ChatGPT

Designing a Beginner-Friendly Interactive CLI Experience

Building a CLI that is **approachable for beginners yet powerful for advanced users** requires careful UX design. This involves intuitive menus and prompts, clear and colorful output formatting, helpful error messages, and onboarding workflows that guide new users. Below, we present best practices (with examples from tools like GitHub CLI, npm, Docker, AWS, kubectl, Heroku, etc.) for designing an interactive CLI – in this case, a *Tableau Workbook Cleanup Agent* – that lets users select workbook files, review validation results, confirm cleanup actions, and see summaries of changes.

Menu Structures and Prompt Styles

Beginner-friendly CLIs often offer **interactive prompts and menus** to guide users through complex commands. Instead of forcing new users to recall flags or syntax, the tool can ask a series of questions. For example, upon launching the workbook cleanup CLI, it could display a menu of detected `.twb/.twbx` files and ask the user which workbooks to clean. This interactive "wizard" approach mimics GUI workflows by **leading users step-by-step** to success [1] [2].

Always allow a non-interactive alternative: Interactive prompts should be **optional**, not mandatory. The CLI must still support direct flags or arguments so that experienced users (or scripts) can skip the Q&A flow [3] [4]. As the Ubuntu CLI guidelines put it, *"if stdin is a tty then prompt... Never require a prompt though. The user needs to be able to automate the CLI in a script, so always allow them to override prompts"* [4]. For instance, our cleanup tool might have a `--file <name>` option to specify a workbook directly, bypassing the selection menu. *GitHub CLI (gh)* follows this pattern: running `gh` commands with no parameters launches an interactive mode, but providing the necessary flags executes the command immediately. If you run `gh pr create` with no flags, it will interactively prompt for a title, body, etc., showing defaults (like using the last commit message as a default title) [5]. If you supply all required info via flags (`-t` title and `-b` body), it **skips prompts entirely** and creates the PR directly [6]. This dual-mode design ensures the CLI is convenient for newcomers and efficient for power users.

Use clear prompt text and defaults: Each prompt should clearly state what input is needed and provide a sensible default when possible. For example, a *confirm* prompt should indicate the default choice in brackets (e.g. *"Proceed with cleanup? (y/N)"* meaning the default is "No") [7]. Defaults reduce friction for beginners – they can just hit Enter to accept. Use **yes/no confirmations** especially for destructive actions (e.g. *"Are you sure you want to overwrite the workbook? [default = no]"*), and default them to the safe option (no) to prevent accidents [7]. In our workbook agent, after listing validation issues, the CLI might ask *"Apply fixes to these 3 issues? [default = no]"* requiring a explicit *"y"* to proceed.

Support lists and multi-select menus: For selection of items (like choosing which workbooks to process), present a **list menu** or checkboxes rather than requiring the user to type filenames. Libraries like *Inquirer.js* (Node) or *prompt_toolkit* (Python) can display an interactive list where the user can move with arrow keys and press Enter to select [8]. For multiple selections, use checkboxes (multi-select) so a user can pick

several items intuitively (often by pressing Space to toggle each) [9] [10]. For example, an Inquirer.js prompt might ask: *"What would you like to clean up?"* with a checkbox list of workbooks. The prompt itself should be descriptive (e.g. *"Select workbooks to clean:"*) and if the list is long, providing a **filter** or search-as-you-type is a nice touch (Inquirer supports this, allowing users to type to narrow options) [8]. The interface should also handle cases where the user's input doesn't match any choice – e.g. by suggesting the closest match if they start typing something invalid [11].

- *Example prompt types:* Modern CLI frameworks support a variety of prompt styles:
- **List menu:** Offer a list of choices for single selection (e.g. list of files or actions). The user selects one option via arrow keys or number input [8]. *(Our CLI could list workbooks 1, 2, 3, ...)*
- **Multi-select (Checkboxes):** Let the user select multiple items from a list. For instance, select several workbooks to clean at once. Mark selected items with an `[x]`. Provide instructions like "(use Space to select, Enter to finish)" to aid newbies [9].
- **Input prompt:** Ask the user to type a free-form response (e.g. a file path, a new value for a setting) [12]. Validate the input format immediately – for example, if expecting a number or date, check that the input is valid and show an error if not [12].
- **Confirm prompt:** Yes/no questions. Show the default in the prompt text as mentioned (e.g. *"[default = no]"*) [7]. The CLI might use `[Y/n]` or `(y/N)` conventions to indicate which is default. Pressing Enter without typing uses the default.

Validate inputs on the fly: Interactive mode is an opportunity to **prevent errors before they happen**. If the user must enter something like a file path or number, the CLI should validate it *as soon as possible* and not let an invalid value through without warning. In GUI forms, users get immediate feedback for bad input; CLI prompts can do the same [13]. For example, if the cleanup tool asks for an output directory, it can check that the path exists and is writable. If the user inputs an invalid value, display a friendly error and re-prompt rather than proceeding and failing later [14]. Inquirer.js and similar libraries allow defining a `validate` function for prompts to enforce such rules [15]. This saves beginners from frustration by catching mistakes early.

