ChatGPT

Scope

The analysis covers two repositories — `AcidicSoil/prompts` and `AcidicSoil/claude-task-master` — examining their architectures, interfaces, and how a cross-platform Node.js/TypeScript CLI might interoperate over the Model-Context Protocol (MCP) via stdio and JSON-schema I/O. We identify existing commands, input/output schemas, configuration formats, dependency graphs, and inter-process communication (IPC) patterns in each repo. We also incorporate best practices from comparable open-source CLIs that expose MCP tools (e.g. search terms: "MCP stdio CLI," "commander/oclif/yargs MCP," "tool orchestration," "plan-execute loop") and references on multi-tool coordination and session state (LangGraph/LangChain MCP bridges, Claude Desktop MCP integration, Smithery, *mcpdoc*). Configuration discovery mechanisms (like `mcp.config.json` or `.mcp.json`, environment variables), cross-repo plugin loading, command design (subcommands, flags, piping, exit codes), logging/telemetry, concurrency, retries, and idempotency are investigated. We also note licensing terms and contributor guidelines for each project. Below is a findings brief organized by the requested points.

(1) Command Taxonomy and MVP Commands for `prompts` CLI

Proposed CLI: `prompts` – a Node.js CLI tool for managing and using prompt "playbooks." This repository currently contains a collection of markdown prompt files and metadata (e.g. YAML front matter in each prompt file, and a generated `catalog.json` index [1] [2]), but no standalone CLI binary yet. The CLI would expose commands to list, view, render, and execute these prompts in an MCP-compatible manner. Minimal viable commands might include:

- `prompts **list**` – List all available prompt playbooks (perhaps grouped by category or tags) by reading the catalog or prompt directory.
- `prompts **show** <name>` – Display the full text of a named prompt (after resolving any template variables) to stdout for review or piping.
- `prompts **run** <name> [--input <file> | --var key=val ...]` – Render a prompt with provided input variables and execute it through an MCP client, outputting the JSON result. This could effectively perform a one-shot prompt execution and return the assistant's result in JSON (following MCP I/O schema).
- `prompts **plan** [<session.yaml>]` – Generate a task **plan** (possibly a directed graph or list of steps) based on prompt metadata or a given session description, and output the plan as JSON. (For example, use the prompt's front matter links like `previous` / `next` to build a sequence [3].) An `--auto` flag could synthesize a plan from the current context or default workflow.
- `prompts **mcp-server**` (or `mcp start`) – Launch a local MCP stdio server that exposes each prompt as a tool. This would read prompt definitions and serve them over MCP (via JSON-RPC over stdio) so that external clients (like `claude-task-master` or IDE plugins) can invoke prompt tools programmatically [4]. The server would handle JSON-schema I/O for each prompt (e.g. define input parameters and output structure).
- `prompts **call** <tool> --json <payload.json>` – Send a JSON request (conforming to MCP's JSON-RPC format) to either a local `prompts` MCP server or a target MCP endpoint, calling a

specified tool (prompt) and returning the result. This would be useful for piping data between processes or testing tool calls from the CLI.

- `prompts **export** [--format llms.txt]` – Export the prompt catalog or metadata in a specified format, such as llms.txt (if needed for integration with other systems), or validate that the prompts meet a certain format (the repo already can build `catalog.json` and has llms.txt references).

Grouping and Usage: These commands can be structured under logical groupings (like how the prompts are grouped by development phase in the README [5]). For example, `prompts plan` and `prompts run` relate to execution, whereas `prompts list` and `prompts show` are informational. If the CLI grows, subcommands (e.g. `prompts mcp ...`) or a plugin system (as provided by frameworks like Oclif) can organize them.

I/O Schema: Each command that executes a prompt should adhere to MCP's JSON schema conventions for inputs and outputs. For instance, `prompts run` would produce output as a JSON object with the assistant's reply, potentially wrapped in an MCP tool result envelope (including fields like `stdout`, `stderr`, `exitCode` if mimicking a process, or more semantically, a `result` field with the content). The **Model-Context Protocol (MCP)** defines standardized JSON-RPC messages for tool invocation and completion [6], which our CLI should respect for interoperability.

Dependency Graph: Internally, the `prompts` CLI can use the existing `catalog.json` (which is regenerated by the repo's build scripts [4]) to map prompt names to files and metadata. The dependency graph among prompts (through `previous`/`next` relations [3]) can help `plan` command to chain tools. The CLI's own dependencies would include a JSON Schema validator (to enforce that input parameters match the prompt's expected format, if defined) and possibly the MCP core library or JSON-RPC utilities to format I/O. Given the repository is primarily content, adding a TypeScript CLI would involve introducing packages like `commander` (for command parsing) [7] [8] or similar, and perhaps using an existing MCP client/server implementation.

