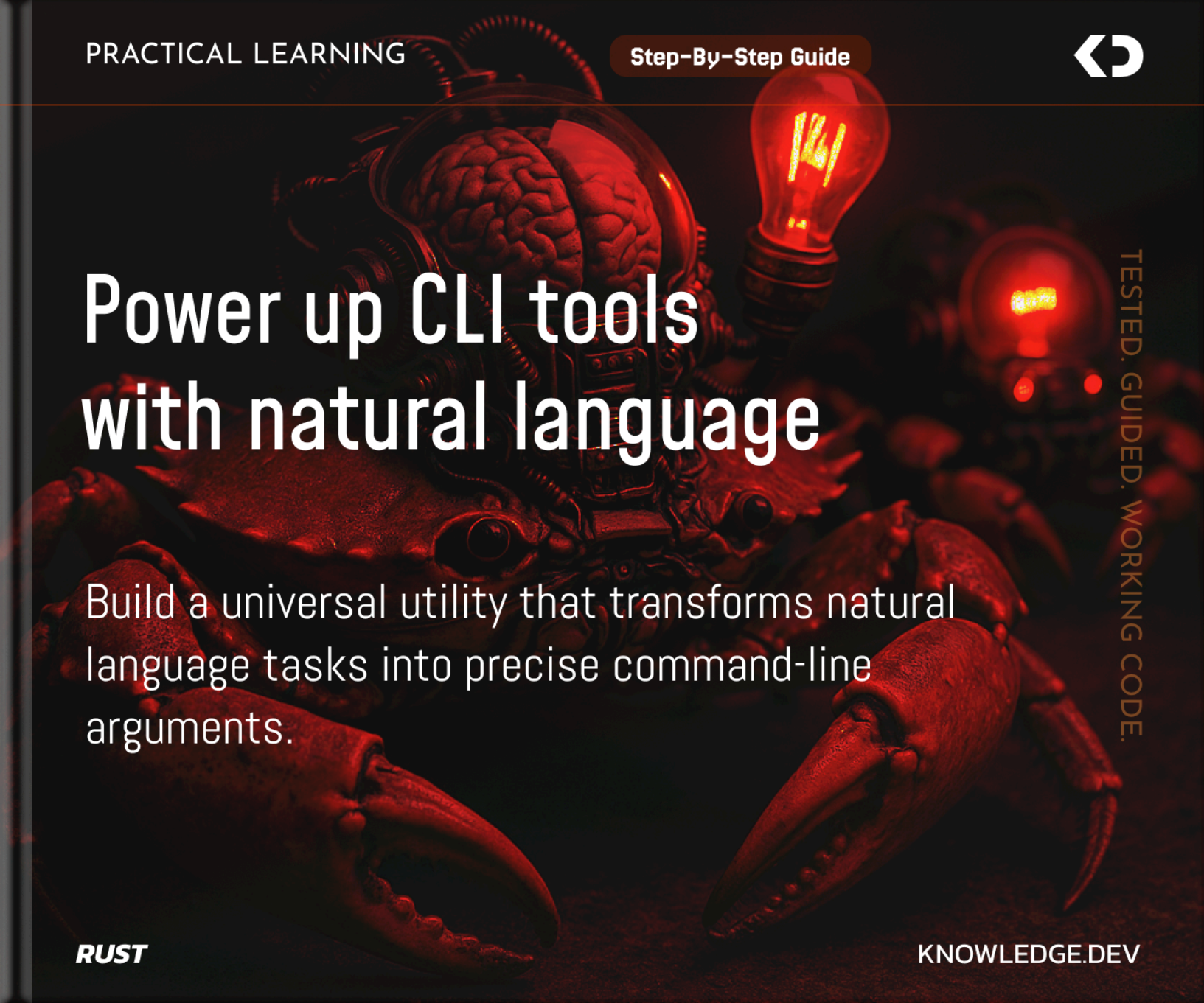
PRACTICAL LEARNING

Step-By-Step Guide



Power up CLI tools with natural language

Build a universal utility that transforms natural language tasks into precise command-line arguments.