Teach through the CLI itself: A clever onboarding pattern is to **echo the resulting command** that corresponds to the interactive choices the user made. Before executing, print the equivalent full command (with all flags) that the CLI is about to run [16]. For instance: *"Running:* `cleanup-tool --file "Sales Workbook.twbx" --fix-level full`*"*. This helps users learn the non-interactive usage by example, building their confidence to use flags directly next time [16]. It demystifies what the interactive wizard is doing. Always give users the ability to confirm or cancel at that point (e.g. *"Press Enter to proceed or Ctrl+C to cancel"*).

Discoverability of features: Interactive prompts act as built-in documentation, revealing what options exist. Instead of reading a manual, the user is *asked* about optional features ("Compress the output? Y/N", "Choose output format: [1] CSV [2] JSON"). This guides newbies through all relevant choices so they don't overlook anything important [17]. For example, if our CLI has an optional flag to compress the cleaned workbook, an interactive flow can simply ask *"Compress the workbook to reduce size? (y/N)"* rather than expecting the user to know about a `--compress` flag [17]. This approach puts **guardrails** on user input and prevents mistakes by constraining choices to valid ones [18].

Finally, design your CLI's commands and subcommands with consistency. Follow common CLI conventions for naming and syntax so that it "feels" familiar. For instance, use `--help` to show help text, use plural

nouns for grouping commands (like `kubectl get` uses resource type plural), and verbs for actions (Heroku's style guide recommends this structure: `heroku apps:create` where **topic** is plural noun and **command** is a verb) [19] [20]. Reusing established patterns means users don't have to re-learn the basics of using your CLI [21]. In short: **interactive menus and prompts should guide the novice**, while a logical structure and optional flags keep the CLI efficient for experienced users.

Output Formatting (Tables, Colors, and Spinners)

A well-designed CLI outputs information in a **structured, readable format**. This includes using colors or highlighting to draw attention, formatting data in tables or aligned columns, and showing progress indicators for long-running tasks. These elements make the tool feel polished and keep users (especially newcomers) confident about what's happening.

Figure: Example CLI output (Yarn package manager initializing a project). It uses colored text and emojis to differentiate types of messages – warnings in yellow, errors in red, success in green (with a emoji) – allowing users to understand status at a glance [22].

Use color and text styling wisely: Humans can parse colored output much faster than monochrome text. Successful CLIs like **Yarn** or **npm** print informational messages, warnings, and success confirmations in different colors (often following conventions like blue or cyan for info, yellow for warnings, red for errors, green for success) [22] [23]. For example, Yarn prints the word "warning" in yellow with a warning emoji, and prints a final " Done" in bright green with a sparkles emoji [22]. This way, a user's eye is immediately drawn to important lines – if they see yellow, they know something might need attention, and green at the end tells them the operation completed successfully [22]. As one author notes, *"when output is displayed this way, users can absorb much more information at a glance… a yellow 'warning' indicates something unexpected, whereas a green message with a sparkles emoji makes it clear everything went okay despite the warning"* [22]. Adding such visual cues can **delight the user and build confidence**, making the tool feel less like an old-school terminal and more like a friendly interface [24].

However, there are important guidelines to follow with color and embellishments [25] [26]:

- **Check capability and allow opt-out:** Not all terminals support colored or extended characters (e.g. Windows default console in some cases, or when output is being piped to a file). Your CLI should detect if color is supported, and either disable colored output or provide a `--no-color` flag or environment variable to disable it [27]. This ensures that users on any platform or using assistive technology don't see garbled `[33m` codes or literal emoji fallback text.
- **Use color consistently and sparingly:** Pick a standard scheme for your domain. For example, the Heroku CLI assigns consistent colors to certain noun types (apps, config keys, etc.) [28] so that those items are always highlighted. Use only a few distinct colors; if you use too many, they will **compete for attention** and nothing will stand out [26]. Often, a couple of colors plus variations (bold or dim text) are enough to create contrast [26]. Also reserve high-attention colors for truly important things (Heroku notes that yellow and red are typically for warnings and errors) [29]. In short, **if everything is highlighted, nothing is highlighted** – make selective use of color to encode meaning [30].
- **Don't rely on color alone:** Ensure that important information is also conveyed in text (for accessibility and greppability). For instance, print the words "ERROR" or "WARNING" in addition to coloring them, so if someone is color-blind or viewing a monochrome log, they can still identify

severity. If you use symbols or emojis, use them as enhancements, not replacements for words [31]. (e.g. showing "✓ Success" in green gives both a symbol and text). Moreover, avoid substituting common words entirely with emoji icons – users can't search log text for an emoji character as easily as a word [31]. A guideline is that output should remain *grep-able*; one should be able to `grep "ERROR"` and find error lines, which wouldn't be possible if you only displayed a red "✖" symbol without the word "ERROR" [30].

