[Pre-Development Specification for Minimalist Rust Utilities](#)

[Introduction](#)

[Purpose](#)

[Methodology](#)

[Core Principles](#)

[Part 1: subst — A Simple Find-and-Replace CLI Tool](#)

[1.1. Product Requirement Document (PRD)](#)

[1.1.1. Objective & Purpose](#)

[1.1.2. Target Audience & User Personas](#)

[1.1.3. Functional Requirements & Acceptance Criteria](#)

[1.1.4. Non-Functional Requirements (NFRs)](#)

[1.1.5. Out of Scope](#)

[1.2. User Journey Map](#)

[1.3. Architecture Diagrams](#)

[1.3.1. Component Architecture](#)

[1.3.2. Data Flow Diagram](#)

[1.3.3. Operational Sequence Diagram](#)

[Part 2: json-fmt — A JSON Pretty-Printer and Validator](#)

[2.1. Product Requirement Document (PRD)](#)

[2.1.1. Objective & Purpose](#)

[2.1.2. Target Audience & User Personas](#)

[2.1.3. Functional Requirements & Acceptance Criteria](#)

[2.1.4. Non-Functional Requirements (NFRs)](#)

[2.1.5. Out of Scope](#)

[2.2. User Journey Map](#)

[2.3. Architecture Diagrams](#)

[2.3.1. Component Architecture](#)

[2.3.2. Data Flow Diagram](#)

[2.3.3. Operational Sequence Diagram](#)

[Part 3: ansi-strip — An ANSI Escape Sequence Stripping Library](#)

[3.1. Product Requirement Document (PRD)](#)

[3.1.1. Objective & Purpose](#)

[3.1.2. Target Audience & User Personas](#)

[3.1.3. Functional Requirements & Acceptance Criteria](#)

[3.3.2. State Machine Logic](#)

[4.3. Architecture Diagrams](#)

[4.3.1. Macro Crate Architecture](#)

[4.3.2. Code Generation Flow](#)

[4.3.3. Macro Expansion Sequence](#)

[Part 5: line-count — A Source Code Line Counter](#)

[5.1. Product Requirement Document (PRD)](#)

[5.1.1. Objective & Purpose](#)

[5.1.2. Target Audience & User Personas](#)

[5.1.3. Functional Requirements & Acceptance Criteria](#)

[5.1.4. Non-Functional Requirements (NFRs)](#)

[5.1.5. Out of Scope](#)

[5.2. User Journey Map](#)

[5.3. Architecture Diagrams](#)

[5.3.1. Component Architecture](#)

[5.3.2. Data Flow Diagram](#)

[5.3.3. Operational Sequence Diagram](#)

[Conclusion and Strategic Recommendations](#)

[Summary of Patterns](#)

[Implementation Priority](#)

[Future Expansion](#)

[Works cited](#)

# **Pre-Development Specification for Minimalist Rust Utilities**

## **Introduction**

### **Purpose**

This report provides a complete and actionable pre-development documentation suite for five distinct, small-scale open-source software projects to be implemented in the Rust programming language. The objective is to furnish a comprehensive blueprint for each project, encompassing a detailed Product Requirement Document (PRD), a User Journey Map, and a set of architectural diagrams. This collection of documents is intended to serve as the single source of truth for any developer undertaking the implementation, ensuring clarity of purpose, a well-defined scope, and a sound architectural foundation from the outset.

### **Methodology**

A standardized framework is applied to each of the five projects to ensure consistency and rigor. This framework involves the systematic creation of three core artifacts:

1. **Product Requirement Document (PRD):** This document defines the *what* and *why* of the project, outlining its objectives, target users, functional and non-functional requirements, and explicit scope limitations. Its structure is informed by established industry best practices for creating clear and effective product specifications.1
2. **User Journey Map:** This visualizes the end-user's interaction with the product from initial awareness to adoption and advocacy. For developer-focused tools, this journey is unique, emphasizing technical evaluation, integration friction, and the "aha!" moment of solving a real-world problem.3 Each map is rendered using Mermaid syntax for clarity and ease of integration into documentation systems.5
3. **Architecture Diagrams:** Three distinct diagrams—Component, Data Flow, and Sequence—are provided for each project. This multi-perspective approach offers a holistic view of the system's design, from its static structure to its dynamic operational flow.

### **Core Principles**

A critical factor distinguishing successful Rust command-line utilities is not mere functionality but superior ergonomics. An analysis of the Rust ecosystem reveals that the most celebrated tools, such as ripgrep, bat, and fd, have gained widespread adoption because they fundamentally enhance common developer workflows.7 These tools are not just replacements for their Unix counterparts (

grep, cat, find); they are significant improvements that offer tangible benefits in performance, usability, and interface design.8 The praise for these utilities centers on their speed, intuitive syntax highlighting, and smarter default behaviors.

