Agent-Friendly CLI Checklist

A practical checklist for designing command-line interfaces that coding agents can discover, run, and verify.

Who this is for

Use this when your product should be usable by coding agents such as Claude Code, Codex, Cursor/IDE-style agent environments, Replit-style agents, or any agent with:

- bash/shell access
- network access
- filesystem access
- environment variables/config
- ability to run commands and inspect outputs

This checklist is **not primarily for chat-only ChatGPT or Claude.ai** unless those environments have tool execution access.

The core idea

> A good CLI is not just a developer convenience. It is an executable product interface for agents.

Agents need CLIs that are:

- discoverable
- predictable
- noninteractive when needed
- machine-readable
- safe around side effects
- easy to authenticate
- easy to verify
- easy to install
- easy to check for updates

The checklist

1. Discoverability

- [ ] `tool --help` exists.
- [ ] `tool <resource> --help` exists.
- [ ] Help text includes real examples.
- [ ] Docs show the top 5 workflows as copy-paste commands.

2. Predictable command grammar

- [ ] Commands use a consistent grammar, e.g. `tool project list/get/create/update`.
- [ ] Resource names are consistent across commands.
- [ ] Flag names are consistent: do not use `--project`, `--project-id`, and `--pid` inconsistently.
- [ ] Similar commands behave similarly.
- [ ] No surprise aliases with different semantics.

3. Noninteractive automation path

- [ ] Core workflows can run without a TTY.
- [ ] Interactive prompts have automation alternatives.
- [ ] Confirmation flags are explicit: `--confirm`, `--yes`, or equivalent.
- [ ] The CLI never hangs waiting for input in automation mode.
- [ ] Browser-based auth is not the only option for agent workflows.

4. Machine-readable output

- [ ] `--json` is available for all inspect/action commands.
- [ ] JSON schemas are stable across versions.
- [ ] stdout contains data.
- [ ] stderr contains errors/progress/logs.
- [ ] Human formatting does not break machine parsing.
- [ ] Color can be disabled with `--no-color` and/or the `NO_COLOR` environment variable.
- [ ] Progress indicators, spinners, and progress bars degrade gracefully when stdout is not a TTY.

5. Actionable errors and exit codes

- [ ] Exit codes are meaningful and documented: `0` means success; nonzero means failure or a documented alternate outcome.
- [ ] The CLI avoids “error text with exit 0.”
- [ ] Error messages say what happened.
- [ ] Error messages explain how to fix it.
- [ ] Auth errors distinguish missing token, expired token, and insufficient permissions.
- [ ] Partial failures are explicit and machine-readable.
- [ ] Important exit-code meanings are documented, especially validation failures, auth failures, not-found cases, conflicts, partial failures, and rate limits.

6. Safe side effects

- [ ] Destructive commands require explicit confirmation.
- [ ] `--dry-run` or `plan` is available before side effects.
- [ ] Output includes the IDs of changed resources.
- [ ] Operations are idempotent or safe to retry where possible.
- [ ] Production-impacting commands are clearly marked.

7. Authentication and environment

- [ ] Env var auth is supported, e.g. `ACME_API_KEY`.
- [ ] Config-file auth is documented.
- [ ] `auth login` exists for humans.
- [ ] `auth status --json` exists for agents.
- [ ] Missing/expired credentials fail clearly.

8. Version and update behavior

- [ ] `tool --version` prints the installed version.
- [ ] The CLI can check whether a newer version exists.
- [ ] Update checks do not block automation or require a TTY.
- [ ] If a newer version exists, the message includes the exact update command.
- [ ] Update notices are machine-readable with `--json` where possible.
- [ ] The CLI warns about stale versions without hiding command output or changing exit semantics

for successful commands.

- [ ] Docs explain package-manager-specific update paths: npm/npx, Homebrew, pipx, Docker, GitHub Releases, etc.

Example JSON:

```
{
  "installed_version": "1.4.2",
  "latest_version": "1.5.0",
  "update_available": true,
  "update_command": "npm update -g acme-cli"
}
```

9. Verification commands

- [ ] Every action has a way to verify result.
- [ ] `get`, `status`, `list`, `logs`, or `history` commands exist where relevant.
- [ ] Verification commands support `--json`.
- [ ] Commands return resource IDs that can be checked later.
- [ ] Docs teach “act, then verify” workflows.

10. Composability

- [ ] Commands work in scripts.
- [ ] Commands that consume input support stdin by default when data is piped, e.g. `cat input.json | tool import`, `tool import --stdin`, or `tool apply -f -`.
- [ ] The CLI explains the Unix `-` convention where relevant: in file arguments, `-` usually means “read this file/input from stdin,” as in `tool apply -f -`.
- [ ] Commands do **not** hang waiting for stdin in automation: if stdin is a TTY, prompt intentionally or fail with a clear message; if stdin is piped/non-TTY, read it.
- [ ] Commands support pipes/files where useful.
- [ ] stdout is reserved for composable data/result output.
- [ ] stderr is reserved for errors, warnings, progress, and diagnostics.
- [ ] Output is deterministic enough for tests.
- [ ] Rate limits/retries are documented.

- [ ] Commands avoid hidden global state where possible.

Short stdin example: `cat input.json | tool import` pipes data into the command; `tool apply -f -` uses the Unix `-` convention to mean “read this file/input from stdin.”

11. Distribution, installation, and updates

- [ ] One-line install exists.
- [ ] `npx your-cli@latest ...` or equivalent latest-version zero-install path exists if appropriate.
- [ ] Version pinning is possible.
- [ ] Update command is documented for every distribution path.
- [ ] Install instructions work in a clean environment.
- [ ] Package distribution is documented: npm, Homebrew, pipx, Docker, GitHub Releases, etc.

12. Agent onboarding docs

- [ ] Quickstart for agents.
- [ ] Common workflows.
- [ ] JSON output examples.
- [ ] Troubleshooting guide.
- [ ] Side-effect and destructive-command safeguards.
- [ ] Optional Skill that packages the CLI docs and workflows.

How to test

Give a coding agent this prompt:

```
Use this CLI to complete [workflow]. Inspect current state, perform the
action, verify the result, and summarize exactly what changed. Do not assume
success; use status/get/logs commands to prove it. <doc link>
```

Watch for:

- Can the agent discover the right command?

- Does it understand the flags?
- Can it authenticate?
- Does it parse output correctly?
- Does it avoid unsafe side effects?
- Does it verify success?
- Does it recover from errors?
- Does it notice when the CLI is stale and know how to update?

If the agent gets stuck, the problem may not be the model. It may be your CLI UX.

© 2026, Level 250, Inc. | Emmanuel Paraskakis