Format data in tables or aligned columns: When presenting structured data (lists of items, results of validation checks, etc.), consider a **tabular output** for clarity. Well-formatted tables with headers make it much easier for users (especially novices) to parse results than raw text or unaligned columns. For example, the AWS and Kubernetes CLIs print resources in columns (with headings like NAME, STATUS, AGE, etc.), which are easy to scan. The Heroku CLI team gives a great example of improving output formatting: originally `heroku regions` printed regions separated by category headings, but this made it hard to filter. They changed it to a tabular format with columns *ID, Location, Runtime* so that each region is one line, and it became easy to grep or scan [32] [33]. An excerpt of the improved output:

```
$ heroku regions
ID         Location          Runtime
─────────  ────────────────  ──────────────
eu         Europe            Common Runtime
us         United States     Common Runtime
frankfurt  Frankfurt, Germany Private Spaces
...        ...               ...
tokyo      Tokyo, Japan      Private Spaces
```

Aligning the columns and underlining headers (as above) creates a clean look. The Heroku style guide even provides utility functions to generate such tables, emphasizing how valuable they are [34]. The key is that **structured output helps both human users and scripts**. Users can visually scan columns, and if someone needs to script against the output, they can easily parse it. Ensure that **columns are separated by spaces (not just visually but actual whitespace)** so tools like `awk` or `cut` can work, and try not to break the alignment in future updates (adding new columns at the end is fine, but don't swap existing ones as it might break parsing) [35]. If the data is too wide for a standard terminal, consider truncating or wrapping sensibly, or offering an alternate output mode (like CSV or wide format).

For our workbook cleanup CLI, think about outputting validation results in a table form. For instance:

```
Workbook Name        Errors   Warnings   Fixes Suggested
──────────────────   ──────   ────────   ─────────────────
SalesReport.twbx     2        1          Fix formulas, Remove unused field
Marketing.twb        0        3          (No critical errors)
```

Such a summary gives a concise overview. The user could then select a particular workbook to see details, or the CLI could output a detailed log below.

Provide machine-readable output for advanced use: While pretty tables and colors are great for humans, advanced users might want JSON or other structured output for scripting. It's a best practice to offer a flag (like `--json`) to output the raw data in JSON or XML so that it can be consumed by other programs [36] [37]. Many CLIs (AWS, Azure, kubectl, etc.) support `--output json` or `-o yaml` for this reason. This does not detract from the default human-friendly output, it just gives more power to those who need it. The Heroku CLI, for example, prints a compact table by default, but `--json` yields a detailed JSON array with all fields (which you could pipe to `jq` for filtering) [38] [39]. The guiding principle: *provide beautiful, friendly output by default, but don't trap data in an unparseable form* [40]. Offering both means novices and experts are both happy.

Show progress and status for long tasks: A common UX failure in CLI tools is to leave the user staring at a blank screen while some process runs. Beginners might wonder if the tool hung or if they did something wrong. **Always provide feedback for ongoing operations.** This can be a spinner animation, a progress bar, or periodic log messages indicating progress [41] [42]. Even a simple text like "Validating workbook... (step 1/3)" that updates is better than silence. Ideally, include a spinner or progress bar to show activity – terminals today can handle dynamic updates to a line, or you can print incremental percentages. For example, when pulling a large Docker image, Docker CLI prints progress bars for each layer download, with percentages and statuses (Downloading, Extracting, Complete) [43]. This granular feedback **entertains the user and sets expectations** for how long they might wait [43] [44]. It's noted that if Docker had no such indicator, many would wrongly assume it froze and might cancel the operation [45]. By seeing progress, users are more likely to remain patient [44].

Figure: Docker's CLI shows detailed progress for each layer of an image being pulled, including download progress bars and extraction status. Each line updates as it reaches completion. Such loading indicators inform and reassure the user that work is being done and how far along it is [43] [44].

Good loading indicators typically have three traits: **1)** They indicate *what* the tool is doing (so the user knows it hasn't crashed) [42]. **2)** They provide a sense of motion or time (e.g. a spinner, progress bar, or step count) to *entertain* or occupy the user while waiting [42]. **3)** They give an idea of *how much is left* – either a percentage, a fraction of steps completed, or at least the fact that it's still making progress [42]. Even if you can't calculate an exact time remaining, showing incremental progress or milestones (like "Validated 10 of 50 worksheets...") is helpful [43] [46]. For unpredictable tasks, a spinner plus changing message (e.g. "Cleaning workbook... Removing unused fields... Optimizing extracts...") can reassure users that things are moving forward [47]. If your CLI performs multiple phases, you might break it down: e.g. *"Analyzing (1/3)"*, then *"Rewriting XML (2/3)"*, then *"Finalizing (3/3)"*. This heuristic of **breaking long tasks into visible sub-tasks** is recommended so that users see progress through each stage [48].

For the workbook cleanup scenario, if a cleanup might take a while (say scanning a large workbook), you could show a progress bar for the validation phase (maybe a percent of workbook scanned or number of issues found) and another for the rewrite phase. Use a library or API to handle spinner/progress updates (e.g. the Python *Rich* library offers high-level APIs for progress bars and spinners, and Node has libraries like *ora* for spinners).

