ChatGPT

Executive Summary

Goal: Enable a *"write once, run anywhere"* plugin system for AI coding assistants, so that custom multi-agent workflows, tools, and commands can be developed *once* and then deployed both in **Anthropic Claude Code** and **GitHub Copilot (VS Code/Visual Studio)** environments. The vision is to package up coding assistant "skills" – such as specialized agents (roles), event hooks, slash commands, and tool integrations – in a portable format that can be recognized and executed by multiple AI coding copilots (Claude, OpenAI/ChatGPT, Google Gemini, etc.). This would allow development teams to share and reuse automation workflows across different AI coding assistants, maintaining consistency and avoiding duplicate effort.

Claude Code's plugin architecture already provides a template for such extensibility. A Claude Code plugin is essentially a bundle of custom capabilities – *slash commands*, *sub-agents* (specialized AI agents), *skills* (contextual knowledge modules), *hooks* (event-triggered actions), and *MCP connectors* (tool/API integrations) – defined by a manifest. These components work in concert to extend Claude's base functionality [1] [2]. For example, a plugin might define a code-review agent with domain-specific knowledge, a set of slash commands to invoke certain analyses, hooks to automatically run tests on file save, and an external tool connector to query a database. Claude Code's plugin system supports complex multi-step workflows by letting **multiple agents collaborate** (e.g. a *planner* sub-agent that drafts a solution and a main agent that implements it [3]), with guardrails like tool permission prompts and separate context windows for sub-agents [4]. These plugins are packaged with a `plugin.json` manifest (in a `.claude-plugin` directory) listing the plugin's contents, and distributed via marketplaces for easy sharing [5] [6]. When installed, Claude Code loads the plugin, registering its commands, agents, skills, etc., and sandboxes it by copying files to a cache for security [7] [8]. Plugins run within the user's environment, so they can execute local shell commands, modify files, or call external APIs (with appropriate user permissions) [9] [10]. In short, Claude Code plugins turn a personal `.claude/` config into a **team-scale, versioned extension** with namespaced commands, event automation, and specialized AI behaviors [11] [12].

GitHub Copilot, on the other hand, does not (as of early 2026) have an identical "plugin" concept, but it has a rich set of **extensibility points** that can achieve similar outcomes. Copilot in VS Code supports:

- **Custom AI agents** (called *Custom Chat Modes* or now *Custom Agents*) defined by markdown profiles (`.agent.md` files) which specify an agent's persona, instructions, and allowed tools [13] [14]. These let you create Claude-like sub-agents (e.g. a "Planner" agent that only reads and plans, or a "Security Auditor" agent with special guidelines). Copilot even allows linking agents in sequence with *handoffs* – suggested next actions that switch the conversation to a different agent along with context [15] [16] (e.g. a *Planning* agent can offer a *"Start Implementation"* handoff that switches to an Implementation agent) – thereby enabling multi-agent workflows akin to Claude's (though sequential rather than truly concurrent).

- **Reusable prompt templates** (*Prompt Files*) which act like slash commands. These are `.prompt.md` files (stored in a `.github/prompts` folder for a workspace) containing preset instructions and optional input parameters. When the user types `/` in Copilot chat, they can invoke these by name to insert the predefined prompt [17] [18]. For example, a `test-gen.prompt.md` file (with

frontmatter specifying `name: test-gen`) could be triggered via `/test-gen` to instruct Copilot to generate tests for the current file [19] [20]. This is analogous to Claude's markdown-based slash commands (though Copilot's are local to the project/user unless shared via source control).

- **Agent "skills"** (also called *Agent Skills*, following an emerging open standard [21]). These are very similar to Claude's auto-invoked skills: they are placed in a `skills/` folder (e.g. `.github/skills/skill-name/SKILL.md`) and contain a name, description (when to use the skill), and detailed instructions or data [22] [23]. Copilot monitors the conversation for triggers matching a skill's description; when relevant, it **automatically loads the skill's content into context** (progressive disclosure: only the description is always loaded, and the full `SKILL.md` content is pulled in on demand when the task matches [24] [25]). This parallels Claude Code's *Agent Skills*, and indeed skills are meant to be *portable across AI platforms* [26] – the same `SKILL.md` structure is supported by Claude Code and Copilot (and even CLI tools), which is a promising basis for our portable spec.