The common thread is a relentless focus on **Developer Experience (DX)**. Success in the crowded space of command-line tools is achieved by reducing friction and adding palpable value to frequent, everyday tasks. This principle, "The Ergonomic Advantage," serves as the guiding philosophy for the specifications within this report. The objective for each proposed utility is not merely to function correctly but to provide an experience that is efficient, intuitive, and ultimately, a pleasure to use.

## **Part 1: subst — A Simple Find-and-Replace CLI Tool**

This section details the specifications for subst, a minimalist, sed-like utility engineered for high-performance, literal string substitution. Its design prioritizes simplicity and seamless integration into shell pipelines.

### **1.1. Product Requirement Document (PRD)**

#### **1.1.1. Objective & Purpose**

To provide developers with a blazingly fast, simple, and intuitive command-line tool for non-regular-expression, find-and-replace operations on streams of text. The utility is explicitly designed to be a more ergonomic and often faster alternative to sed for simple substitution tasks, directly embodying the "Ergonomic Advantage" principle by optimizing a common, specific workflow.

#### **1.1.2. Target Audience & User Personas**

The primary user is a developer who is deeply integrated into the command-line environment and values efficiency and performance in their tooling.

* **Primary Persona: "Alex, the Command-Line Connoisseur"**
  + **Role:** Backend or Systems Developer.
  + **Behavior:** Alex spends a significant portion of the workday in a terminal, building, testing, and deploying software. They are proficient in shell scripting and frequently construct complex data processing pipelines using tools like grep, awk, and curl.
  + **Needs & Goals:** Alex seeks tools that are performant, reliable, and have a minimal cognitive overhead. They are often frustrated by the cumbersome syntax of traditional Unix tools for simple tasks and are drawn to the modern, fast alternatives emerging from the Rust ecosystem.7 They need  
    subst to integrate flawlessly into a pipeline (e.g., cat log.txt | subst 'ERROR' 'WARN' > new\_log.txt) without slowing down their workflow.

#### **1.1.3. Functional Requirements & Acceptance Criteria**

For a command-line utility, the user interface is defined by its arguments, I/O behavior, and exit codes. To eliminate ambiguity and provide a clear specification for implementation and testing, these behaviors are best defined in a structured format. The following table outlines the functional requirements and their corresponding acceptance criteria, forming an unambiguous contract for the tool's behavior.2

| Requirement ID | Feature/Requirement Description | Acceptance Criteria |
| --- | --- | --- |
| FR-SUB-01 | Argument Parsing | - MUST accept exactly two positional arguments: SEARCH\_STRING and REPLACEMENT\_STRING. - MUST return a non-zero exit code and a helpful error message to stderr if the argument count is not two. |
| FR-SUB-02 | Stdin Handling | - MUST read data from standard input (stdin). - MUST handle large input streams efficiently by processing line-by-line, without loading the entire stream into memory at once. |
| FR-SUB-03 | Core Substitution Logic | - MUST replace the *first* occurrence of SEARCH\_STRING with REPLACEMENT\_STRING on each line. - MUST perform a case-sensitive, literal (non-regex) string comparison. |
| FR-SUB-04 | Stdout Handling | - MUST write the processed output to standard output (stdout). - MUST pass through lines that do not contain SEARCH\_STRING unchanged. |
| FR-SUB-05 | Error Handling | - MUST handle I/O errors gracefully (e.g., a broken pipe when the consumer process terminates early) and exit with a non-zero status code. |

#### **1.1.4. Non-Functional Requirements (NFRs)**

* **NFR-SUB-01 (Performance):** The tool's performance must be a key feature. It should be benchmarked against the equivalent sed -e 's/SEARCH/REPLACE/' command and demonstrate comparable or superior throughput, particularly on large input files. This aligns with the high-performance characteristic expected of Rust CLI tools.7
* **NFR-SUB-02 (Portability):** The tool must compile and run correctly on major operating systems: Linux, macOS, and Windows.
* **NFR-SUB-03 (Code Size):** The entire implementation must be constrained to under 300 lines of code (LOC), adhering to the project's core constraint.

#### **1.1.5. Out of Scope**

To maintain simplicity and focus, the following features are explicitly excluded from this version:

* Regular expression (regex) support.
* In-place file editing (e.g., a -i flag).
* Global replacement (replacing all occurrences on a line).
* Any command-line flags or options (e.g., for case-insensitivity).

### **1.2. User Journey Map**

The journey of a developer adopting a new CLI tool is a funnel of increasing commitment, moving from initial problem awareness to eventual advocacy. This map visualizes that process for subst, highlighting critical stages and potential friction points.3 The "Aha!" moment is the point where the tool's value becomes immediately apparent, solidifying the user's decision to adopt it.