TESTED. GUIDED. WORKING CODE.

RUST

KNOWLEDGE.DEV

Contents

Introduction

The tool we'll build is an interactive utility that turns natural-language instructions into command-line arguments for a specific command.

For example, to undo the last commit in a Git repository, we can run:

```sh
git reset --soft HEAD^
```

It would be handy to implement a command that accepts free-form text as its argument and derives the necessary options, for example:

```sh
git+ drop the last commit
```

Let's think about how we might implement this. In any case, this is a terminal program, so it won't need any graphical or programmatic interface.

We need to make the executable universal, so it can run as a replacement for any other command. We could achieve this by extracting the argument list. Since the first argument is the executable file name, we could easily understand what command name is being invoked.

The remaining arguments should simply be collected into a string and placed in a prompt that gets sent to the LLM. The prompt needs to be crafted so it outputs only the parameters for the command without extra noise, allowing us to use the generated output directly by substituting it into the original command and executing it.

We'll also ask for user confirmation — I wouldn't blindly execute what the model generates: actions could turn out to be destructive, especially if the user didn't formulate the request very precisely.

Binary Setup and Parsing

Initialize CLI App

Create an empty crate with an application (executable file).

You can do this by running the `cargo new` command with the `--bin` flag, specifying the name of the application (in our case, it will be `add-plus`).

.

```sh
$ cargo new --bin add-plus
```

Cargo.toml

```toml
[package]
name = "add-plus"
version = "0.1.0"
edition = "2024"

```

src/main.rs

```rust
fn main() {
}
```

Rust does not have a very convenient error type. Therefore, I recommend adding the `anyhow` crate, which provides a universal error type.

It also offers a type alias for the `Result` type, with the error type from this crate pre-configured.

Cargo.toml TOML

```toml
[dependencies]
anyhow = "1.0.99"
```

src/main.rs RUST

```rust
use anyhow::Result;
```

src/main.rs RUST

```rust
fn main() -> Result<()> {
    Ok(())
}
```

Extract Command Name

Our utility will take and process parameters passed through the command line, so we need to access these parameters, and we can do this using the `env` module from the standard library.

This module contains the `args` function, which returns an iterator over the command line arguments.

Obtain this iterator and store it in a variable called `args`.

src/main.rs RUST

```rust
use std::env;
```

src/main.rs RUST

```rust
fn main() -> Result<()> {
    let mut args = env::args();
    Ok(())
}
```

Since the `args` structure is an iterator, we use the `next` method once to get the first argument, which is the name of the executable file in terms of command-line parameters.

src/main.rs RUST

```rust
fn main() -> Result<()> {
    let mut args = env::args();
    let alias_name = args.next();
    Ok(())
}
```

The return value is an `Option`, which you can print using the standard `unwrap()` method. However, it's more elegant to use the `context()` method provided by the `Context` trait from the `anyhow` crate. In this case, the `Option` is converted into a `Result` that contains an error if the value is not set.

Can there be a situation where the executable file name is not in the arguments? The operating system does not enforce what must be included in the parameters when running a command. However, by convention, at least one argument should always be present. Still, this does not guarantee that the argument is the name of the binary file. Also this argument can be an empty string. If you plan to reuse this code in the future or add custom argument processing, it's better to make the algorithm more robust, so it handles the absence of arguments correctly.

src/main.rs RUST

```rust
use anyhow::{Context as _, Result};
```

src/main.rs RUST

```rust
fn main() -> Result<()> {
    let mut args = env::args();
    let alias_name = args.next()
        .context("No name of the app provided")?;
    Ok(())
}
```

The executable file name is not just the name of the file. Technically, it can be the path to the file along with its name.