- **Built-in and extensible tools** (analogous to Claude's tool palette). In Copilot's *"Agent mode"* (autonomous multi-step mode), the AI can call various tools: e.g. reading or writing files, running terminal commands, searching the codebase, browsing documentation, etc. VS Code's Copilot implementation exposes an API for extensions to **register custom tools** (via the *Language Model Tool API* or the newer *Model Context Protocol (MCP)* standard) [27] [28]. For instance, Microsoft's MSSQL extension registers a *chat participant* ("@mssql") that contributes slash commands and also implements an MCP server allowing the AI to run SQL queries against a database [29] [30]. Likewise, any extension can define an MCP server that provides additional *tools* to Copilot's agent (for use in its reasoning steps) or *prompts/resources* that the user can inject [31] [32]. This is directly comparable to Claude's MCP connectors (which similarly let Claude call out to external APIs/services via a standardized interface). The MCP in VS Code supports connectors over stdio or HTTP, with features like *elicitation* (asking the user for input, e.g. an OAuth login), *resource browsing*, etc. [28] [32]. In short, Copilot can be extended with *plugin-like tools*, but via VS Code extensions rather than a simple manifest drop-in.

- **Event handling and automation**: Copilot does not yet expose "hook" events (like Claude's PreToolUse or OnFileSave hooks) in a simple declarative config. However, one can achieve similar automation via traditional extension code. A VS Code extension can subscribe to editor events (file saves, test runs, etc.) and then invoke Copilot actions programmatically. For example, an extension could watch for a test suite finishing and then programmatically prompt Copilot to analyze failures. This isn't as straightforward as Claude's JSON-based hooks configuration [33] [34], but it is feasible. (Notably, Claude Code's hook events like *Stop* or *SubagentStop* [35] don't have direct analogs in Copilot's public API, since Copilot's agent loop is largely internal. Achieving similar guardrails might require the extension itself to mediate the agent's steps – for example, running one step at a time and evaluating if it should continue.)

In summary, **Claude Code provides a turnkey plugin system** with a manifest that bundles all extension types, whereas **GitHub Copilot's extensibility is more fragmented** – it has *custom agents*, *prompt commands*, *skills*, etc., which can be combined to emulate a plugin's functionality, but there's no single "plugin pack" format for them yet. We can bridge this gap by defining a **Portable Plugin Specification** that serves as the source of truth, and building adapters to deploy it to each platform: - For **Claude Code**, the adapter would generate a standard Claude plugin folder (with `plugin.json`, and subfolders for

commands, agents, skills, hooks, etc.) that Claude can load natively. - For **Copilot/VS Code**, the adapter could have two modes: (a) a *zero-install mode* that outputs the appropriate `.github/` files (custom agents, prompt files, skills) directly into a repository so that Copilot Chat will pick them up for that project; and (b) a *VS Code extension* that can be installed to provide any additional capabilities (for instance, if we need a background service or to handle events/permissions beyond what the file-based approach offers). This extension could register a Copilot Chat **participant** (if needed) or an MCP server to supply tools and coordinate multi-agent workflows. The extension would read the same portable plugin spec (perhaps packaged with the project or embedded in the extension) and dynamically register equivalent Copilot features (slash commands, etc.).

Outcomes: By following this blueprint, we expect: - A clear mapping of Claude plugin concepts to Copilot concepts, identifying which features align well (e.g. Skills and custom agents) and which need creative workarounds (e.g. hooks/event automation). - A **portable plugin manifest schema** to describe plugin components in an agnostic way (agents, commands, skills, etc., plus metadata like permissions or required tools). - A reference implementation plan (including repo structure and a development timeline) for creating the core runtime and environment-specific adapters. - An understanding of **risks and limitations**: e.g. Copilot might not support fully autonomous multi-agent loops without user approval at each step, or might restrict certain operations for security; we will outline how to mitigate these (or at least flag them) in a cross-platform plugin design.

The following sections provide a deep dive into Claude Code's plugin model (our source of truth), GitHub Copilot's extensibility mechanisms (our target runtime), the design of a unifying plugin specification, and concrete steps to implement and test this system. A side-by-side capability matrix and a "Hello World" example plugin are included to ground the discussion. Finally, we discuss the development roadmap, security considerations, and a checklist for ensuring portability.