Table: Example Commands for `prompts` CLI

Command	Description
`prompts list`	List all available prompts (by name or category).
`prompts show <name>`	Display the content of a prompt file (resolved markdown).
`prompts run <name>`	Render and execute a prompt with given inputs, output result as JSON.
`prompts plan [<spec>]`	Generate a plan (task graph or sequence) from prompt metadata or a given spec.
`prompts mcp-server`	Start an MCP stdio server exposing prompts as tools (for integration).
`prompts call <tool> --json <file>`	Invoke a prompt tool via MCP, sending a JSON request and outputting the response.

Command	Description
`prompts export [--format llms.txt]`	Export or validate prompt catalog in specified format (e.g., llms.txt).

These constitute a minimal useful set of capabilities to navigate and utilize the prompt repository in both standalone and integrated fashions.

(2) Compatibility Layer Spec (`prompts` ↔ `claude-task-master`)

For `prompts` and `claude-task-master` to interoperate, they should communicate via MCP stdio streams or a common IPC mechanism using JSON. **Claude Task Master** (from `AcidicSoil/claude-task-master`, originally by eyaltoledano) is designed as an orchestrator that can run tasks using tools over the Model-Context Protocol. In practice, this means `claude-task-master` can act as an MCP client, calling out to tool providers, or even as an MCP server for certain built-in tools. Our compatibility layer must allow the `prompts` CLI to either **be called by** Task Master (as an MCP server exposing new tools) or **call into** Task Master (as a client invoking Task Master's planning/execution commands).

Key elements of the spec:

- **Transport Protocol:** Use MCP's **stdio transport**, which is essentially JSON-RPC 2.0 over standard input/output [6]. Both processes communicate by sending JSON messages delimited by newline (or some framing). This approach avoids network complexity and is cross-platform for local interoperability.
- **Discovery & Configuration:** To connect the two systems, follow conventions used by Task Master:
- Look for a configuration file like `.mcp.json` in the working directory or project root. Task Master uses such a file to discover available MCP servers and tools. For example, `.mcp.json` might list local tool providers (with commands to launch them, port numbers, etc.) [9]. The `prompts` CLI can read and register itself if needed or at least honor any config (e.g., environment variables like `MCP_SERVERS` list).
- Provide environment variable overrides. If `claude-task-master` expects environment configs (for example, to specify the path or port of an external tool provider), the `prompts` CLI should either read those or allow configuring via its own flags that align with Task Master's conventions.
- **JSON Schemas for I/O:** Each prompt tool should have a defined input and output schema. For instance, a prompt might accept a structured input (like a JSON object with fields corresponding to variables in the prompt) and produce a structured output (the assistant's answer, plus possibly metadata). Using JSON-Schema definitions for these ensures that `claude-task-master` can validate tool inputs/outputs. In an MCP handshake, tools can advertise their schema. The compatibility layer could have `prompts mcp-server` advertise each prompt's schema (perhaps gleaned from front matter or a standard convention in the prompt file). This aligns with MCP's design of **introspection** where tools can describe their interface [6].
- **Method Namespacing:** Define how Task Master identifies the prompt tools. For example, if Task Master tries to call a tool named `"render_prompt"`, the `prompts` MCP server should know which prompt to run. We might give each prompt a tool name (perhaps the filename or an alias). A

simple approach is to use the prompt's filename or an explicit identifier in its metadata as the MCP method name. The CLI can map method -> prompt file and execute accordingly.

- **Interoperability Direction:** Two possible integrations:
- **Task Master as Orchestrator, Prompts as Tools:** In this mode, `prompts mcp-server` runs and Task Master connects to it. Task Master's plan executor can call prompt tools as needed (for example, if a plan step is to use a specific prompt for a subtask). This requires minimal changes in Task Master aside from pointing it to the new MCP server (via config). It treats prompts just like any other tool endpoint.
- **Prompts CLI as Caller:** Alternatively, the `prompts run` or `prompts plan` commands could internally invoke Task Master's capabilities. For example, `prompts plan --auto` might call out to `claude-task-master` to generate a sophisticated task graph (since Task Master has planning logic) and then use that plan. This would mean `prompts` acts as an MCP client to Task Master's server. In this case, Task Master would likely be running as a background service (possibly started via its own CLI or an API). The compatibility spec should define how `prompts` discovers a running Task Master (e.g., via `.mcp.json` entry or known port/socket).
- **Data Formats:** Ensure that the **JSON messages** exchanged conform to the MCP standard. A typical JSON-RPC request from `claude-task-master` to a prompt tool might look like:

```
{
  "jsonrpc": "2.0",
  "id": 42,
  "method": "prompts/plan_session",
  "params": { "sessionSpec": "...", "options": { ... } }
}
```