Confirm and summarize actions: After an operation completes, especially one initiated by an interactive prompt, provide a clear success message or summary of changes. New users need closure that "the task is done" and what the outcome was [49]. For example, after cleaning a workbook, the CLI might output something like: *" Cleaned 'SalesReport.twbx': removed 2 unused fields, fixed 3 broken formulas."* This one-line

summary confirms success. If multiple items were processed, a brief report table or list is great (e.g. list each workbook and the number of issues fixed in each). This practice of giving a **reaction for every action** is crucial – if a user runs a command and sees no output, they might be unsure if it executed at all [50]. Even a simple "*Done!*" or "*Cleanup complete.*" in green text can serve as positive feedback. Conversely, if something failed, clearly indicate that (perhaps returning a non-zero exit code as well) and direct the user to logs or next steps (this overlaps with error handling, discussed next).

In short, format your CLI's output to **maximize readability and reassurance**. Use visual aids (colors, alignment, spinners) in a thoughtful way to make information stand out without overwhelming. By doing so, your CLI will communicate effectively with novices and still offer the precision and structure that advanced users expect.

Error Display and Input Validation

Even with good guidance, users will run into errors – maybe a bad input, a missing file, or an internal failure. How your CLI communicates errors can make the difference between a frustrated novice and an empowered one. The goal is to produce **human-readable, actionable error messages** and to validate inputs to prevent errors when possible.

Fail fast with helpful feedback: Input validation (mentioned earlier for interactive prompts) should also occur for command-line arguments. If the user runs `cleanup -f NonExistent.twbx`, it's better to immediately say *"File not found: 'NonExistent.twbx'"* than to proceed and throw a stack trace about an IO exception. A good CLI checks preconditions and prints **clear, concise error messages** explaining the issue [51]. For example, if a user tries to run `npm install` outside of a Node project, npm responds with a straightforward error that it cannot find a *package.json* in the working directory [52]. It also adds context: *"This is related to npm not being able to find a file."* and suggests what might be wrong (are you in the correct folder?) [53].

Make error messages explanatory and constructive: Always describe *what went wrong* in plain language, and if possible *how to fix it* [54]. For instance, instead of a cryptic "NullReferenceException in Module X," say *"Error: Workbook XML is missing a required section <datasources>. The file may be corrupted."* and then suggest: *"Try restoring from a backup or re-saving the workbook."* If the error is due to user input, guide them: *"Unrecognized option --cleen. Did you mean --clean?"* – many CLI parsers (like Python's Click or Go's Cobra) can automatically suggest the closest match for unknown commands or flags [55]. Git does this nicely: typing `git comit` (typo) yields *"git: 'comit' is not a git command. See 'git --help'. Did you mean 'commit'?"* [55]. Implementing such suggestions (via Levenshtein distance on commands) greatly improves UX for typos and is considered a "low-hanging fruit" that many tools unfortunately overlook [56]. Include enough detail for the user to take action – for example, *"Cannot open output directory /path: Permission denied"* is more helpful with an added *"(check your write permissions or choose a different output location)."*

Keep the tone polite and avoid blaming the user. Instead of *"Invalid input"* alone, say *"The date format is invalid. Please use YYYY-MM-DD."* If the tool can fix or ignore an error safely, ask if the user wants to proceed or auto-correct it.

Provide context and next steps: If an error is not directly the user's fault (e.g. a network error, or an internal exception), apologize and point to resources. NPM, for example, will distinguish errors that are

likely not the user's fault (network issues, bugs) and print lines like *"This is most likely not a problem with npm itself"*, followed by hints (e.g. *"check your network connectivity or proxy settings"*) [57]. It even tells the user where to find a detailed log file and asks them to include it when seeking support [58]. You might adopt a similar approach: if something truly unexpected occurs (like an unhandled exception), catch it and output something like *"Oops, something went wrong. Please check the log file at … or rerun with --debug for more info."* This is far better than dumping a stack trace on a novice user's screen, which would likely confuse or scare them. Reserve raw stack traces or debug dumps for a verbose mode or log file. The default error output should be **clean and informative**, not a wall of technical jargon.

Be specific and avoid ambiguity: Tailor error messages to the specific error condition. For example, if the user enters an out-of-range value, say *"5 is not a valid choice; please enter a number 1-4."* or if a required subcommand is missing, say *"No action specified. Use* `cleanup --help` *to see available commands."* Many CLI frameworks will automatically show usage help when a command is incomplete or arguments are wrong; leveraging that can turn an error into a mini-teachable moment (the user sees the correct syntax in the error output). If your CLI has error codes, try to make them meaningful or document them. Some tools output codes like `ERROR 42` which are not useful unless explained; if you must use codes, accompany them with a human message, or provide a reference (e.g. *"Error HE0030: Device not connected (see docs: XCode Errors -> HE0030)"* [59]).