To correctly handle file paths, convert `alias_name` to the `Path` type using the `new()` method. Then, extract the file name from the parsed path by calling the `file_name()` method, and save the result in the variable `binary_name`.

src/main.rs RUST

```rust
use std::{
    env,
    path::Path,
};
```

src/main.rs RUST

```rust
fn main() -> Result<()> {
    let mut args = env::args();
    let alias_name = args.next()
        .context("No name of the app provided")?;
    let binary_name = Path::new(&alias_name)
        .file_name();
    Ok(())
}
```

The `file_name()` method returns an `Option` because the path might be empty, a root path, or end with a separator.

While the provided path to the executable file should always contain a file name if executed correctly, we cannot fully ensure this. Our utility might be run programmatically, where command line parameters can be arbitrary. Therefore, we will use the `context()` method to extract the value or return an error if the file name is absent.

src/main.rs RUST

```rust
fn main() -> Result<()> {
    let mut args = env::args();
    let alias_name = args.next()
        .context("No name of the app provided")?;
    let binary_name = Path::new(&alias_name)
        .file_name()
        .context("Invalid path")?;
    Ok(())
}
```

The returned file name is of type `OsStr`, as the operating system may use different file systems and different encodings for file names. This means the returned string is not guaranteed to be a Unicode string.

However, we can attempt to convert it using the `to_str()` method, which also returns an `Option` if the file name cannot be converted to a Unicode string. If it cannot, we will use the `context()` method to return an error.

src/main.rs RUST

```rust
fn main() -> Result<()> {
    let mut args = env::args();
    let alias_name = args.next()
        .context("No name of the app provided")?;
    let binary_name = Path::new(&alias_name)
        .file_name()
        .context("Invalid path")?
        .to_str()
        .context("Invalid UTF-8 in filename")?;
    Ok(())
}
```

By convention, the name of the binary file should match the name of the utility for which we are setting parameters, with the only difference being that a `+` sign is added to the name. This indicates that it is an extension or addition to the original command, using AI to generate parameters.

To remove this plus sign and obtain the name of the utility for which we are setting parameters, use the `trim_end_matches()` method. Pass the character that needs to be removed from the end of the string. Do this and save the result in the variable `real_name`.

src/main.rs RUST

```rust
fn main() -> Result<()> {
    let mut args = env::args();
    let alias_name = args.next()
        .context("No name of the app provided")?;
    let binary_name = Path::new(&alias_name)
        .file_name()
        .context("Invalid path")?
        .to_str()
        .context("Invalid UTF-8 in filename")?;
    let real_name = binary_name.trim_end_matches('+');
    Ok(())
}
```

Collect all remaining arguments into a vector using the `collect()` method. Then, combine them into a single string using the `join()` method, with a space (`" "`) as the separator.

Store the result in a variable called `natural_args`. This variable will be a `String` containing a natural language query, which we will pass to the model in a prompt.

src/main.rs RUST

```rust
fn main() -> Result<()> {
    let mut args = env::args();
    let alias_name = args.next()
        .context("No name of the app provided")?;
    let binary_name = Path::new(&alias_name)
        .file_name()
        .context("Invalid path")?
        .to_str()
        .context("Invalid UTF-8 in filename")?;
    let real_name = binary_name.trim_end_matches('+');
    let natural_args = args.collect::<Vec<_>>()
        .join(" ");
    Ok(())
}
```

Prompt to Parameters

Add Prompt Template

We have the required input data: the name of the command, and a query in natural language. Now we need to create a prompt for the LLM to transform the natural language query into formal parameters for this command.

Because a prompt is typically substantial text, keep it in a separate file. Conveniently, Rust lets us embed strings at compile time via the `include_str!` macro. Create an empty `prompt.md` under `src` (if you end up with many prompts, consider a top-level folder).