The `prompts` server would respond with a result or error object. Using namespacing like `"prompts/..."` can avoid collision with Task Master's own methods. This example shows how a *plan_session* tool might be called. Similarly, a `render_prompt` method could run a specific prompt. - **Error Handling and Exit Codes:** The compatibility layer should define how errors are propagated. If a prompt tool fails (e.g., required variable missing), it should return a JSON-RPC error object. On the CLI side (if running a single command), it might also exit with a non-zero status code to signal failure for scripting environments. - **Performance and Lifecycle:** Spawning a new process for each tool call (starting `prompts` CLI each time) could be expensive. Using a persistent `prompts mcp-server` process that stays running and handles multiple requests (tools) is more efficient for continuous orchestration. We should ensure the CLI can run in a long-lived server mode. Task Master would then keep a connection (stdio or perhaps upgrade to a socket) open to send multiple tool calls. If using stdio, this means keeping the process alive and reading/writing on the pipes. - **Example Integration Scenario:** Suppose `claude-task-master` wants to use a prompt from `prompts` as part of its execution plan (say a prompt to generate a PRD or do a code review). We configure `.mcp.json` to include an entry like:

```
{
  "servers": [
    {
      "name": "PromptsTools",
      "command": "prompts mcp-server",
```

```
      "autoStart": true,
      "port": 0  /* 0 implies stdio or auto-assigned port if networked */
    }
  ]
}
```

Task Master reads this, launches `prompts mcp-server`, and then calls e.g. `method: "PromptsTools/execute_prompt"` (depending on naming) with appropriate params. The prompts server executes the prompt and returns the result, which Task Master incorporates into the task workflow. This **compatibility layer spec** ensures both sides agree on naming, schema, and transport ahead of time.

(3) Sample End-to-End Flows

To illustrate the interplay, here are a couple of end-to-end usage scenarios bridging planning and execution between the `prompts` CLI and `claude-task-master`:

Flow A: Plan with Prompts, Execute with Task Master (Two-step)
1. **Plan generation:** A developer wants to create a plan for a coding task. They use the prompts CLI to leverage its prompt knowledge for planning:

```
prompts plan --auto > plan.json
```

This could produce a JSON file (plan.json) describing a sequence of steps or a graph of tasks derived from prompt templates (perhaps using a prompt that encapsulates project planning logic). The plan might include steps that require AI assistance or code generation. 2. **Execution:** The developer then hands off this plan to Claude Task Master for execution:

```
claude-task-master run --plan plan.json --out results.json
```

Here, `claude-task-master` reads the plan and executes each step. When it encounters a step that calls for a specific prompt (say "Use prompt X to do Y"), it will call the corresponding tool via MCP. As long as `prompts mcp-server` is running and registered, Task Master will invoke it. The outputs of each step (including any text generated by prompts via Claude API) are collected and saved to `results.json`.
In this flow, **prompts CLI is used for planning** and **Task Master for execution**.

Flow B: Interactive Prompt Execution with Piping (One-step orchestration)
This scenario uses shell piping to chain the tools more interactively:

```
prompts show define_prd_template | \
prompts call render_prompt --json - | \
claude-task-master ingest --stdin
```

- `prompts show define_prd_template` outputs the raw content of a prompt (for example, a prompt template to define a Product Requirements Document). This content could be piped as context or input. - `prompts call render_prompt --json -` reads a JSON from stdin (here the JSON might contain a user query or variables for the prompt) and sends it to the `render_prompt` tool (hosted by a running prompts MCP server). The `--json -` indicates it should take JSON input from the pipeline. The command returns the prompt's executed result (the filled template or AI-generated content). - `claude-task-master ingest --stdin` (hypothetical command) could take the resulting content and ingest it into the Task Master session or knowledge base. Alternatively, one could pipe directly into another tool or a file.

In this pipeline, all tools speak JSON via stdout/stdin, maintaining the MCP contract. This demonstrates how one can use the `prompts` CLI tools in a shell script alongside Task Master without manual file juggling, leveraging UNIX pipelines for tool composition.

Flow C: `prompts` **CLI invoking Task Master for Complex Workflows**
If the `prompts` CLI wants to utilize Task Master under the hood (e.g., to benefit from its multi-step orchestration), it might offer a higher-level command:

```
prompts orchestrate <prompt-name> --goal "<objective>" --use-taskmaster
```

This command could internally call Task Master's API to generate a detailed task list for accomplishing `<objective>` (using the specified prompt as a starting point or constraint), then execute it. The CLI would manage launching Task Master (if not already running) and streaming results back to the user. While this flow is more speculative, it shows the flexibility once both systems speak MCP.

Each of these flows relies on a **shared MCP channel** and agreed-upon schemas. They highlight that planning (deciding *what* to do) can be separated from execution (actually *doing* it with AI and code), and the two repos can cooperate by specializing in each.

