# CLI Task Manager — Rust with JS/Python Parallels

## 1. Project Overview

- Simple CLI todo app: add, list, mark done, remove tasks stored as JSON.
- Built in Rust to experience memory safety/borrowing vs. the garbage-collected world of Node/Django.
- Stack: `clap` for CLI parsing (like Express route matching), `serde` for JSON (like `json.dumps`/`JSON.stringify`), `anyhow` for ergonomic errors (like a typed `try/except`).

## 2. Prerequisites

- Rust toolchain: `curl https://sh.rustup.rs -sSf | sh`
- Comfortable with a shell, JSON, and basic data structures.

- Optional: VS Code + `rust-analyzer` for inline borrow checker feedback.

## 3. Quick Start

```
cargo run -- add "Buy milk"
cargo run -- list
cargo run -- done 1
```

## 4. Architecture Walkthrough

### `src/main.rs` — Entry Point & Command Dispatch

**What it does**: Parses CLI args, resolves the data file path, loads tasks, routes subcommands, saves state.

**JS equivalent**:

```
const express = require('express');
app.post('/tasks', addTask);
app.get('/tasks', listTasks);
app.post('/tasks/:id/done', markDone);
```

**Key Rust concepts**:

- `match cli.command`: Like `switch`, but compiler forces you to handle every variant.
- `task::load_tasks(&data_path)?`: `?` = propagate errors upward (no silent failures).
- `&mut tasks` vs `&tasks`: Mutable borrow for writes, shared borrow for reads—compiler enforces who can mutate.

**Annotated code**:

```
fn main() -> anyhow::Result<()> {
    let cli = Cli::parse();                    // clap derives parser from the
struct
    let data_path = get_data_path()?;          // resolve platform-specific
data dir
```

```
    let mut tasks = task::load_tasks(&data_path)?; // ? = early return on
Err

    match cli.command {                        // pattern-match every CLI
variant
        Commands::Add { description } => {
            task::add_task(&mut tasks, description)?; // &mut = exclusive
write
            task::save_tasks(&data_path, &tasks)?;    // & = shared read
        }
        Commands::List { all } => task::list_tasks(&tasks, all),
        Commands::Done { id } => {
            task::mark_done(&mut tasks, id)?; // mutably update one task
            task::save_tasks(&data_path, &tasks)?;
        }
        Commands::Remove { id } => {
            task::remove_task(&mut tasks, id)?;
            task::save_tasks(&data_path, &tasks)?;
        }
    }
    Ok(()) // success exit code
}
```

**AI prompts that helped**:

- "Explain `&mut` vs `&` with a Django ORM analogy"
- "Why does Rust require `mod task;` in main.rs but Python doesn't?"
- "Show how `?` replaces nested try/catch in Node"

## `src/task.rs` — Data Model & Business Logic

**What it does**: Defines `Task`, loads/saves JSON, adds/lists/mutates tasks with validation.

**Python equivalent**:

```
import json
def load_tasks(path):
    if not os.path.exists(path): return []
    return json.load(open(path))
```

**Key Rust concepts**:

- `Result<T, E>`: Like `try/except`, but the compiler makes you handle both `Ok` and `Err`.

- `Option`-like patterns via `iter_mut().find(...)`: Returns `Some`/`None` instead of throwing.
- Atomic file write: write to `tasks.json.tmp`, flush, then `rename` (replaces the file).

**Annotated code**:

```rust
#[derive(Serialize, Deserialize)] // serde derives JSON (like pydantic/dataclasses)
pub struct Task {
    pub id: u32,
    pub description: String,
    pub completed: bool,
}

pub fn load_tasks(path: &Path) -> anyhow::Result<Vec<Task>> {
    if !path.exists() { return Ok(vec![]); }                      // missing file = empty list
    let data = fs::read_to_string(path)?;                         // IO Result<T, E>
    if data.trim().is_empty() { return Ok(vec![]); }              // tolerate empty file
    let tasks = serde_json::from_str(&data)?;                     // JSON -> Vec<Task>
    Ok(tasks)
}

pub fn save_tasks(path: &Path, tasks: &[Task]) -> anyhow::Result<()> {
    if let Some(parent) = path.parent() { fs::create_dir_all(parent)?; } // mkdir -p
    let data = serde_json::to_string_pretty(tasks)?;                    // Vec<Task> -> JSON
    let tmp = path.with_extension("tmp");
    let mut file = fs::File::create(&tmp)?;                             // write to temp first
    file.write_all(data.as_bytes())?; file.sync_all()?;                 // flush to disk
    fs::rename(&tmp, path)?;                                            // atomic replace
    Ok(())
}

pub fn add_task(tasks: &mut Vec<Task>, description: String) -> anyhow::Result<()> {
    let desc = description.trim();
    if desc.is_empty() { anyhow::bail!("Task description cannot be empty"); }
    let next_id = tasks.iter().map(|t| t.id).max().unwrap_or(0) + 1; // compute next id
    tasks.push(Task { id: next_id, description: desc.to_owned(), completed: false });
    Ok(())
}
```

**AI prompts that helped**:

- "Explain `.iter_mut().find()` as the Rust equivalent of Python's `next((t for t in tasks if ...), None)`"
- "How do I write files atomically in Rust like Python's `tempfile` + `os.replace`?"
- "Why does `&[Task]` satisfy a `&Vec<Task>` parameter (slices vs vectors)?"