Add the file to the program by assigning it to a static variable `PROMPT`.

src/main.rs RUST

```rust
6 static PROMPT: &str = include_str!("prompt.md");
```

You've probably heard that LLMs benefit from setup — a prompt that assigns a role. That framing determines how the model will behave. I wouldn't say formality matters; on the contrary, the more naturally you phrase the role, the more useful and creative the responses tend to be.

We should also state our agent's role in the prompt. As you know, we want to convert natural language into formal command-line parameters for a specific command.

Note that we are using a `{COMMAND}` placeholder for the command. We'll substitute it later with the command name you extracted from the first command-line argument (the invoked executable).

src/prompt.md MARKDOWN

```
1 You are given a natural language instruction from a user that
  describes how they want to use the command `{COMMAND}`.
```

With the role defined, we should clearly state the task — what we expect the model to do. Our agent's job is to convert the instruction into a list of valid parameters.

This overlaps with the role, but there's a difference: the role tells the agent who it is, while the task tells it what to do.

src/prompt.md MARKDOWN

```
2 Your task is to convert this natural language instruction into a
  valid set of parameters for the command.
```

We expect the result in a specific format. It's important to be explicit here: ideally, it's just **a single line** of parameters that we can pass to the real command.

Also the model will behave more reliably if we show a concrete example of **the desired output**.

From the example it's clear that parameters (named and positional), flags, and even subcommands may appear. It also makes explicit that the command itself is omitted — we only want the parameters.

src/prompt.md MARKDOWN

```
Only output the parameters in a single line for the command in the terminal.

Output example:
"subcommand -f --flag --parameter=value arg1 arg2"
```

There's a nuance: we've defined what the model should do, but not what it must avoid. A chatty model might explain which parameters it selected — still a single line, but with extra text.

To keep the output clean, add a constraint and tell the model to output only the parameters and nothing else.

The example already nudges the model toward returning only parameters; the constraint ensures it won't improvise — for instance, by adding shell comments.

src/prompt.md MARKDOWN

```
7 Do not include explanations, extra text, or the command name
  itself.
```

Sometimes the instruction won't be specific enough. For example, a command may require mandatory parameters the user didn't include. In that case, prefer sensible defaults: omit those parameters rather than inventing values — i.e., avoid adding anything extra.

That effectively forbids fabricating data. Requests can also contain contradictions; in those cases, ask the model to pick a single, reasonable interpretation.

These hints make the agent more proactive: its goal is to help, and with them the model will choose a minimal, workable solution.

src/prompt.md MARKDOWN

```
8  If some parameters are missing, omit them rather than inventing
   values.
9
10 If the user input is ambiguous, choose the most reasonable
   interpretation.
```

We've set the model's role, shown an example, and added constraints. Now we can specify the input — the data the model receives.

Because the user provides the request at runtime, we don't know it in advance. We'll use the `{INPUT}` placeholder and later substitute the natural-language description we constructed by joining the remaining arguments into a sentence.

src/prompt.md MARKDOWN

```
11 Input:
12 "{INPUT}"
```

Fill Prompt Template

Now we need to fill in the template for the prompt, which we have included in the static variable `PROMPT`.

Our template has 2 placeholders that we need to replace. The first one is `{COMMAND}`, where we need to insert the name of the command for which we are selecting parameters. To do this, we can use the `replace()` method, providing the command placeholder and the utility name that was previously stored in the variable `real_name` as parameters.

src/main.rs RUST

```
7  fn main() -> Result<()> {
19     let prompt = PROMPT
20         .replace("{COMMAND}", real_name);
21     Ok(())
22 }
```

The second placeholder we need to replace is `{INPUT}`. We have already constructed a natural language query from the provided arguments and stored it in the variable `natural_args`. Use it in the `replace()` method to substitute the placeholder with the input parameters.

src/main.rs RUST

```
 7 fn main() -> Result<()> {
19     let prompt = PROMPT
20         .replace("{COMMAND}", real_name)
21         .replace("{INPUT}", &natural_args);
22     Ok(())
23 }
```

Send Prompt to Model

To interact with models, we need a crate to avoid implementing the API of an LLM directly. The Rig project can assist with this, especially through their `rig-core` crate. It provides a set of functions necessary for interacting with models. Add it to your `Cargo.toml` file.