(4) Security and Sandboxing Notes

When building a system that executes code or tools (especially via AI instructions), security is a vital concern:

- **Local Execution vs Sandboxing:** Both the `prompts` CLI and `claude-task-master` should treat each other's calls as potentially unsafe if not fully trusted. Running a prompt might involve invoking external tools or producing code to execute. We should ensure any actual code execution (if Task Master compiles/runs code or if prompts triggers scripts) is sandboxed. For instance, if `claude-task-master` has a step to run a shell command, it should use a restricted environment. In the context of MCP, tools might be marked as safe or dangerous. The CLI can include a flag like `--sandbox` to run tools in a confined manner (e.g., using Node's `vm` module, Docker containers, or a separate process with reduced privileges).
- **IPC Security:** Using stdio for IPC between `prompts` and `task-master` is secure in that it's a local channel not exposed to the network. However, if an MCP server is exposed via network (for example, some MCP servers use WebSockets or TCP [6]), care must be taken to authenticate clients or restrict

access (especially since tools can potentially execute arbitrary commands). In our use-case, we assume local machine integration, so the threat is primarily that a malicious prompt or task could harm the local system. Keep the stdio channel strictly between the two processes; do not open random ports without authentication.

- **Validation of Inputs:** Since Task Master will feed unstructured data (often from AI) into our `prompts` tools, and vice versa, each side should **validate JSON inputs against the schema** before executing actions. This prevents accidental mis-usage or exploitation through malformed data. The `prompts` CLI can use JSON Schema definitions to check input parameters for each prompt tool, rejecting anything unexpected or dangerous. Likewise, Task Master should validate the outputs it receives from prompts (which are AI-generated) before using them in subsequent steps.
- **Limiting Side Effects:** The `prompts` CLI should avoid uncontrolled side effects. For example, a prompt that generates a file or modifies state should be treated carefully. Possibly run such actions in a temp directory or require explicit user consent. Similarly, Task Master, when orchestrating tasks, should have clear boundaries on what it can do on the host system (perhaps requiring user review for destructive actions).
- **Auditing and Logging:** Enable verbose logging for the interactions. Both tools could log MCP requests and responses (excluding sensitive data) to allow debugging and auditing of what actions were taken. If something goes wrong (or a security incident occurs), logs can help trace whether it was a prompt tool or the orchestrator that introduced an issue.
- **API Keys and Secrets:** If either tool needs API keys (for example, Task Master might call Claude API or `prompts` might call OpenAI/Anthropic to actually get AI completions), never hard-code these. Use environment variables or config files, and ensure those are not exposed in logs. Possibly integrate with a secrets manager (the `prompts` repository has a `/secrets-manager-setup` prompt [10] for example).
- **Commons Clause Implication:** As an aside, the Commons Clause on Task Master's license (see Licensing below) prevents selling the software as a service [11]. While not a technical security issue, it does mean if one integrates these tools into a product, one must not violate those terms. This might influence deployment (e.g., avoiding cloud-hosted public versions of Task Master without permission).

In summary, favor a defensive programming approach: each side treats the other's data as untrusted, validate thoroughly, and limit what each can do without user oversight. Running everything locally under user control mitigates many network-related risks, but one should still guard against logic misuse or accidental damage (e.g., overwriting files, infinite loops, etc.).

(5) Phased Roadmap (with Implementation References & Snippets)

Implementing the cross-platform CLI and integration can be tackled in stages:

Phase 0 – Repository Preparation and Scaffolding
Goal: Set up the basic project structure for the `prompts` CLI and ensure both repos are ready for extension.
- **Project Initialization:** Add necessary Node.js project files to `AcidicSoil/prompts` if not present. Ensure a `package.json` exists and TypeScript is set up. (The prompts repo already has `package.json` and some scripts [4], likely for validation tooling, so building on that is feasible.) If not already, install a CLI framework like **Commander** [7] or **Yargs** for argument parsing. Commander is straightforward for a quick CLI and widely used [7] [8]. For example, to define a simple `prompts` command with Commander:

```
#!/usr/bin/env node
import { Command } from 'commander';
import { version } from './package.json';
const program = new Command();
program.name('prompts').version(version);
program
  .command('list')
  .description('List all available prompts')
  .action(() => {/* list implementation */});
program.parse(process.argv);
```

This snippet sets up a basic CLI with a `list` command. Similar blocks can be added for `show`, `run`, etc. - **Link Repos for Local Dev:** Check out both repos locally and configure a dev environment where `claude-task-master` can be run (it's a Node project too). Ensure you can make changes and test the integration in a sandbox. If `claude-task-master` has a test suite, run it to ensure baseline behavior. - **Dependency Management:** Add dependencies for MCP communication. If a package exists for MCP (for example, an NPM package that can encode/decode MCP JSON-RPC messages), include it. Otherwise, plan to use a lightweight JSON-RPC encoder (could be as simple as `JSON.stringify` and appending newline, since MCP stdio is line-delimited JSON [6]). Also add dev dependencies like TypeScript types for Node, etc.