Claude Code Plugin Architecture

Claude Code's plugin system is **comprehensive and modular**, built around a *manifest* that declares various extension components. At runtime, Claude Code (the CLI/editor environment) loads installed plugins and integrates their functionality into the assistant's behavior. The key building blocks of a Claude plugin are:

- **Plugin Manifest (`plugin.json`)**: A JSON file containing metadata (name, version, description, author, etc.) and pointers to the plugin's components [36] [37]. Notably, the manifest can list custom paths for commands, agents, skills, hooks, MCP servers, etc., but by default Claude looks for standard directories (as below) [38] [39]. The plugin's *name* also serves as a namespace for its slash commands (avoiding conflicts by prefixing commands with `name:`) [40] [41].

- **Slash Commands**: Custom user-invoked commands, defined as markdown files under `commands/`. Each file (e.g. `hello.md`) represents a command; its filename becomes the command name (prefixed by plugin name). The content of the markdown is essentially what Claude should do when the command is run. Claude Code will include the markdown's content (after a YAML frontmatter) as part of its response generation. For example, a `hello.md` might simply instruct Claude to greet the user [42] [43]. Commands can accept arguments – in the markdown, special placeholders like `$ARGUMENTS` capture user-provided text after the command [44] [45]. This is how one can implement parameterized commands. In execution, when a user types `/myplugin:hello Alice`,

Claude will process the `hello.md` content with `$ARGUMENTS="Alice"` and produce the result. Commands run inline within Claude's normal operation (no separate process), and they **cannot directly execute code** – they rely on Claude's AI capabilities and tool use. (If a command needs to perform an action like running tests or modifying files, Claude must do so via its tools like Bash, Edit, etc., possibly requiring a permission if destructive.)

- **Custom Agents (Subagents)**: These are secondary AI agents defined in the plugin (under `agents/` directory, each as a markdown file). A subagent markdown file typically contains a YAML frontmatter with a *name* and *description* (and optionally a specified `model` to use, e.g. Claude Instant vs. Claude-2, and tool access restrictions) [46] [47], followed by the agent's system prompt/persona instructions in the body. Once loaded, users can invoke a subagent by name (e.g. by mentioning `@agentname` in chat or using the `/agents` command to switch) [48]. In conversation, a subagent runs with its **own context window** (isolated from the main chat's history except what is explicitly given) and potentially a narrower toolset – this is a design to help focus the agent and save tokens [49] [47]. Subagents are typically used for **multi-step workflows**: they can be called to perform planning or specialized analysis, then hand results back to the main agent. For example, one might have a `planner.md` subagent that generates a step-by-step plan (using only read/query tools), and a main agent that then executes the plan (using write/edit tools). This pattern – "subagent analyzes, main Claude executes" – has been observed as an effective strategy [50] [51]. Claude Code treats subagents as "tools" themselves in a sense: there is a *Task* tool that spawns a subagent and returns its output to the main flow [52]. Plugins simply provide new subagent definitions, effectively adding new expertise "personas" Claude can deploy.

- **Agent Skills**: Self-contained knowledge or ability modules that Claude can auto-invoke. In a plugin, skills are defined under `skills/<skill-name>/SKILL.md` (each skill in its own folder) [53] [54]. The `SKILL.md` has a YAML header with `name` and `description`, and the body contains guidance/information [55] [56]. Claude will automatically include a skill's content when its description "triggers" – i.e., when the conversation context seems to match the skill's intended usage [57] [58]. For example, a skill description might say "Auto-invoke when user asks for code review" and the body might contain a checklist for code quality [56] [58]. Skills are a way to give Claude *persistent knowledge or behavior* beyond the prompt, without the user explicitly calling them each time. They load transparently when relevant. In plugin development, skills let you package domain expertise (security guidelines, architecture principles, etc.) that your agents can tap into on-the-fly [59]. Under the hood, skills function by matching the *description* against the task – the quality of this description is crucial for proper activation (devs have learned to use a WHEN/WHEN NOT format to be precise) [60] [58]. Multiple skills can trigger together if relevant, and Claude merges their instructions.