Cargo.toml TOML

```toml
[dependencies]
anyhow = "1.0.99"
rig-core = "0.18.2"
```

The `rig-core` crate implements different APIs, and in our case, we will use OpenAI. Import this module into your project from the `providers` submodule.

In this module you can find a structure called `Client`. We can create an instance of this structure using the `from_env()` method, which is part of the `ProviderClient` trait. This method allows us to create a client instance based on environment variables.

src/main.rs RUST

```rust
use rig::{
    client::ProviderClient,
    providers::openai,
};
```

src/main.rs RUST

```rust
fn main() -> Result<()> {
    let client = openai::Client::from_env();
    Ok(())
}
```

The client is set up using only the API access token. To use a specific model, utilize the agent builder by calling the `agent()` method from the `CompletionClient` trait.

Provide the model's name or alias as a parameter. If no extra configuration is needed, simply call the `build()` method.

src/main.rs RUST

```rust
use rig::{
    client::{CompletionClient, ProviderClient},
    providers::openai,
};
```

src/main.rs RUST

```rust
fn main() -> Result<()> {
    let client = openai::Client::from_env();
    let agent = client.agent("gpt-5").build();
    Ok(())
}
```

In `rig` terminology, a model you can interact with is called an `Agent`. This type has a method `prompt()`, which allows you to send a chat completion request to the model. We already have a prepared request stored in the variable `prompt`, so simply pass this variable as a parameter to the method.

However, the `prompt()` method is asynchronous, so we need an asynchronous runtime. In other words, we need to use the `await` operator to execute it and get the result.

src/main.rs RUST

```rust
use rig::{
    client::{CompletionClient, ProviderClient},
    completion::Prompt,
    providers::openai,
};
```

src/main.rs RUST

```rust
fn main() -> Result<()> {
    let client = openai::Client::from_env();
    let agent = client.agent("gpt-5").build();
    let real_args = agent.prompt(prompt);
    Ok(())
}
```

The most popular and convenient asynchronous runtime in Rust is Tokio. Add the `tokio` crate to the dependencies list in your `Cargo.toml` file and enable the `full` feature. This will give you access to the multi-threaded runtime and a special macro for making the `main()` function asynchronous.

In your `main.rs` file, modify the `main()` function to make it asynchronous. Use the `#[tokio::main]` attribute from the `tokio` crate directly on your `main` function.

Cargo.toml TOML

```toml
[dependencies]
anyhow = "1.0.99"
rig-core = "0.18.2"
tokio = { version = "1.47.1", features = ["full"] }
```

src/main.rs RUST

```rust
#[tokio::main]
async fn main() -> Result<()> {
    Ok(())
}
```

Now we can use the `await` operator to get the result of the request. It returns a result, so we use the `?` operator to handle any errors if something goes wrong when calling the method, and halt program execution.

src/main.rs RUST

```rust
#[tokio::main]
async fn main() -> Result<()> {
    let client = openai::Client::from_env();
    let agent = client.agent("gpt-5").build();
    let real_args = agent.prompt(prompt).await?;
    Ok(())
}
```

Confirm Command Execution

Confirm Command

Now we have the utility name and its actual arguments stored in the `real_name` and `real_args` variables. Let's combine them into a single string using the `format!` macro to ask the user for confirmation if they agree to execute the resulting command and parameters.

src/main.rs RUST

```rust
#[tokio::main]
async fn main() -> Result<()> {
    let prompt = format!("{real_name} {real_args}");
    Ok(())
}
```

The most elegant way to request user confirmation in the terminal is by using the `dialoguer` crate. It provides a `Confirm` structure, which you can create by calling the `new` method.