Code snippet

flowchart TD  
 subgraph "Awareness"  
 A[Problem: "I need to do a simple find-and-replace in a script. `sed` syntax is annoying for this."] --> B{Search: "rust simple sed alternative", "fast string replace cli"}  
 end  
 subgraph "Evaluation"  
 B --> C  
 C --> D{Onboarding: Reads README, sees simple usage: `cat file | subst old new`}  
 D --> E[Action: `cargo install subst` and tries it on a test file]  
 E --> F{"Aha!" Moment: "Wow, that was simple and fast."}  
 F --> G{Pain Point Check: "Is it faster than `sed`? Does it handle edge cases?"}  
 end  
 subgraph "Adoption & Retention"  
 F --> H[Integration: Uses `subst` in a personal shell script]  
 H --> I  
 end  
 subgraph "Advocacy"  
 I --> J  
 end  
  
 style F fill:#d4edda,stroke:#155724

### **1.3. Architecture Diagrams**

The following diagrams provide three distinct views of the subst architecture, illustrating its components, data flow, and operational sequence.11

#### **1.3.1. Component Architecture**

This diagram shows the high-level static components of the application and their relationships. It defines the primary modules or logical units of the binary.

Code snippet

graph TD  
 subgraph "subst Binary"  
 A[CLI Argument Parser] -- "Config" --> B(Core Logic)  
 C -- "Input Lines" --> B  
 B -- "Output Lines" --> D  
 B -- "Errors" --> E  
 end

#### **1.3.2. Data Flow Diagram**

This diagram traces the path of data as it moves through the system, from initial user input to final output. It emphasizes the transformation of data at each stage.

Code snippet

graph TD  
 A(User Input via Shell) --> B[stdin stream]  
 B --> C{Reads one line into buffer}  
 C --> D{Performs substitution on buffer}  
 D --> E[stdout stream]  
 E --> F(Terminal/Next Command)

#### **1.3.3. Operational Sequence Diagram**

This diagram illustrates the time-ordered sequence of interactions between the user, the operating system, and the subst process during a typical execution.

Code snippet

sequenceDiagram  
 participant User  
 participant OS  
 participant subst  
 User->>OS: `cat data.txt | subst 'foo' 'bar'`  
 OS->>subst: Spawns process with args and piped stdin  
 subst->>subst: Parse args ('foo', 'bar')  
 loop For each line from stdin  
 OS->>subst: Provides line  
 subst->>subst: Apply substitution logic  
 subst->>OS: Write processed line to stdout  
 end  
 subst->>OS: Exits with code 0

## **Part 2: json-fmt — A JSON Pretty-Printer and Validator**

This section provides the specifications for json-fmt, a utility designed to read a JSON string, validate its structural integrity, and output a human-readable, pretty-printed version.

### **2.1. Product Requirement Document (PRD)**

#### **2.1.1. Objective & Purpose**

To provide developers with a fast, lightweight, and reliable command-line tool for validating and formatting JSON data. This tool is essential for workflows involving API development and data inspection, serving as a minimal, single-purpose alternative to more complex tools like jq when only formatting is required.

#### **2.1.2. Target Audience & User Personas**

The primary user is any developer who regularly interacts with JSON data in a terminal environment.

* **Primary Persona: "Priya, the API Developer"**
  + **Role:** Full-Stack or Backend Developer.
  + **Behavior:** Priya frequently uses curl to test API endpoints, which often return minified, single-line JSON responses. She needs a quick and dependable way to make this output readable for debugging and verification directly in her terminal.
  + **Needs & Goals:** Priya's workflow is heavily reliant on shell pipelines (e.g., curl... | tool). She needs json-fmt to be a "good citizen" in this environment: it must be fast, handle stdin correctly, and provide clear error feedback when the input is not valid JSON.12

#### **2.1.3. Functional Requirements & Acceptance Criteria**

The tool's contract is defined by its ability to parse JSON correctly and provide predictable output for both valid and invalid inputs.

| Requirement ID | Feature/Requirement Description | Acceptance Criteria |
| --- | --- | --- |
| FR-JSN-01 | Input Handling | - MUST read a single, continuous stream of data from stdin. The tool should expect to read the entire JSON object from its input before processing. |
| FR-JSN-02 | JSON Parsing & Validation | - MUST parse the input stream using a robust JSON library (e.g., serde\_json). - If parsing succeeds, the tool MUST proceed to the formatting stage. - If parsing fails, the tool MUST exit with a non-zero status code and print a descriptive error (e.g., "Invalid JSON at line X, col Y") to stderr. |
| FR-JSN-03 | JSON Formatting | - MUST output a pretty-printed version of the valid JSON to stdout. - The default indentation for the formatted output MUST be 2 spaces. |
| FR-JSN-04 | Error Handling | - MUST handle I/O errors gracefully (e.g., failure to read from stdin). |

#### **2.1.4. Non-Functional Requirements (NFRs)**

* **NFR-JSN-01 (Performance):** The tool must be benchmarked against jq '.' and demonstrate significantly faster performance for the task of parsing and pretty-printing.
* **NFR-JSN-02 (Dependencies):** The implementation should rely on a well-vetted, community-trusted JSON library such as serde\_json to ensure correctness and security.
* **NFR-JSN-03 (Code Size):** The implementation must be under 300 LOC.