Phase 1 – Basic CLI Features for `prompts`
Goal: Implement the core CLI commands (from part 1 above) without full MCP integration yet, to handle local usage.
- **Listing and Showing:** Implement `list` (reading the prompt files from the repository or `catalog.json` and printing their names or descriptions). Implement `show <name>` to print the content of a prompt file. This involves locating the markdown file (the repo's structure might be flat or in subfolders) and streaming its contents to stdout. - **Rendering and Running:** Implement `run <name>`. Initially, since integration with an AI model (Claude/LLM) might require API calls, you could stub this by simply outputting the prompt text or a message. Ultimately, `run` should call an LLM (probably via MCP or a direct API) to actually get a completion for the prompt. For now, focus on reading the prompt, injecting any `--var` values into it (if the prompts use placeholders), and outputting a JSON structure. For example:

```
{ "tool": "<name>", "input": { /* vars */ }, "output": "<prompt text filled or result>" }
```

This ensures the CLI always outputs valid JSON for `run` (even if it's not yet calling the model). This also sets up the shape for MCP tool results. - **Planning:** Implement a simple `plan` command. This could read the front matter of all prompts and, if `--auto` is used, produce a JSON that outlines a possible sequence. For instance, find a prompt with `phase: P1` as a start and chain through `next` references [3]. The output might be something like:

```
{
  "plan": [
    { "step": "/planning-process", "description": "Draft feature plan" },
```

```
    { "step": "/prototype-feature", "description": "Create a prototype" },
    // ...
  ]
}
```

This is a simplistic planning mechanism to have the command available. Later, this could be replaced or augmented by using `claude-task-master`'s own planning logic. - **Local Testing:** At this phase, test each command in isolation. Ensure `prompts list` and `prompts show` work cross-platform (Windows paths vs Unix paths), and that `prompts run` yields well-formed JSON. Add basic unit tests if possible (the Task Master repo has a `tests/` directory [12] that might offer patterns for writing tests, possibly using Jest or Mocha).

Phase 2 – MCP Stdio Server Integration
Goal: Enable the `prompts` CLI to function as an MCP server and interact with `claude-task-master`.
- **MCP Server Mode:** Implement `prompts mcp-server`. This command should switch the CLI into a **server loop**: read JSON-RPC requests from stdin and write responses to stdout continuously (no program exit after one command). There should be an initialization handshake if required by MCP spec (for example, some MCP clients send an `initialize` or `hello` message [6]; our server can respond accordingly). Then it should listen for `{"method": "...", "params": ...}` calls. - For each request, map the method to a prompt tool. A naming scheme must be decided (as discussed in part 2). Perhaps the request's `"method"` field could match the file name or an alias. A robust approach: require methods be prefixed (e.g., `"prompts/<prompt-slug>"`). If a method is unknown, respond with a JSON-RPC error (-32601 Method Not Found). - On a valid tool call, execute the corresponding prompt. This is similar to what `run` does, except now the input comes from the JSON `params`. Perform the prompt rendering and call the AI model if available. If not (for initial version), you can return a placeholder result or echo the input in some way. The response should be a JSON-RPC result, e.g.:

```
```json
{ "jsonrpc": "2.0", "id": 42, "result": { "output": "<text or data>" } }
```

Include any other relevant fields (if a tool should provide structured output,
embed it under `result`). If there was an error (exception, missing prompt),
return a JSON-RPC error object with code and message.
```

- Reuse code from `run` for executing prompts, but refactor so that both `run` and the server use a common function to handle a prompt given an input object.
- **Claude Integration (Optional in this phase):** To actually execute prompts meaningfully, integrate with the Claude API (or OpenAI if using GPT) to get completions. Since `claude-task-master` presumably is built around using Claude's API for AI reasoning, the prompts CLI could similarly use an API key (possibly reading from an env var like `ANTHROPIC_API_KEY`). This might be complex to implement fully, so it could be in a later phase or only done for the server mode. For now, ensure the structure is in place so that when an AI call is added, the rest of the pipeline remains the same.
- **Task Master Config:** Update or create an `.mcp.json` in a test project to include the prompts server. Also, modify `claude-task-master` config if needed so it knows to launch or connect to the

prompts CLI. If the integration is intended to be dynamic, consider adding a feature in Task Master's startup to auto-detect the prompts CLI (maybe via a known file path or a package presence). In the interim, manual config is fine.