Use consistent formatting for errors: It helps to have a standard like prefixing errors with a label (many tools use `Error:` or `ERROR:` in red text). This consistency lets users spot an error line quickly. Similarly, warnings could be prefixed with `Warning:` and maybe output to stderr as well. Speaking of which, **send error output to STDERR** (standard error stream) and normal output to STDOUT [60]. This separation allows advanced usage like `mycli ... 2>errors.txt` to capture errors. It's a classic UNIX convention: *"All errors go to stderr (2), all other output to stdout (1)"* [61]. Following it means your CLI plays nicely in pipelines and scripts, which is part of being "powerful" for advanced users.

Example – Validation errors in our cleanup tool: Suppose a workbook has inconsistent XML. The validation phase could catch this and output something like:

```
ERROR: Worksheet "Sales Stats" contains an invalid formula reference.
→ Cause: The field "Annual Sales" no longer exists.
→ Fix: Remove or update the formula in the Tableau workbook.
```

This hypothetical error message clearly labels the error, points to the cause, and even hints at a solution. The arrow or indent for the cause/fix is a stylistic choice to structure the information. After listing such errors, the CLI might ask if it should attempt an auto-fix (if that's a feature) or direct the user to fix it manually in Tableau. The key is that the user knows *what* is wrong and *what to do next*, rather than just seeing *"Error code 1005 at line 34"* and being left puzzled.

In summary, treat error messages as part of the UX. They should be written in plain language, anticipate what the user might not understand, and guide them out of the problem. As Atlassian's CLI design principles state: every error message should include a description of what went wrong and suggestions for how to fix it [54]. If you adhere to that rule and validate inputs to prevent common mistakes, even errors can become a moment of learning rather than frustration for a beginner.

Accessibility and Simplicity for New Users

A beginner-friendly CLI must be **accessible** (in terms of usability by people of varying skill levels and needs) and **simple** in its presentation. Simplicity doesn't mean lacking power; it means the first experience is straightforward and not intimidating. Here are several tactics to achieve this:

Use clear, concise language: Write all prompts, help messages, and outputs in plain language, free of unnecessary jargon. For example, instead of "Error in Module 7 of Datasource XML," say "**Error:** A problem was found in the workbook's data source section." Use terminology the user understands (domain terms like "worksheet" or "field" in the Tableau context) and avoid internal implementation terms. Keep sentences short. Atlassian's team found that users often *skim* CLI output, so they recommend instructions or descriptions be no more than 3 sentences at a time [62]. Dense paragraphs in a terminal will overwhelm new users. Break up long explanations into steps or bullet lists where possible (just as we are doing here) [63]. Key information can be emphasized with capitalization or symbols (like "IMPORTANT:" or an arrow ➜) but use such formatting sparingly so that when you do, it actually stands out [63].

Follow familiar patterns and key conventions: Align your CLI with conventions that users may have seen in other tools [21]. This includes things like supporting `--help` and `-h` for help, using `--version` to show version, using `-v` or `--verbose` for verbose output, etc. If your CLI requires authentication or configuration (like AWS's `aws configure`), naming it `configure` or `login` is clearer than inventing a new term. Following widely-used CLI conventions (Heroku's and Microsoft's style guides are explicitly cited as good references [64]) means a novice can transfer knowledge from other commands they've learned. For example, using flags (with `--long-name` and `-s` short versions) is now a standard; even if they've only used `git` or `npm`, a beginner will try `--help` instinctively. By meeting those expectations, you reduce the learning curve.

Design for 80x24 and consider readability: Many terminals default to 80 characters width. Ensure your help text, error messages, and outputs **wrap nicely** within such a width [65]. Heroku's guidelines advise keeping descriptions under 80 chars [65]. This avoids horizontal scrolling or ugly wraps that can confuse the layout (imagine a table that wraps lines incorrectly). Also consider users who might be using assistive technologies like screen readers – purely visual cues (like color alone) might not be conveyed, so structure your text logically (e.g. use punctuation, line breaks, indentation to denote hierarchy). While screen-reader usage for CLIs is less common, keeping output text straightforward benefits everyone.

Ensure the CLI is cross-platform: Many beginners on Windows might use PowerShell or CMD to run the tool. Test that things like colors, Unicode symbols (like the emoji or line-drawing characters for tables) degrade gracefully on these platforms. Use libraries that abstract away differences (for instance, Python's Rich and Node's Chalk handle Windows terminal quirks). Provide fallback text if needed when Unicode is not supported (e.g. replace an emoji with a star or just omit it). The CLI should also handle different keyboard layouts or input methods if you use keys (for example, **don't hardcode a prompt to press "↵ Enter" if on some systems it might be "Return" – saying "press Enter" is generally fine though**). These are minor details, but polished UX considers them.