#### **2.1.5. Out of Scope**

* JSON querying, filtering, or data extraction (this is the domain of jq).
* In-place editing of files.
* Configurable options such as indentation width or output colorization.

### **2.2. User Journey Map**

This journey focuses on the immediate feedback loop a developer experiences when working with APIs. The tool's value is in its ability to instantly transform unreadable data into a structured, understandable format.

Code snippet

flowchart TD  
 subgraph "Awareness & Evaluation"  
 A --> B{Search: "cli json pretty print rust"}  
 B --> C  
 C --> D[Action: `curl... | json-fmt`]  
 end  
 subgraph "Adoption"  
 D --> E{"Aha!" Moment: The unreadable blob is instantly formatted correctly.}  
 E --> F{Validation Test: Tries it on an invalid JSON file.}  
 F --> G  
 G --> H[Habit: Aliases `json-fmt` to `jf` in shell profile for quick access.]  
 end  
 style E fill:#d4edda,stroke:#155724  
 style G fill:#d4edda,stroke:#155724

### **2.3. Architecture Diagrams**

The architecture revolves around a deserialize-then-serialize pattern, with distinct paths for success and failure.

#### **2.3.1. Component Architecture**

This diagram shows the logical separation of concerns within the binary, highlighting the roles of I/O, deserialization, serialization, and error handling.

Code snippet

graph TD  
 subgraph "json-fmt Binary"  
 A[I/O Handler] -- "Raw String" --> B(JSON Deserializer via `serde\_json`)  
 B -- "Valid JSON Value" --> C(JSON Serializer via `serde\_json`)  
 B -- "Parse Error" --> D[Error Formatter]  
 C -- "Formatted String" --> A  
 D -- "Error Message" --> A  
 end  
 A -- "Input/Output" --> E((Terminal))

#### **2.3.2. Data Flow Diagram**

This diagram shows the transformation of data from a raw byte stream into structured Rust types and back into a formatted string.

Code snippet

graph TD  
 A(curl response) --> B[stdin]  
 B --> C{Read entire stdin into a String}  
 C --> D{`serde\_json::from\_str`}  
 D -- "Ok(Value)" --> E{`serde\_json::to\_string\_pretty`}  
 D -- "Err(e)" --> F{Format error message}  
 E --> G[stdout]  
 F --> H[stderr]

#### **2.3.3. Operational Sequence Diagram**

This sequence diagram details the conditional logic based on the validity of the JSON input, showing the interaction with the serde\_json library.

Code snippet

sequenceDiagram  
 participant User  
 participant json\_fmt  
 participant serde\_json  
 User->>json\_fmt: Pipes JSON string to stdin  
 json\_fmt->>json\_fmt: Reads all of stdin into a buffer  
 json\_fmt->>serde\_json: `from\_str(&buffer)`  
 alt Input is Valid JSON  
 serde\_json-->>json\_fmt: Returns `Ok(Value)`  
 json\_fmt->>serde\_json: `to\_string\_pretty(&Value)`  
 serde\_json-->>json\_fmt: Returns `Ok(String)`  
 json\_fmt->>User: Writes formatted string to stdout  
 else Input is Invalid JSON  
 serde\_json-->>json\_fmt: Returns `Err(error)`  
 json\_fmt->>User: Writes error message to stderr  
 end

## **Part 3: ansi-strip — An ANSI Escape Sequence Stripping Library**

This section details ansi-strip, a library crate (not a binary) that provides a single, efficient function to remove ANSI escape codes from strings. The idea for such a utility is drawn from community wishlists for small, focused Rust tools.13

### **3.1. Product Requirement Document (PRD)**

#### **3.1.1. Objective & Purpose**

To provide Rust developers with a simple, zero-dependency, and high-performance utility function for cleaning ANSI escape codes from strings. This functionality is critical when logging raw output from command-line tools to files, or when displaying such output in environments that do not correctly render ANSI color and formatting codes.

#### **3.1.2. Target Audience & User Personas**

The user of a library crate is another developer integrating it into a larger project.

* **Primary Persona: "Sam, the Library Author"**
  + **Role:** Rust Developer, Open-Source Contributor.
  + **Behavior:** Sam is building a larger Rust application, such as a test runner, a continuous integration tool, or a logging framework. Their application needs to capture and process the stdout and stderr streams from other command-line programs.
  + **Needs & Goals:** Sam needs a reliable and performant way to sanitize this captured output before writing it to a log file or displaying it in a plain text UI. They prioritize crates that are lightweight (zero-dependency is ideal), well-documented, and have a stable API to avoid future maintenance burdens.