- **Test Integration:** Start the prompts MCP server in one terminal, and run a Task Master command that should call it. For example, if Task Master has a CLI command to list available tools or run a specific tool, try to see if it discovers the prompt tools. You might use Task Master's `tasks.json` format or its interactive mode (depending on features of Task Master 2.0). The goal is to see a JSON request come into `prompts mcp-server` and a correct response go back. Debug any mismatches in JSON formatting or encoding. Use logging liberally; e.g., have `prompts mcp-server` print to stderr what requests it's handling (since stdout is reserved for protocol data).
- **Concurrent Requests:** Consider whether to handle one request at a time or allow concurrent processing. For simplicity, start with single-threaded processing (one prompt at a time). MCP JSON-RPC can handle multiple requests (and even batch requests), but managing concurrency would require threading or child processes. We can document that as a future improvement for performance. Node's event loop can manage overlapping I/O, but if an AI call is in progress, we might queue other requests or spawn separate processes for isolation.

Phase 3 – Enhanced Features and Hardening
Goal: Expand functionality and robustness of both CLI and integration, drawing on community best practices and advanced use-cases.
- **CLI Framework & Plugins:** If the CLI grows complex, consider migrating to **Oclif** [7] in this phase for better structure (especially if plugin architecture or an update mechanism is desired). Oclif would allow packaging commands as plugins (for example, if other prompt packs or tools want to extend the CLI). However, note that Oclif might be overkill for the current size; we might stick with Commander for MVP and revisit if scaling up. - **Cross-Platform Testing:** Ensure the CLI works on Windows (particularly file path handling, newline issues for JSON output, etc.). Use CI to run tests on Linux, Windows, macOS if possible. For instance, test that `prompts list` works in a Windows shell, and that `prompts mcp-server` can be started via a Windows command (there might be differences in how processes detach or signals). - **Logging & Telemetry:** Implement a verbose mode (e.g., `prompts --verbose` or an environment variable) to output debug info. This helps users troubleshoot integration issues. If desired, incorporate a telemetry option (opt-in) to track usage or performance, but be mindful of privacy and the context of these tools (likely local developer tools, so telemetry might not be critical). - **Error Recovery & Retries:** If a prompt fails (e.g., AI service fails, or schema validation fails), determine retry logic. Possibly use an exponential backoff for transient errors (like API rate limits). The `p-retry` npm package [7] could be used to wrap API calls with retry logic. Also, `p-limit` [7] can control concurrency if we later allow multiple prompt executions at once (to avoid hitting API limits). - **Cache and Idempotency:** For idempotent operations like planning, caching results could save time if nothing changed. For instance, if `prompts plan` was already generated and the environment is the same, we might reuse the plan (unless forced to refresh). Similarly, if certain prompt outputs are deterministic given the same input (not usually the case with AI, but maybe for static or pseudo-random tasks), caching those within a session can prevent duplicate API calls. Implement a simple in-memory cache or use Task Master's session state to store outcomes. - **Integration with Editors/IDE:** Since MCP is designed for tool use in editors (Cursor, VS Code via Claude, etc.), consider adding a command like `prompts init` that sets up necessary config in a project to use the prompts with those editors. For example, it could create a `.mcp.json` with the prompts server entry, or a VS Code tasks configuration that knows how to launch it. This eases adoption: a user can run one command and get the integration scaffolded. - **Documentation and Examples:** By this phase, update the README in `prompts` repository to include usage instructions for the CLI. Provide example JSON inputs and outputs for each command.

Possibly add a tutorial in the docs (e.g., "How to integrate prompts CLI with Task Master for your project"). Also, ensure that any new code is well-documented (JSDoc comments can help; the snippet referencing `execa` above shows how you can spawn processes [7], and using such utilities should be documented in code).

