highly maintainable long-term** – it prevents accidental coupling of embedded and host concerns and makes it obvious which code is meant for which environment. As noted in community discussions, when binaries have different dependency requirements, **embracing a two-crate design is often the cleanest solution for clarity and maintainability** [29] [17].

In conclusion, the *canonical* Rust answer is: **use optional dependencies + features if you insist on one crate**, but be aware of the caveats; otherwise, **split into a small no_std library crate and a companion binary crate** for the best dev and user experience [27] [22]. This approach aligns with what many authoritative Rust projects do and will save you and your users from feature-coordination headaches in the long run.

Sources:

- Rust Forum – *"Dependencies from library and binary crate in a package"* (explains that package dependencies apply to both lib and bin, and suggests using an optional feature + `required-features` or splitting into separate crates) [1] [6].
- Stack Overflow – *"How can I specify binary-only dependencies?"* (KennyTM's answer outlining three methods: examples with dev-deps, optional deps with required-features, or separate packages) [8] [30].
- Axo.dev Blog – *"It's a library AND a binary"* by **Gankra** (deep dive on the pitfalls of single-crate library+binary design and why two packages is often the true solution) [14] [27].
- Rust Cargo Book – **Cargo Targets** (official docs on the `required-features` field for binaries) [5] and **Features** reference [31].
- Various Rust community discussions on Reddit and Users Forum (consensus that the "standard solution" is a core library crate plus a CLI crate with a `-cli` or `-tool` suffix) [22] [17].

---

[1] [3] [4] [6] [10] [12] [29] Dependencies from library and binary crate in a package - The Rust Programming Language Forum
https://users.rust-lang.org/t/dependencies-from-library-and-binary-crate-in-a-package/123206

[2] [5] Cargo Targets - The Cargo Book
https://doc.rust-lang.org/cargo/reference/cargo-targets.html

[7] [8] [9] [11] [24] [25] [30] rust - How can I specify binary-only dependencies? - Stack Overflow
https://stackoverflow.com/questions/35711044/how-can-i-specify-binary-only-dependencies

[13] [14] [15] [16] [27] [28] axo blog - It's a library AND a binary
https://blog.axo.dev/2024/03/its-a-lib-and-a-bin

[17] Publishing crate with lib and binary - The Rust Programming Language Forum
https://users.rust-lang.org/t/publishing-crate-with-lib-and-binary/124519

[18] [19] [20] [21] [22] [23] [26] Why doesn't rust have bin-dependencies? : r/rust
https://www.reddit.com/r/rust/comments/1fi32rm/why_doesnt_rust_have_bindependencies/

[31] Features - The Cargo Book
https://doc.rust-lang.org/cargo/reference/features.html