#### **3.1.3. Functional Requirements & Acceptance Criteria**

The PRD for a library must focus on the public API contract: the function signatures, their documented behavior, and their performance characteristics. This is the "user interface" for the developer consuming the crate.

| Requirement ID | Feature/Requirement Description | Acceptance Criteria |
| --- | --- | --- |
| FR-ANS-01 | Public API | - MUST expose a single public function, for example: pub fn strip(input: &str) -> String. - The function MUST be thoroughly documented with clear examples in the lib.rs file, making it discoverable and understandable on docs.rs. |
| FR-ANS-02 | Core Logic | - MUST correctly identify and remove standard ANSI/VT100 escape sequences, specifically Control Sequence Introducer (CSI) sequences of the form `\x1b --> B{Search: "rust remove ansi escape codes crate"} |

B --> C  
 C --> D{Evaluation: Checks docs.rs for API, sees it's one simple function. Checks for dependencies (sees zero). Looks at download numbers.}  
end  
subgraph "Integration & Adoption"  
 D --> E[Action: `cargo add ansi-strip`]  
 E --> F[Implementation: `let clean\_output = ansi\_strip::strip(&raw\_output);`]  
 F --> G{"Aha!" Moment: "It just works." The code is clean and the problem is solved.}  
 G --> H  
end  
style G fill:#d4edda,stroke:#155724

### 3.3. Architecture Diagrams  
  
For a simple library, the architecture focuses on the module structure, the internal algorithm, and the external API contract.  
  
#### 3.3.1. Library Module Structure  
  
This diagram shows the separation between the public-facing API and the internal implementation details, a key aspect of good library design.  
  
```mermaid  
graph TD  
 subgraph "ansi-strip crate"  
 A(Public Module `lib.rs`) -- "Exposes" --> B[pub fn strip(input: &str)]  
 B -- "Calls" --> C(Internal state-machine logic)  
 D(Test Module `tests.rs`) -- "Tests" --> B  
 end

#### **3.3.2. State Machine Logic**

The core of the library is a simple state machine for parsing the input string. This is the most efficient architectural pattern for this task.

Code snippet

graph TD  
 A(Start) --> B{Read char};  
 B -- "Not ESC (`\x1b`)" --> C[Append char to output];  
 C --> B;  
 B -- "ESC (`\x1b`)" --> D{Enter 'Escape' State};  
 D --> E{Read next char};  
 E -- "'  
  
### 4.1. Product Requirement Document (PRD)  
  
#### 4.1.1. Objective & Purpose  
  
To reduce repetitive boilerplate code for Rust developers by automatically generating a standard `::new()` constructor function for any struct to which the macro is applied. This enhances developer productivity and maintains code cleanliness by abstracting away a common, mechanical task.[16]  
  
#### 4.1.2. Target Audience & User Personas  
  
The user is any Rust developer who frequently defines new data structures.  
  
\* \*\*Primary Persona: "David, the DRY Developer"\*\*  
 \* \*\*Role:\*\* Rust Software Engineer.  
 \* \*\*Behavior:\*\* David adheres to the "Don't Repeat Yourself" (DRY) principle. He writes a lot of Rust code and finds the manual creation of `pub fn new(...) -> Self { Self {... } }` for every new struct to be tedious and error-prone, especially for structs with many fields.  
 \* \*\*Needs & Goals:\*\* David values tools and language features that automate repetitive tasks. He wants to use `StructNew` to simply declare his intent (`#`) and let the compiler handle the mechanical implementation, allowing him to focus on more complex business logic.  
  
#### 4.1.3. Functional Requirements & Acceptance Criteria  
  
The "user interface" of a procedural macro is the syntax it enables and, crucially, the quality of the compiler errors it produces when misused. The PRD must therefore focus on this compile-time interaction.  
  
| Requirement ID | Feature/Requirement Description | Acceptance Criteria |  
| :--- | :--- | :--- |  
| FR-MAC-01 | Derive Macro Implementation | - MUST be implemented as a procedural derive macro, invoked with `#`.<br>- MUST be packaged in a dedicated crate with the `proc-macro = true` setting in its `Cargo.toml`.[14] |  
| FR-MAC-02 | Code Generation | - MUST generate a public function named `new`.<br>- The `new` function's arguments MUST match the struct's fields in both name and type, and appear in the order they are defined in the struct.<br>- The `new` function MUST have a return type of `Self`. |  
| FR-MAC-03 | Compatibility | - MUST work correctly on structs defined with named fields.<br>- MUST correctly handle generic type parameters and lifetime parameters present on the struct definition.<br>- MUST handle a wide variety of field types, including primitives, standard library types (e.g., `String`, `Vec<T>`), and other user-defined types. |  
| FR-MAC-04 | Error Handling | - MUST produce a clear and helpful compile-time error via `compile\_error!` if the `#` attribute is applied to an `enum`, `union`, or a tuple struct. |  
  
#### 4.1.4. Non-Functional Requirements (NFRs)  
  
\* \*\*NFR-MAC-01 (Dependencies):\*\* The implementation will require the `syn` and `quote` crates as dependencies. This is standard and accepted practice for writing robust procedural macros in Rust.[16, 17]  
\* \*\*NFR-MAC-02 (Compile-Time Performance):\*\* The macro expansion process should be efficient and not introduce a noticeable slowdown to the overall compilation time for typical projects.  
  
#### 4.1.5. Out of Scope  
  
\* Generation of a `::default()` implementation. The `#` macro already serves this purpose.  
\* Support for attributes to modify behavior (e.g., an attribute to skip a field or provide a default value), as seen in more complex builder-pattern macros.[15]  
\* Support for `union` types.  
  
### 4.2. User Journey Map  
  
The user journey for a procedural macro is unique as it occurs entirely within the developer's code editor and during the compilation process. The key moments are the ease of use and the clarity of feedback from the compiler.  
  