To interact with the user, use the `interact()` method. This method is called without any parameters and returns a `Result`. Make sure to handle the result with the `?` operator. Note that this method is synchronous because terminal interactions cannot be done asynchronously.

The method returns a `bool` value. If the user confirms the command, it returns `true`, and if they cancel, it returns `false`. Simply store this boolean value in a variable named `run`.

Cargo.toml TOML

```toml
[dependencies]
anyhow = "1.0.99"
dialoguer = "0.12.0"
rig-core = "0.18.2"
tokio = { version = "1.47.1", features = ["full"] }
```

src/main.rs RUST

```rust
use dialoguer::Confirm;
```

src/main.rs RUST

```rust
#[tokio::main]
async fn main() -> Result<()> {
    let prompt = format!("{real_name} {real_args}");
    let run = Confirm::new()
        .interact()?;
    Ok(())
}
```

You can specify a message to display when asking the user for confirmation using the `with_prompt()` method, which takes the message as a parameter. You can use a variable named `prompt` for this purpose.

The `report()` method allows you to turn off the reporting of the chosen option. We will do this because we display the prompt as a command with arguments, and extra output would make it cluttered.

src/main.rs RUST

```
13 #[tokio::main]
14 async fn main() -> Result<()> {
32     let prompt = format!("{real_name} {real_args}");
33     let run = Confirm::new()
34         .with_prompt(prompt)
35         .report(false)
36         .interact()?;
37     Ok(())
38 }
```

Execute Confirmed Command

We now have a variable `run` that indicates we need to execute the received command.

Add an `if` block with this variable that will execute only if the user has confirmed the execution.

src/main.rs RUST

```
13 #[tokio::main]
14 async fn main() -> Result<()> {
37     if run {
38     }
39     Ok(())
40 }
```

The `tokio` crate provides the `process` module, which allows you to run various commands. It contains the `Command` struct, responsible for building a command to execute and interacting with it afterward.

Import this type and use the `new()` method to create the command you want to run. Pass the variable `real_name` as its name and store the result in a mutable variable `cmd`, since we will still configure arguments and input-output for it.

src/main.rs RUST

```rust
use tokio::process::Command;
```

src/main.rs RUST

```rust
#[tokio::main]
async fn main() -> Result<()> {
    if run {
        let mut cmd = Command::new(real_name);
    }
    Ok(())
}
```

In the past, we combined arguments into a single sentence to create a natural language query. Now, we need to do the opposite with the `real_args` variable returned by the model. This `String` contains a list of arguments that we need to pass to the command.

To split the string into arguments, we can use the string method `split_whitespace()`, and then collect everything into a `Vec` of string references, where each element represents one argument.

However, this method of splitting the string has a small drawback. If quotes are used and the model provides complex parameters with multiple words, this case will not be handled correctly. As an additional exercise, you can think about and try to improve this later.

src/main.rs RUST

```rust
#[tokio::main]
async fn main() -> Result<()> {
    if run {
        let mut cmd = Command::new(real_name);
        let args_list: Vec<&str> = real_args
            .split_whitespace()
            .collect();
    }
    Ok(())
}
```

Now we can add these arguments to the command using the `args()` method. You need to provide a reference to the list of arguments as a parameter. This list must be iterable, and the method will automatically go through all the arguments and add them to the command.

src/main.rs RUST

```rust
#[tokio::main]
async fn main() -> Result<()> {
    if run {
        let mut cmd = Command::new(real_name);
        let args_list: Vec<&str> = real_args
            .split_whitespace()
            .collect();
        cmd.args(&args_list);
    }
    Ok(())
}
```

To redirect the input and output of the executed command to the I/O used by our executable file at runtime, we use the methods `stdin()`, `stdout()`, and `stderr()` of the `Command` struct.