**`Cargo.toml` — Dependencies & Metadata**

**JS/Python equivalent**: `package.json` / `requirements.txt`.

```
[dependencies]
clap = { version = "4", features = ["derive"] }   # like yargs/argparse
serde = { version = "1", features = ["derive"] }  # like
pydantic/JSON.stringify
serde_json = "1"                                  # JSON parser/printer
anyhow = "1"                                      # ergonomic error type
directories = "6"                                 # platform data dirs
(~/.local/share/...)
```

**AI prompt that helped**:

- "Map these Rust crates to Node/Python libraries I already know"

## 5. Concepts Explained Through Code

### 5.1 Ownership & Borrowing

**The problem JS/Python hide**:

```
let tasks = [{ id: 1, desc: "Task" }];
function clearTasks(list) { list.length = 0; } // silently mutates caller
clearTasks(tasks);
```

**Rust's solution**:

```
fn clear_tasks(tasks: &mut Vec<Task>) { tasks.clear(); } // explicit mutable
borrow
```

```
fn read_tasks(tasks: &Vec<Task>) { /* cannot modify tasks here */ } //
immutable borrow
```

**Rule**: One `&mut` **or** many `&` at a time. The compiler prevents data races and spooky action-at-a-distance.

**AI prompt**: "Show me a real bug in Python that Rust's borrow checker prevents"

## 5.2 Error Handling

**Python approach**:

```
try:
    data = json.load(file)
except json.JSONDecodeError:
    print("Invalid JSON")
```

**Rust approach**:

```
let data: Result<Vec<Task>, _> = serde_json::from_str(&contents);
match data {
    Ok(tasks) => println!("Loaded"),
    Err(e) => eprintln!("Error: {e}"),
}
```

**Why better**: `Result` forces you to handle `Err` at compile time; no swallowed exceptions.

**AI prompt**: "Compare Rust's Result to Python's try/except—when is each better?"

## 5.3 Enums (Discriminated Unions)

**TypeScript equivalent**:

```
type Command =
  | { type: 'add', description: string }
  | { type: 'done', id: number };
```

**Rust version**:

```
enum Commands {
    Add { description: String },
    List { all: bool },
    Done { id: u32 },
    Remove { id: u32 },
}
```

**Power**: Each variant carries different data; `match` is exhaustive so you don't forget a branch.

**AI prompt**: "Why use enums instead of strings for CLI commands?"

## 6. Common Gotchas (From My Learning)

| Error | Cause | Fix |
|---|---|---|
| `use of moved value` | Passed ownership, tried to reuse | Borrow with `&` or clone intentionally |
| `cannot borrow as mutable` | Immutable borrow still in scope | Limit the immutable borrow or restructure code |
| `unresolved module 'task'` | Missing `mod task;` | Add `mod task;` at the top of `main.rs` |

**AI prompt that saved me**: "Explain 'cannot borrow as mutable more than once' with a file I/O example"

## 7. Testing

```
cargo test
cargo run -- add "Test task"
cargo run -- list
```

## 8. Next Steps for Learning

- Add persistent IDs that never reuse after deletion.
- Implement undo (keep an action log).
- Expose `--data-path` flag to point at alternate JSON files (handy for tests).
- Use `clap`'s `--help` on subcommands to document usage (`cargo run -- --help`).
- Read The Rust Book chapters 4 (Ownership) and 9 (Error Handling).

## 9. AI-Assisted Learning Log

**Questions I asked AI**:

1. "Explain Rust modules vs Python imports"
2. "Why does `&[Task]` accept `&Vec<Task>`? (slices vs vectors)"
3. "Show me how `?` replaces nested try/catch"
4. "Refactor this Python task manager to Rust—explain each change"

**Most useful prompt pattern**: "Compare [Rust concept] to [Python/JS equivalent] with a real code example"

## 10. Project Files Reference

- `Cargo.toml`: Dependencies (like `package.json` / `requirements.txt`)
- `src/main.rs`: Entry point + CLI routing
- `src/task.rs`: Data model + business logic
- `~/.local/share/ian/mwirigi/cli_task_manager/tasks.json`: Persistent storage (created on first run; adjust via `XDG_DATA_HOME` if you want it elsewhere)