- **Hooks**: Event-driven triggers that run automatically in response to certain situations in the coding session. Plugin hooks are declared in a `hooks.json` file (or multiple JSON files) under `hooks/` [61] [62]. The hooks JSON defines, for each event (like `PreToolUse`, `PostToolUse`, `UserPromptSubmit`, etc.), a list of actions to take. These actions can be of type `"command"` (execute a shell command or even a Claude slash command) or type `"prompt"` (invoke an LLM prompt to decide something). For example, a plugin might use a PostToolUse hook to auto-run `npm run lint:fix` after the `Write` tool is used (i.e., after Claude writes code to a file) [34]. Another example: a PreToolUse hook could intercept a dangerous tool usage (like deleting a file) and ask for confirmation or impose a policy check [63] [64]. Hooks effectively allow automation and guardrails – they can allow or deny tool usage (including simulating a "user always clicks yes" for

permissions) by returning an `ok: true/false` JSON through a prompt hook [65] [66]. They can also perform side-effects via shell commands (for instance, a Notification hook could play a sound or send an alert when a long process finishes) [67] [68]. In Claude's architecture, hooks run in-process: a command-type hook runs a local script via Claude's own tool mechanism (Bash tool), and a prompt-type hook actually asks the Claude model to make a decision by presenting it with a question and expecting a JSON answer [69] [70]. The plugin developer can thus inject logic at key points of the AI's operation. The set of hookable events in Claude Code includes: **PreToolUse**, **PermissionRequest** (when Claude asks the user to approve a tool), **PostToolUse**, **Notification** (various internal events), **UserPromptSubmit** (just as user hits enter), **Stop** (when main agent finishes an answer), **SubagentStop** (when a subagent finishes) and **PreCompact** (before Claude compacts context) [9] [71] [72]. This breadth means plugins can automate tasks (run builds on every edit, etc.), enforce policies (stop generation if it violates certain rules), or coordinate multi-agent sequences (detect a subagent's output and initiate the next step, akin to a pipeline).

- **MCP and LSP Servers**: For advanced integrations, a plugin can include definitions for *Model Context Protocol* servers and *Language Server Protocol* servers. An **MCP server** allows Claude to interface with external processes or APIs through a consistent protocol. In a Claude plugin, the `.mcp.json` (or an entry in `plugin.json`) can define servers to launch when the plugin loads [73] [74]. For example, a plugin could start an API client as an MCP server that Claude can send queries to (Anthropic's docs show specifying a command like `npx @company/mcp-server` to run, which Claude will treat as a new tool) [75] [74]. Once running, the server's capabilities appear as tools that Claude can call (the server communicates via JSON over stdin/stdout or HTTP per the MCP spec). This is how Claude can be extended to do things like interact with a web browser, run complex analyses in Python, etc., beyond its built-in tools [6] [76]. Similarly, **LSP servers** (configured via `.lsp.json`) let the plugin provide language intelligence (autocomplete, refactoring support) to Claude's editor interface [77] [78]. For our purposes, MCP is more relevant (since it parallels extending Copilot with external tools). The key point is that *Claude plugins can bundle code to be executed* (in contrast to skills/commands which are just text for the model). That code can be managed via MCP so that the model treats it as part of its toolkit.

These components are **packaged in a standard structure**. Recapping from Claude's docs, a typical plugin layout is:

```
my-plugin/
├── .claude-plugin/
│   └── plugin.json       (manifest)
├── commands/             (markdown files for slash commands)
├── agents/               (markdown files for subagents)
├── skills/               (directories each containing a SKILL.md)
├── hooks/                (hooks.json configuration)
├── .mcp.json             (optional MCP server config)
└── .lsp.json             (optional LSP server config)
```

[79] [80]. Only the manifest goes inside the special `.claude-plugin` dir; other folders are at root level (this tripped up some new developers) [81] [82]. When Claude installs or loads a plugin, it copies this whole directory into its plugin cache (isolating it) [7] [83].