Phase 4 – Advanced Orchestration and Multi-Tool Coordination
Goal: Incorporate more sophisticated orchestration capabilities, drawing from advanced frameworks and ensuring long-term maintainability.
- **LangChain/LangGraph Bridges:** Investigate if libraries like LangChain offer MCP bridges or tool management that we can leverage. For instance, *LangChain's MCP integration* (if any) or similar projects (Smithery, etc.) could provide patterns for multi-step tool calls. This might help implement complex flows like tool-selection or dynamic prompt selection during a session. If a suitable open-source project exists that already provides a "plan-execute" loop using JSON I/O (like a planner that outputs a chain of tool calls), consider integrating or at least learning from it. - **Parallel Task Execution:** Enable Task Master to call multiple prompt tools in parallel if the plan allows (for example, two independent subtasks). This may require the prompts server to handle concurrent requests. Options include spawning a new worker process for each request (Node's cluster or worker_threads, or simply multiple instances launched on different ports and load-balanced). Alternatively, using an async queue within the single process could suffice if the AI calls are the main wait (since they are I/O-bound, Node can handle concurrent calls fairly well). - **Result Aggregation and Persistence:** Improve how results are collected and stored. For instance, ensure `claude-task-master` writes outputs in a structured format (it likely already does, e.g., tasks.json or results.json). The prompts CLI could also offer to save outputs of `run` to a file, or integrate with Task Master's storage (maybe through a shared folder or an API call). This is especially useful for long sessions – consider implementing a lightweight database or using Task Master's existing logging mechanism for continuity. - **User Feedback Loop:** Implement interactive modes where the user can approve or adjust steps. For example, a `prompts plan` could open an interactive session (maybe in a TUI) to let the user remove or reorder steps before executing with Task Master. This crosses into UX design for CLI, but can be powerful for adoption. - **Plugin Loading Across Repos:** If the `prompts` CLI becomes a generic framework, allow it to load prompt definitions from multiple sources (like other repos or npm packages). This way, `claude-task-master` (or any other project) could bundle its own prompts and have them added to the `prompts` CLI tool registry. Using a convention (like placing prompt files in a known directory or providing a config listing external prompt paths) can achieve this. Ensuring that cross-repo plugin loading is safe is important (only load trusted prompt definitions, or have a review step, to avoid executing malicious prompts). - **Continuous Improvement via Community:** Given that `claude-task-master` is a popular open-source project (over 22k stars on upstream [13]), engage with its community. New versions (Task Master 2.0 and beyond) might introduce changes to how integration works (for example, new config file formats or additional capabilities). Keep the `prompts` CLI updated to remain compatible. Also, welcome contributions from the community to add prompts or improve the CLI (which ties into having clear contribution guidelines as noted below).

Roadmap Code Snippets and References

Throughout the roadmap, here are some key reference points and code examples aligned with our tasks:

- **Commander vs Oclif vs Yargs:** The CLI frameworks choice was guided by familiarity and the need for plugins. **Commander.js** is sufficient for an MVP CLI and is used via simple import and chaining API [7]. If scaling up, **Oclif** provides a structured approach for large CLIs with many commands or

plugin support [14]. **Yargs** is another alternative for quick CLI building but less robust if we anticipate complex parsing. Given our CLI's scope, Commander is a good start [8].

- **MCP Protocol Handling:** The Model-Context Protocol spec [6] and in particular the **Transports** documentation [9] describe how to format JSON-RPC messages and route them over stdio or other channels. We base our MCP server implementation on these guidelines. For example, sending a JSON-RPC request in Node might look like:

```
const request = { jsonrpc: "2.0", id: 1, method: "someTool", params: { /*...*/ } };
process.stdout.write(JSON.stringify(request) + "\n");
```

  and reading responses via `process.stdin.on('data', ...)`. We may use existing JSON-RPC libraries, but since the protocol is simple, manual handling or a light wrapper is fine.
- **Process Management:** Using **execa** (by Sindre Sorhus) for spawning subprocesses provides a nice interface for stdio streams [15]. For instance, Task Master could use execa to launch `prompts mcp-server` and keep the process handle. We too might use execa in tests or if `prompts` CLI ever needs to launch Task Master (inversely). Execa's documentation [15] shows examples of running a command and piping streams, which is directly applicable for connecting the two processes.
- **Retry & Concurrency Utilities:** The mention of `p-limit` and `p-retry` relates to controlling promise concurrency and retrying failed operations respectively. These are lightweight npm packages [7] that can wrap around our AI API calls or tool executions to make the system more robust in the face of transient failures (e.g., API timeouts).
- **Existing Tools Inspiration:** *MCP Inspector* [16] is a GUI tool to test MCP servers. We could use it during development to verify that our `prompts` MCP server correctly lists tools and responds. Similarly, **Claude Desktop/VS Code** integrations might have docs or code examples on how they launch/connect to MCP servers (for instance, how Cursor or Windsurf launches Task Master). Though not cited above, one can refer to those projects for integration patterns.

Finally, ensure all code added adheres to the licensing and contribution standards of each project, as discussed next.

Additional Notes: Licensing and Contribution Guidelines

- **Licensing:** The `claude-task-master` project is open source under the **MIT License with a Commons Clause** [6]. This means the core is MIT (permissive use, modification, distribution allowed) **except** you *cannot sell* the software or offer it as a paid service (Commons Clause restriction) [11]. Anyone integrating or redistributing Task Master or derivatives (like our integrated CLI) must respect these terms. The `AcidicSoil/claude-task-master` fork inherits this license. The `AcidicSoil/prompts` repository does not explicitly list a license file in the content we reviewed, which implies it may default to "All Rights Reserved" or require clarification from the owner. Given it extends the "Codex CLI prompts," it might be intended for free use within that ecosystem. It would be prudent to add an open-source license (e.g., MIT) to `prompts` if it's to be widely used, ensuring others can legally use and contribute to it. Always check for a LICENSE file or ask the maintainer for clarification if planning commercial use of `prompts`.
- **Contributor Guidelines:** Both repositories have instructions for contributors:

- The `prompts` **repo** includes a `CONTRIBUTING.md` [17] [18] that outlines how to maintain prompt metadata. It specifies to run validation (`npm run validate:metadata`) and rebuild the catalog (`npm run build:catalog`) before committing changes [18]. This ensures every prompt's YAML front matter remains consistent with the defined workflow and that the index (`catalog.json` and README tables) is up-to-date. Contributors are expected to follow this process so that CI checks pass. There's also an implied development workflow: clone into `~/.codex/prompts`, use npm scripts, etc., which new contributors should read about in the README [19].
- The `claude-task-master` **repo** (upstream) likely has its own contributing guidelines (the original repository by eyaltoledano has a `CONTRIBUTING.md`, hinted by the find results). Common practices would include discussing significant changes via issues or discussions, writing tests for new features (since the repo has a tests suite [12]), and following the project's coding style. Given the large number of stars and forks, the project is active; contributors should also pay attention to its release notes and roadmap (Task Master 2.0 announcements, etc.). We did not retrieve specific contributing steps for Task Master, but generally one should fork the repo, create a feature branch, run all tests (perhaps `npm test` or similar) and ensure compatibility with existing functionality before submitting a pull request.
- It's important to note that when contributing to Task Master, the **Commons Clause** doesn't prevent contributions; it only restricts certain uses. However, any code contributed will be under that same license, so contributors must agree to those terms.

In summary, `prompts` appears to be a content-oriented repo that will benefit from an open-source license and already has strict contribution checks for consistency, while `claude-task-master` is a widely-used MIT+CommonsClause project encouraging community involvement under a shared understanding not to commercialize it directly. Aligning our new CLI work with these guidelines (e.g., including proper license headers, following commit message conventions, and running validation/tests) will smooth the integration into their ecosystems.

References:

1. Model-Context Protocol – *Specification (2025-03-26)* [6] [11] – Defines the JSON-RPC based protocol for AI tool interoperability, including transports and message schema.
2. Node.js Documentation – *Process I/O* [6] – Describes how to work with stdin/stdout in Node, relevant for implementing the MCP stdio server.
3. **Commander.js** (npm package) [7] [8] – A popular Node.js library for building CLI interfaces; used for parsing commands and options in our CLI implementation.
4. Model-Context Protocol – *Transports Guide* [9] – Details on how MCP messages are transmitted (stdio, sockets, etc.), ensuring our CLI and Task Master use a compatible approach.
5. **AcidicSoil/prompts** – *GitHub Repository* (Prompt catalog and tooling) [1] [2] – Source of prompt files and metadata; basis for designing the prompts CLI's commands and data structures.
6. **AcidicSoil/claude-task-master** – *GitHub Repository Fork* (Claude Task Master code) [20] [6] – Provides the task management and MCP orchestration logic that our CLI will integrate with.
7. **MCP Inspector** (fazer-ai/mcp-inspector) [16] – A tool for testing and visualizing MCP servers, useful for debugging our MCP stdio implementation during development.
8. **execa** (Node.js process execution library) [7] [8] – Simplifies spawning child processes and capturing their output; can be used if our CLI needs to launch external processes (or by Task Master to launch our CLI).

9. **p-limit** (npm) [7] – Utility for limiting concurrency of asynchronous operations, relevant if the CLI later executes multiple prompts in parallel.
10. **oclif** – Open CLI Framework docs [14] – Provides patterns for building more complex CLI applications with plugin architecture; a consideration for future CLI versions.
11. **LangChain MCP (mcpdoc)** (langchain-ai/mcpdoc) [20] – Example of exposing llms.txt (prompt collections) to IDEs via MCP, indicating how prompt libraries can bridge into development workflows.
12. **Contributing Guidelines for prompts** [17] [18] – Instructions for contributors to maintain consistency in the prompt metadata and project structure, ensuring any extensions to the CLI also respect these practices.

---

[1] [2] [3] [4] [5] [10] [19] README.md
https://github.com/AcidicSoil/prompts/blob/a3918aa24c76570f3e14aca4671a9ee2a8fbbb52/README.md

[6] [7] [8] [9] [11] [12] [13] [14] [15] [16] [20] GitHub - eyaltoledano/claude-task-master: An AI-powered task-management system you can drop into Cursor, Lovable, Windsurf, Roo, and others.
https://github.com/eyaltoledano/claude-task-master

[17] [18] CONTRIBUTING.md
https://github.com/AcidicSoil/prompts/blob/a3918aa24c76570f3e14aca4671a9ee2a8fbbb52/CONTRIBUTING.md