Interactive accessibility: If your CLI has an interactive interface (like menus), make sure there are clear instructions for how to navigate (e.g. *"Use arrow keys to move, Space to select, Enter to confirm"* as seen in GitHub CLI prompts [9]). Most interactive prompt libraries print these hints by default. Ensure that the text

for options is descriptive. For example, in a menu, instead of just listing file names that might be cryptic, you could list files with additional info (size, date) or index numbers so the user has context on what to choose. Always give an option to cancel (like a **"Cancel"** entry in a menu or a way to select nothing) to avoid trapping the user. Remind users of the escape hatch: in any prompt, they can usually hit `Ctrl+C` to abort. In fact, explicitly mention that in a long-running interactive session – e.g. *"(Press Ctrl+C at any time to quit without making changes)"*. This provides a sense of control, much like a cancel button in a GUI [66].

Confirmation and safety for new users: Novices may be anxious about breaking things. Your CLI can alleviate this by **being safe by default**. We mentioned defaulting confirmations to "no". Similarly, consider a **dry-run mode** or a prompt that shows what will happen and asks for final confirmation before doing an irreversible action. For instance, the cleanup CLI could have a `--dry-run` flag that just simulates the cleanup and reports what it *would* change, without actually saving anything. Interactive mode could integrate this by first showing the summary of proposed changes and then asking *"Apply these changes? (y/N)"*. This way, a new user can confidently say "yes" knowing exactly what will happen. Many package managers do this (e.g. `apt-get upgrade` will list packages to be updated and ask for confirmation).

Sensible defaults and context-awareness: Make smart assumptions to streamline the experience, but always let the user override. For example, if the tool is run without specifying an output location, you might default to the current directory or a `./output` folder – and clearly say so: *"No output folder specified, using current directory."* Similarly, your CLI can detect context; for instance, if the current directory has only one `.twbx` file, you might default to that file rather than asking the user (still printing *"Found one workbook: 'SalesReport.twbx'. Selecting it by default."* so they know what's happening). **Context-aware behavior** can reduce the steps a beginner has to take [67] [68]. NPM is cited for doing this well – it will behave differently depending on context (like detecting a package.json in the folder) to do the right thing [67]. Our cleanup tool could do similar detection (e.g. auto-detect Tableau version or whether a workbook is packaged `.twbx` vs `.twb` and adjust accordingly) to spare the user from needing to specify that info. The overriding principle is reduce what the user has to think about **unless** they explicitly want to take control.

Keep operations and options simple: Especially in "beginner mode" (interactive or default usage), don't present too many choices at once. It's better to have a guided series of 3-4 questions than one giant prompt with 10 checkboxes covering every possible tweak. Break down complex operations. If there are advanced options (e.g. an obscure flag to handle edge-case XML), don't surface them in the main flow for a novice. Those can be left for advanced usage via flags or a separate subcommand. This is analogous to a GUI having basic vs advanced settings – you show the basic ones by default. A CLI can do this by structuring subcommands or modes. For example, your CLI might have a simple `clean` command that applies common fixes, and maybe an `advanced-clean` or extra flags for expert tunings. The **first-run experience** should be optimized for the common use case with minimal flags.

Consistency and learnability: Ensure that once a user learns one command, others feel familiar. Use consistent terminology across commands. If you have an `--all` flag in one place to mean "process all items", use `--all` in other commands for similar effect (don't suddenly call it `--everything` elsewhere). Keep the order of prompts consistent: e.g. always ask "are you sure?" at the end, always ask for the file selection at the beginning. This consistency builds a mental model for the user. The Heroku CLI's mission statement literally says the CLI is *"for humans before machines"* and that *"input and output should be consistent across commands to allow the user to easily learn how to interact with new commands"* [69]. Embrace that philosophy. If your tool has multiple subcommands (like `validate`, `cleanup`, `export`), ensure they share options where appropriate (if all need a workbook file input, use the same flag name across

them). This reduces cognitive load on the user – they don't feel like each command is a whole new interface to learn.

Accessibility for disabilities: Though not always at the forefront for CLI, it's worth noting a few things: - If targeting visually impaired users, ensure that important output is textual (screen readers can read text but not interpret color or ascii art reliably). So, for instance, a progress bar made of `#####.....` might not be very screen-reader friendly, but if accompanied by a percentage "(50%)", it gives a number the reader can speak. Some CLI frameworks have started considering this (e.g. GitHub CLI added features to be more screen-reader accessible [70]). - High contrast themes: some users might use terminal themes that make certain colors hard to see (e.g. light text on white background). Test your color choices on both dark and light backgrounds, or provide a way to choose a different color theme if needed. At minimum, ensure that if color is off, the messages are still understandable.

To conclude this section, **simplicity is about not overwhelming the user**. Strip out unnecessary info, use sensible defaults, guide them with clear text, and ensure they always feel in control (able to cancel or get help). By making the CLI's behavior predictable and its language friendly, you create an environment where a new user can gradually build confidence. They can start with interactive guided usage and, as they get comfortable, transition to faster, flag-driven workflows – all within the same tool.

CLI Onboarding and Guided Usage Patterns

Onboarding refers to the user's first-run experience and how you help them ramp up. For a beginner-friendly CLI, consider implementing **guided modes and first-time tips** that shorten the learning curve.