You need to pass a handle of the stream that will be used for reading or writing data. In our case, we will simply inherit our standard input and output. To do this, import the `Stdio` type from the `process` module and call the `inherit()` method, which will return a handle to the current command in the current execution context.

src/main.rs RUST

```rust
use std::{
    env,
    path::Path,
    process::Stdio,
};
```

src/main.rs RUST

```rust
#[tokio::main]
async fn main() -> Result<()> {
    if run {
        let mut cmd = Command::new(real_name);
        let args_list: Vec<&str> = real_args
            .split_whitespace()
            .collect();
        cmd.args(&args_list);
        cmd.stdin(Stdio::inherit());
        cmd.stdout(Stdio::inherit());
        cmd.stderr(Stdio::inherit());
    }
    Ok(())
}
```

Everything is now ready to run the command. Since we don't need to capture its output and only want to show it to the user, we can simply execute the command and wait for it to finish. We can use the `status()` method for this. It requires no parameters and returns a `Future`. We use the `await` operator to get the result.

The result we receive is the command's execution result. It includes the exit code if the command succeeds, or an error if the command fails. Note that this error is not about the command's execution itself. The exit code already indicates if the command was successful or not. The result from the `status()` method tells us whether the command could not be started or if the status code could not be retrieved. This is separate from the command's execution result and is related to the attempt to run it.

src/main.rs RUST

```rust
#[tokio::main]
async fn main() -> Result<()> {
    if run {
        let mut cmd = Command::new(real_name);
        let args_list: Vec<&str> = real_args
            .split_whitespace()
            .collect();
        cmd.args(&args_list);
        cmd.stdin(Stdio::inherit());
        cmd.stdout(Stdio::inherit());
        cmd.stderr(Stdio::inherit());
        let status = cmd.status().await?;
    }
    Ok(())
}
```

The `status()` method returns an instance of the `ExitStatus` type. This instance may not always contain an exit code, but we can obtain it using the `code()` method, which returns an `Option`.

We expect to always get an exit code. Therefore, we will use the `context()` method to turn an empty `Option` into an error, indicating that the exit code was not obtained.

src/main.rs RUST

```rust
#[tokio::main]
async fn main() -> Result<()> {
    if run {
        let mut cmd = Command::new(real_name);
        let args_list: Vec<&str> = real_args
            .split_whitespace()
            .collect();
        cmd.args(&args_list);
        cmd.stdin(Stdio::inherit());
        cmd.stdout(Stdio::inherit());
        cmd.stderr(Stdio::inherit());
        let status = cmd.status().await?;
        let code = status
            .code()
            .context("Cannot get the exit code")?;
    }
    Ok(())
}
```

Let's improve our command so that it also returns an exit code. To do this, we need to import the `ExitCode` type from the standard `process` module and return it as the result from the `main()` function.

We should also return a success code at the end of the command. There is an associated constant `SUCCESS` with the `ExitCode` type for this purpose.

src/main.rs RUST

```
3  use std::{
4      env,
5      path::Path,
6      process::{ExitCode, Stdio},
7  };
```

src/main.rs RUST

```
15 #[tokio::main]
16 async fn main() -> Result<ExitCode> {
53     Ok(ExitCode::SUCCESS)
54 }
```

Now we need to consider how to obtain the result code from command execution. In other words, how do we convert the `ExitStatus` into an `ExitCode`?