```mermaid  
flowchart TD  
 subgraph "Authoring Time"  
 A[Problem: "I need a constructor for this new struct."] --> B` above the struct definition]  
 B --> C  
 end  
 subgraph "Compile Time"  
 C --> D{Compiler runs the `StructNew` proc-macro}  
 D -- "Success" --> E  
 D -- "Failure (e.g., on an enum)" --> F  
 end  
 subgraph "Usage"  
 E --> G  
 end  
 style C fill:#d4edda,stroke:#155724

### **4.3. Architecture Diagrams**

The architecture of a procedural macro is centered on the transformation of Rust code at compile time.

#### **4.3.1. Macro Crate Architecture**

This diagram illustrates the relationship between the compiler, the macro crate, and its essential dependencies, syn and quote.

Code snippet

graph TD  
 A[Compiler (`rustc`)] -- "Input TokenStream" --> B(struct\_new\_macro)  
 subgraph "struct-new (proc-macro crate)"  
 B -- "Uses for Parsing" --> C[syn crate]  
 C -- "Produces AST" --> D{Macro Logic}  
 D -- "Uses for Code Gen" --> E[quote crate]  
 E -- "Produces" --> F  
 end  
 F -- "Generated Code" --> B  
 B --> A

#### **4.3.2. Code Generation Flow**

This diagram shows the logical flow of transforming the input struct definition into the output impl block.

Code snippet

graph TD  
 A[Input: `struct Point { x: i32, y: i32 }`] --> B{Parse with `syn::parse`}  
 B --> C  
 B --> D[Extract: Fields (`x: i32`, `y: i32`)]  
 C & D --> E{Construct function signature and body with `quote!`}  
 E --> F

#### **4.3.3. Macro Expansion Sequence**

This sequence diagram details the step-by-step interaction between the compiler and the macro during the build process.

Code snippet

sequenceDiagram  
 participant Developer  
 participant rustc as Compiler  
 participant StructNewMacro as Macro  
 Developer->>rustc: `cargo build`  
 rustc->>rustc: Encounters `#` on `struct Point`  
 rustc->>StructNewMacro: Invokes macro with `Point`'s TokenStream  
 activate StructNewMacro  
 StructNewMacro->>StructNewMacro: Parse tokens into an Abstract Syntax Tree (AST) using `syn`  
 StructNewMacro->>StructNewMacro: Generate new tokens for the `impl` block and `new` function using `quote`  
 StructNewMacro-->>rustc: Returns generated TokenStream  
 deactivate StructNewMacro  
 rustc->>rustc: Splices the returned tokens into the code for compilation  
 rustc-->>Developer: Build successful

## **Part 5: line-count — A Source Code Line Counter**

This section details line-count, a simple utility inspired by more comprehensive tools like tokei.7 It is designed to analyze a single source file and report a breakdown of its lines.

### **5.1. Product Requirement Document (PRD)**

#### **5.1.1. Objective & Purpose**

To provide a simple, fast command-line tool that analyzes a single file and provides a count of total lines, code lines, blank lines, and comment lines. It serves as both a useful learning project for Rust beginners and a lightweight utility for developers wanting a quick high-level analysis of a file's composition.

#### **5.1.2. Target Audience & User Personas**

The target user is a developer who needs a quick, informal measure of a file's structure.

* **Primary Persona: "Leo, the Learner"**
  + **Role:** Junior Developer or a developer new to a specific codebase.
  + **Behavior:** Leo is exploring an unfamiliar project and wants to get a quick sense of individual files. Before diving deep into the logic, they want to answer high-level questions like: "Is this file heavily commented? Is it dense with code, or does it have a lot of whitespace?"
  + **Needs & Goals:** Leo needs a tool that is trivial to install and run. The output must be simple, clear, and immediate, providing a useful heuristic about the file's nature without the complexity of a full static analysis tool.