Provide helpful guidance on first run (or no arguments): If the user runs your tool with no subcommand or with a `--help` flag, use that opportunity to show a **helpful welcome message** or quick start. Instead of dumping a dry usage syntax only, you can display a brief description of what the tool does and list the most common commands with examples. For instance:

```
$ tableau-cleanup-agent

Welcome to the Tableau Workbook Cleanup CLI! This tool helps validate and
optimize .twb/.twbx workbooks.

Common commands:
  cleanup   Interactive cleanup wizard for one or more workbooks.
  validate  Validate a workbook and report issues (no changes made).
  help      Show detailed help for a command.

Try "tableau-cleanup-agent cleanup" to get started with an interactive cleanup.
```

This kind of **instructive onboarding message** nudges the user toward the next step [71]. Lucas F. Costa suggests highlighting the command a new user is most likely to need first, rather than overwhelming them with every option [72]. For example, if "cleanup" is the primary function, you might prominently suggest that. Some CLIs even detect first-time use (perhaps by storing a config file on first run) and display a special

one-time welcome or tutorial message. You can implement that if it makes sense – for instance, first time, print "Tip: run `setup` to configure default settings" or similar, then don't show that every time.

Built-in help and examples: Ensure each command has a clear `--help` output with usage examples. Beginners often copy-paste examples. The help text should list subcommands and flags with brief, **simple descriptions** (one line each) [73]. If possible, include an example invocation for complex commands in the help output (e.g. under `cleanup --help`, show "Example: `tableau-cleanup-agent cleanup workbook.twbx --backup --compress` ... does XYZ"). Atlassian's CLI principles emphasize that `--help` is an essential part of CLI UX for both new and experienced users [74]. New users discover capabilities through it, and experienced users use it as reference. So, make sure `cli --help` and `cli <command> --help` are comprehensive but well-organized (use sections, indentation, or formatting to make it scannable).

Interactive "wizard" or guided mode: In addition to interactive prompts that occur when a command is run without arguments (like `gh pr create` we discussed), some CLIs offer a dedicated **guided mode**. For example, Heroku CLI has a global `--prompt` flag that turns any command into an interactive Q&A session [75]. A user unsure of the syntax can do `heroku config:set --prompt` and it will interactively ask for the config key and value, rather than requiring them to type it all in one go [76]. This kind of feature is extremely welcoming to newcomers, as it allows them to gradually learn the command. You might consider a similar feature: e.g. `tableau-cleanup-agent --prompt` could launch a general interactive flow, or a subcommand like `tableau-cleanup-agent wizard` that walks through the entire cleanup process. The benefits, as Heroku notes, are that prompting helps users *"never forget optional arguments,"* learn required inputs, see descriptions for each input as they go, and get validation before execution [77] [78]. Essentially, it's like an interactive tutorial embedded in the CLI.

If you implement such a wizard, design it to be forgiving. Allow going back a step if that's feasible, or canceling at any time. Provide defaults and example inputs at each question to guide the user. For instance, *"Enter the path of the workbook [default: current directory]:"*. If they just hit Enter, you use the default. This reduces friction in the learning phase.

First-run configuration: Many powerful CLIs require some initial setup – e.g. authentication tokens, config files, etc. If your tool needs configuration (perhaps not much in this Tableau cleaner, except maybe default directories or settings), consider a guided setup command. For example, AWS CLI has `aws configure` which interactively asks for access key, secret, region and sets up the config file. GitHub CLI's `gh auth login` is another good example: it's an interactive authentication flow by default (it even opens a web browser for OAuth and then returns to the terminal) [79]. The user just types `gh auth login` and follows prompts like "What account do you want to log into? Github.com or GitHub Enterprise?," "How would you like to authenticate? (Browser, Paste an auth token, etc.)". This is much easier for a beginner than having to manually generate tokens or edit config files. For our tool, if there were a need to configure, say, a default backup folder or turn on/off certain rules, an interactive `configure` command or a first-run prompt ("Would you like to enable auto-backup of workbooks? Y/n") could help users set it up correctly from the start.

Onboarding documentation and resources: In the CLI output or help, you can also point users to further resources (web links to documentation or examples). For instance, after a successful run or when the user runs `--help`, you might print "☞ More examples and documentation: https://github.com/yourrepo/

cleanup-agent/docs". Many users appreciate having a URL they can visit for more detailed guides or troubleshooting.

Suggest next steps and related commands: A newbie might not know what to do after finishing one task. Atlassian recommends to *"suggest the next best step"* at the end of a command's output [80]. For example, after a `validate` command that just reports issues, you could add *"Tip: run* `cleanup` *to automatically fix these issues."* Or after a `cleanup`, maybe *"You can open the cleaned workbook in Tableau to review the changes."* In a multi-step interactive session, after completing, you could ask *"Do you want to clean another workbook? (y/N)"* – this keeps the user engaged and aware of what they can do next without having to consult the docs again. This kind of guidance is akin to a UI wizard saying "You're all set! Would you like to [Run again] or [Exit]."