We have already converted the `ExitStatus` to a code of type `i32`. However, an exit code can only be created from a `u8` type. Therefore, use the `try_into` method to try converting the returned code into the desired type.

src/main.rs RUST

```rust
#[tokio::main]
async fn main() -> Result<ExitCode> {
    if run {
        let mut cmd = Command::new(real_name);
        let args_list: Vec<&str> = real_args
            .split_whitespace()
            .collect();
        cmd.args(&args_list);
        cmd.stdin(Stdio::inherit());
        cmd.stdout(Stdio::inherit());
        cmd.stderr(Stdio::inherit());
        let status = cmd.status().await?;
        let code: u8 = status
            .code()
            .context("Cannot get the exit code")?
            .try_into()?;
    }
    Ok(ExitCode::SUCCESS)
}
```

Now we can construct the exit code and return it as the result of the entire program using `return`.

src/main.rs RUST

```rust
#[tokio::main]
async fn main() -> Result<ExitCode> {
    if run {
        let mut cmd = Command::new(real_name);
        let args_list: Vec<&str> = real_args
            .split_whitespace()
            .collect();
        cmd.args(&args_list);
        cmd.stdin(Stdio::inherit());
        cmd.stdout(Stdio::inherit());
        cmd.stderr(Stdio::inherit());
        let status = cmd.status().await?;
        let code: u8 = status
            .code()
            .context("Cannot get the exit code")?
            .try_into()?;
        return Ok(ExitCode::from(code));
    }
    Ok(ExitCode::SUCCESS)
}
```

A more elegant solution would be to return the result of the command execution if it runs. Otherwise, return a success exit code.

To do this, add an `else` branch to our `if` statement, and in this alternative branch, simply return a success result. Remove the previous result that was returned at the end of the `main()` function.

src/main.rs RUST

```
15 #[tokio::main]
16 async fn main() -> Result<ExitCode> {
57
58 }
```

src/main.rs RUST

```
15 #[tokio::main]
16 async fn main() -> Result<ExitCode> {
39     if run {
54     } else {
56     }
57
58 }
```

src/main.rs RUST

```
15 #[tokio::main]
16 async fn main() -> Result<ExitCode> {
39     if run {
54     } else {
55         Ok(ExitCode::SUCCESS)
56     }
57
58 }
```

Since the main function now ends with an if block that has two branches, the primary and the alternative, the return statement is no longer needed. You can remove it.

src/main.rs RUST

```
15 #[tokio::main]
16 async fn main() -> Result<ExitCode> {
39     if run {
40         let mut cmd = Command::new(real_name);
41         let args_list: Vec<&str> = real_args
42             .split_whitespace()
43             .collect();
44         cmd.args(&args_list);
45         cmd.stdin(Stdio::inherit());
46         cmd.stdout(Stdio::inherit());
47         cmd.stderr(Stdio::inherit());
48         let status = cmd.status().await?;
49         let code: u8 = status
50             .code()
51             .context("Cannot get the exit code")?
52             .try_into()?;
53         Ok(ExitCode::from(code))
54     } else {
56     }
57 }
```

Release Build and Alias

Install Release Binary

Our team is fully prepared for the challenge. Build the release version of the application using the `build` command with the `--release` flag.

Then, copy the compiled binary to a directory such as `bin`, which you can access from your terminal. For example, I use a directory named `~/.local/bin`.

Also, create an alias in this folder, and name it after the command you want to run. This alias will help convert natural language arguments into formal ones. In my case, the command is `git`, so I name the alias `git+`.

```
.                                                                    SH
$ cargo build release
```

```
.                                                                    SH
$ cp target/release/add-plus ~/.local/bin/add-plus
```

```
.                                                                    SH
$ ln -s ~/.local/bin/add-plus ~/.local/bin/git+
```

Run the Tool

To create the OpenAI client, we used environment variables. Before running the command, you need to set the environment variable `OPENAI_API_KEY` with your OpenAI API key.

Once this is done, you can execute the command. In my case, I run it using the alias `git+`, specifying that it should **show a compact list of commits**.

Remember to run this command from a folder with a real git repository. As a result, you'll get a prompt to try running the `git` command with the necessary parameters to see this list, like:

```sh
git log --oneline
```

It's convenient and effective.

```sh
$ export OPENAI_API_KEY="<your key>"
```

```sh
$ git+ show a compact list of commits
```