#### **5.1.3. Functional Requirements & Acceptance Criteria**

The requirements focus on correct file handling and a clear, simple set of rules for line classification.

| Requirement ID | Feature/Requirement Description | Acceptance Criteria |
| --- | --- | --- |
| FR-LNC-01 | Argument Parsing | - MUST accept exactly one positional argument: FILE\_PATH. - MUST return a non-zero exit code and a user-friendly error message to stderr if the specified file does not exist or cannot be read due to permissions. |
| FR-LNC-02 | File Reading | - MUST read the specified file efficiently on a line-by-line basis to handle large files without high memory consumption. |
| FR-LNC-03 | Line Classification | - A line containing only whitespace characters (spaces, tabs) is classified as a "Blank Line". - A line whose first non-whitespace characters are // or /\* is classified as a "Comment Line". - Any other non-blank line is classified as a "Code Line". - "Total Lines" is the sum of all physical lines read from the file. |
| FR-LNC-04 | Output Formatting | - MUST print a human-readable summary to stdout. - The output MUST include distinct labels and their corresponding counts for Total Lines, Code Lines, Blank Lines, and Comment Lines. |

#### **5.1.4. Non-Functional Requirements (NFRs)**

* **NFR-LNC-01 (Performance):** The tool should process files quickly by streaming the content rather than loading the entire file into memory.
* **NFR-LNC-02 (Simplicity):** The classification logic must remain simple and heuristic-based. It should not attempt to be a full language parser.
* **NFR-LNC-03 (Code Size):** The implementation must be under 300 LOC.

#### **5.1.5. Out of Scope**

* Recursive analysis of directories or multiple files.
* Language-specific comment syntax beyond // and /\* (e.g., # for Python, -- for SQL).
* Distinguishing between documentation comments (e.g., ///) and regular comments.
* Machine-readable output formats like JSON or CSV.

### **5.2. User Journey Map**

The journey for this tool is short and transactional, focused on answering a single, immediate question for the user.

Code snippet

flowchart TD  
 subgraph "Discovery & Use"  
 A[Problem: "I wonder how complex this `main.rs` file really is."] --> B[Action: `cargo install line-count`]  
 B --> C[Execution: `line-count src/main.rs`]  
 C --> D  
 end  
 subgraph "Insight"  
 D --> E{"Aha!" Moment: "Okay, it's mostly code, not too many comments. I can get a quick feel for a file's density with this."}  
 end  
 style E fill:#d4edda,stroke:#155724

### **5.3. Architecture Diagrams**

The architecture is a straightforward file processing pipeline that maintains a simple state (the counters).

#### **5.3.1. Component Architecture**

This diagram shows the main logical components and how they collaborate to produce the final report.

Code snippet

graph TD  
 subgraph "line-count Binary"  
 A[Arg Parser] -- "File Path" --> B  
 B -- "Line by Line" --> C[Line Classifier]  
 C -- "Updates" --> D(Counter State)  
 A -- "Triggers on Completion" --> E  
 E -- "Reads from" --> D  
 E -- "Prints to" --> F((stdout))  
 end

#### **5.3.2. Data Flow Diagram**

This diagram traces the flow from the file on disk, through the classification logic, to the final aggregated counts.

Code snippet

graph TD  
 A --> B{Open file stream}  
 B --> C{Read one line}  
 C --> D{Classify line (Code, Comment, Blank)}  
 D --> E{Increment appropriate counter}  
 subgraph "Loop until EOF"  
 C -- More lines? --> C  
 end  
 C -- No more lines --> F{Format and print final counts}

#### **5.3.3. Operational Sequence Diagram**

This diagram provides a step-by-step trace of the program's execution for a single run.

Code snippet

sequenceDiagram  
 participant User  
 participant line\_count  
 User->>line\_count: Executes with file path `src/main.rs`  
 activate line\_count  
 line\_count->>line\_count: Parses argument, gets file path  
 line\_count->>line\_count: Opens file for reading  
 line\_count->>line\_count: Initializes counters (total, code, etc.) to zero  
 loop For each line in file  
 line\_count->>line\_count: Reads line  
 line\_count->>line\_count: Trims whitespace from the line  
 alt Line is empty after trim  
 line\_count->>line\_count: Increment blank\_counter  
 else Line starts with '//' or '/\*' after trim  
 line\_count->>line\_count: Increment comment\_counter  
 else  
 line\_count->>line\_count: Increment code\_counter  
 end  
 line\_count->>line\_count: Increment total\_counter  
 end  
 line\_count->>line\_count: Formats results into a summary string  
 line\_count->>User: Prints summary to stdout  
 deactivate line\_count

## **Conclusion and Strategic Recommendations**

### **Summary of Patterns**

The detailed analysis of these five distinct projects reveals recurring architectural patterns tailored to their specific use cases. For the command-line interface (CLI) tools (subst, json-fmt, line-count), a consistent three-stage pipeline architecture emerges: **1. Parse Arguments**, **2. Process Stream**, and **3. Format Output**. This model proves effective for tools that conform to the Unix philosophy of doing one thing well and operating on standard I/O streams.

For the ansi-strip library crate, the architectural focus shifts from I/O pipelines to the public API contract. The primary pattern is the exposure of a **minimal, well-documented API surface** that hides a more complex internal implementation (in this case, a state machine). This prioritizes ease of integration and long-term stability for consumer crates.

Finally, for the StructNew procedural macro, the architecture is defined by the compiler's metaprogramming hooks. The pattern is a robust **TokenStream-to-TokenStream transformation**, heavily reliant on the syn and quote crates to parse the input Rust code into an Abstract Syntax Tree (AST) and generate new, valid Rust code as output.