Fail-safe and undo: This might be a bit beyond classic CLI UX, but for a cleanup tool specifically, consider user peace-of-mind features like automatically creating a backup of the workbook before modifying it (or offering that option in a prompt). Then if something goes wrong or if the user is unhappy with the changes, they have a fallback. If you do create backups or logs, tell the user where they are (e.g. *"A backup of the original workbook was saved as SalesReport_backup.twbx"*). Knowing they can rollback makes beginners more comfortable saying "yes" to that final confirmation.

Testing the onboarding with real beginners: A best practice is to do some usability testing – have a colleague or friend who's not familiar with the tool try to use it with no prior context, and see where they get confused. Their experience can highlight if your prompts are unclear or if the help is insufficient. Then iterate. Many open-source CLIs gather feedback from first-time users to polish these aspects (for example, GitHub CLI's team made accessibility improvements after users noted issues [70]).

In essence, **onboarding is about reducing "time to value"** – the time it takes a new user to get a useful result [81]. GUIs hand-hold users; CLI tools often historically threw a manual at the user. But we can do better by integrating guided experiences into the CLI itself [82]. By using menus, wizards, informative help text, and thoughtful first-run behavior, your CLI can be welcoming from the very first use.

In conclusion, designing a beginner-friendly, interactive CLI involves balancing guidance with power. We've seen that using interactive prompts (menus, confirmations, etc.) can lead users through tasks in a way that feels safe and intuitive – much like a GUI, the CLI asks and the user answers, rather than dumping them in a sea of options [83] [3]. At the same time, preserving direct commands and scriptability ensures the tool remains **powerful** for advanced users or automation scripts. A well-designed CLI like this Tableau Workbook Cleanup Agent might use an interactive wizard for most users, but under the hood it's the same commands that an expert could run with flags in a single line.

By applying consistent menu styles, clear colored output and progress feedback, robust error handling with guidance, and onboarding best practices, you can create a CLI that **"delights the user"** (as Heroku's guide puts it) [69]. The user should feel in control and informed at each step – from selecting items in a menu, to watching a spinner tick by, to reading a success message at the end. Borrowing ideas from popular CLIs (`npm init`, `gh`, `docker pull`, `heroku --prompt`, etc.) and using libraries like *Inquirer.js* (for prompts), *Click/Cobra* (for structuring commands and validations), or *Rich* (for beautiful output) can accelerate this development, since these tools embody many of the best practices discussed. Ultimately, the goal is a CLI that a newcomer can use without frustration or fear, yet one that an expert can utilize

efficiently – achieving both approachability and power. By following the guidelines and examples above, you'll be well on your way to designing a CLI interface that **feels interactive, friendly, and robust** all at once. [69] [36]

Sources:

- Lucas F. Costa – *UX patterns for CLI tools* [84] [22] [30] [42]
- Ubuntu CLI Guidelines – *Interactive Prompts* [4] [7]
- Atlassian – *10 Principles for Delightful CLIs* [54] [80] [66]
- Heroku Dev Center – *CLI Style Guide* [69] [23] [32] [40]
- GitHub CLI (gh) – *Design & Usage Examples* [5] [6] [9]
- Heroku Dev Center – *Heroku CLI Interactive Prompt* [85] [76]
- Evil Martians – *CLI UX: Progress Displays* [43] [44]
- Various CLI tool docs (npm, Docker, etc.) for illustrative behavior [53] [57].

---

[1] [2] [3] [13] [14] [17] [18] [22] [24] [25] [30] [31] [42] [43] [44] [45] [46] [47] [51] [52] [53] [55] [56] [57] [58] [59] [61] [67] [68] [71] [72] [81] [82] [83] [84] UX patterns for CLI tools
https://www.lucasfcosta.com/blog/ux-patterns-cli-tools

[4] [7] [8] [11] [12] [15] [16] Interactive prompts - CLI Guidelines - Ubuntu Community Hub
https://discourse.ubuntu.com/t/interactive-prompts/18881

[5] [6] [9] GitHub: top commands in gh, the official GitHub CLI - Adam Johnson
https://adamj.eu/tech/2025/11/24/github-top-gh-cli-commands/

[10] A step-by-step guide to using Inquirer.js for creating a CLI app in Node.js
https://geshan.com.np/blog/2023/03/inquirer-js/

[19] [20] [23] [26] [27] [28] [29] [32] [33] [34] [35] [36] [37] [38] [39] [40] [60] [65] [69] CLI Style Guide | Heroku Dev Center
https://devcenter.heroku.com/articles/cli-style-guide

[21] [41] [48] [49] [50] [54] [62] [63] [64] [66] [73] [74] [80] 10 design principles for delightful CLIs - Work Life by Atlassian
https://www.atlassian.com/blog/it-teams/10-design-principles-for-delightful-clis

[70] Building a more accessible GitHub CLI
https://github.blog/engineering/user-experience/building-a-more-accessible-github-cli/

[75] [76] [77] [78] [85] Heroku CLI Interactive Prompt | Heroku Dev Center
https://devcenter.heroku.com/articles/cli-prompt

[79] gh auth login - GitHub CLI
https://cli.github.com/manual/gh_auth_login