### **Implementation Priority**

Based on an assessment of complexity and the progressive introduction of core Rust concepts, the following implementation order is recommended. This path allows for a gradual increase in difficulty, providing a structured learning curve.

1. **subst:** The most straightforward project, focusing on fundamental Rust concepts like argument parsing and basic, line-by-line I/O.
2. **line-count:** Introduces simple state management (the counters) and more nuanced string processing logic.
3. **json-fmt:** Adds the complexity of managing external dependencies (serde\_json) and handling more structured, multi-faceted errors from a third-party library.
4. **ansi-strip:** Shifts the focus to library design principles, zero-dependency code, and the implementation of a more formal algorithm (a state machine).
5. **StructNew:** The most complex project, requiring a deep dive into Rust's metaprogramming ecosystem, including the syn and quote libraries and the intricacies of procedural macros.

### **Future Expansion**

While each project is designed to be minimal and complete, they all possess clear avenues for future expansion. This provides a roadmap for evolving these simple utilities into more powerful tools.

* **subst (v2.0):** Introduce a -g flag for "global" replacement (replacing all occurrences on each line, not just the first). This would require a change in the core substitution logic from replacen to replace.
* **json-fmt (v2.0):** Add a --color or -c flag to enable syntax-highlighted JSON output, and a --indent <WIDTH> flag to allow for configurable indentation.
* **ansi-strip (v2.0):** Expand the library to handle a wider range of terminal escape codes beyond basic CSI sequences, potentially including OSC (Operating System Command) sequences.
* **StructNew (v2.0):** Enhance the macro with helper attributes, such as #[new(skip)] to exclude a field from the constructor (requiring it to have a Default implementation) or #[new(default)] to use a field's default value instead of requiring it as an argument. This would align it more closely with the advanced factory pattern macro described in the research.15
* **line-count (v2.0):** Add support for directory traversal to analyze all files within a project. This would also necessitate introducing language-aware comment detection (e.g., recognizing # in Python files and // in Rust files) to provide more accurate, language-specific counts.

#### Works cited

1. Product Requirements Document Template - PRD Template | Slite.com, accessed on August 9, 2025, <https://slite.com/templates/product-requirements-document>
2. Best 10 PRD: Product Requirements Doc Templates for Product Analysts - Notion, accessed on August 9, 2025, <https://www.notion.com/templates/collections/best-10-prd-product-requirements-doc-templates-for-product-analysts>
3. Mapping the developer journey, accessed on August 9, 2025, <https://developerrelations.com/guides/mapping-the-developer-journey/>
4. Mapping the developer journey - Developer Relations, accessed on August 9, 2025, <https://developerrelations.com/guides/mapping-the-developer-journey>
5. Getting Started - Mermaid Chart, accessed on August 9, 2025, <https://docs.mermaidchart.com/mermaid-oss/intro/getting-started.html>
6. mermaid-js/mermaid: Generation of diagrams like flowcharts or sequence diagrams from text in a similar manner as markdown - GitHub, accessed on August 9, 2025, <https://github.com/mermaid-js/mermaid>
7. A curated list of command-line utilities written in Rust - GitHub Gist, accessed on August 9, 2025, <https://gist.github.com/sts10/daadbc2f403bdffad1b6d33aff016c0a>
8. sts10/rust-command-line-utilities: A curated list of command ... - GitHub, accessed on August 9, 2025, <https://github.com/sts10/rust-command-line-utilities>
9. Getting started - Command Line Applications in Rust, accessed on August 9, 2025, <https://rust-cli.github.io/book/index.html>
10. Free Product Requirement Document Templates | Smartsheet, accessed on August 9, 2025, <https://www.smartsheet.com/content/free-product-requirements-document-template>
11. Mermaid FlowChart Basic Syntax - Mermaid Chart - Create complex ..., accessed on August 9, 2025, <https://mermaid.js.org/syntax/flowchart.html>
12. Writing command line utilities in Rust - Tonsser Tech Blog, accessed on August 9, 2025, <https://techblog.tonsser.com/posts/writing-command-line-utilities-in-rust>
13. my-rust-lists/rust-project-ideas.md at master - GitHub, accessed on August 9, 2025, <https://github.com/brson/my-rust-lists/blob/master/rust-project-ideas.md>
14. Procedural Macros - The Rust Reference, accessed on August 9, 2025, <https://doc.rust-lang.org/reference/procedural-macros.html>
15. Simple steps to Procedural Macros in Rust | by Basillica - Medium, accessed on August 9, 2025, <https://basillica.medium.com/automating-the-factory-pattern-in-rust-with-procedural-macros-a00e8e6816c7>
16. Procedural Macros in Rust – A Handbook for Beginners - freeCodeCamp, accessed on August 9, 2025, <https://www.freecodecamp.org/news/procedural-macros-in-rust/>
17. Guide to Rust procedural macros | developerlife.com, accessed on August 9, 2025, <https://developerlife.com/2022/03/30/rust-proc-macro/>