Plugin Lifecycle: In use, a Claude plugin's lifecycle is: (1) Developer creates the plugin files and tests locally with `claude --plugin-dir path/to/my-plugin` (this CLI flag loads an uninstalled plugin for dev/test) [84] [85]. (2) The plugin is distributed (e.g. published to a Git-based marketplace or shared privately) – essentially just the files in a zip or git repo. (3) A user installs the plugin via Claude's `/plugin install` command (specifying a marketplace and plugin name/version), which fetches the plugin files and adds an entry in Claude's settings JSON to enable it [86]. It can be installed as a user plugin (available globally) or project plugin (listed in the project's `.claude/settings.json`) [87]. (4) Claude Code (on startup) loads all enabled plugins: it reads each manifest, registers each slash command (so `/pluginname:cmd` becomes available), registers agent definitions (so they appear in the `/agents` list), loads skill descriptions (so they're considered for auto-invoke), and starts any MCP servers, etc. [88]. Once loaded, the plugin's functionality is at Claude's disposal: the user can call its commands/agents, and Claude will automatically use its skills or triggers its hooks when appropriate.

Permissions & Security: Claude Code runs in the user's environment, so a plugin's capabilities that perform actions (like running a Bash command or connecting to the internet) are subject to Claude's permission system. Claude will typically **prompt the user for permission** the first time a plugin tries to use a tool that could be sensitive – e.g., running a shell command, or writing to a file outside the allowed scope [89] [90]. Hooks can be used to auto-grant or deny these (for power users or enterprise policy) [91]. Also, the plugin cannot escape its folder in terms of file references – any attempt to access files outside the plugin's directory (during installation copy) is blocked unless explicitly symlinked [92] [93]. This prevents a malicious plugin from arbitrarily reading files on the user's system just by referencing `../` paths. There's also a **"strict" mode** field in marketplace entries that can enforce verification of plugin files. In general, because Claude Code executes plugin hooks and MCP processes locally, one must trust the plugin source or audit it (similar to VS Code extensions).

In essence, Claude Code's plugin system provides **first-class support for multi-agent orchestration and tool use**: you can define multiple agents that collaborate (the developer must script the logic in their prompts or via hooks), and you can extend the AI's toolset safely. This is exactly what we want to replicate in GitHub Copilot.

(For a visual summary, the Claude plugin architecture can be thought of as: an AI Orchestrator (Claude) that loads a set of capability modules from each plugin – commands for direct user invocation, sub-agents for specialized tasks, skills for contextual knowledge, hooks for automation, and connectors for external tools. All of these modules ultimately influence the AI's behavior within guardrails.)

GitHub Copilot Extensibility in VS Code/Visual Studio

GitHub Copilot (especially "Copilot X" features in VS Code as of 2025) has evolved into a more agentic system as well, though it presents it a bit differently. Instead of user-installed "plugins" with a manifest, Copilot offers **multiple extension points** and configuration files:

Built-in Copilot Features Relevant to Plugins

- **Copilot Chat and Agent Modes**: Copilot has a chat interface in VS Code (and Visual Studio) which the user can interact with using natural language. By default, Copilot Chat operates in either an **"Ask" mode** (single-turn Q&A type interactions, often called the "Chat" or "ghost text" mode) or an **"Edit" mode** (conversational code editing) [94]. Additionally, Copilot includes an **"Agent" mode** which is more autonomous, executing multi-step tasks and using tools – very analogous to Claude's main coding agent. In fact, agent mode can "automatically handle compile and lint errors, monitor terminal output, and iterate until the task is complete" [95]. This implies Copilot's agent mode already does some internal hooking: e.g. it notices a compile error (likely via a background watcher or tool) and decides to fix code, looping until success. While these specifics are mostly internal, they show that Copilot has an *orchestrator loop* conceptually similar to Claude's – but currently, **customization of that loop (like injecting new steps or different multi-agent sequences) is limited**. We may leverage agent mode's existing behaviors (like using tools sequentially) and extend its toolkit to approximate multi-agent workflows.

- **Slash Commands and Prompt Files**: Copilot Chat supports **slash commands** that help users insert common prompts or perform predefined actions. Out of the box, there are commands like `/tests` (which instructs Copilot to write tests for the selected code) [96], `/explain` (to explain code), etc. These are essentially shortcuts for user prompts. Importantly, **users or extensions can add new slash commands** in two ways:

- *Prompt Files:* If a repository contains a `.github/prompts/` directory with `*.prompt.md` files, each will become a slash command in Copilot Chat [97] [17]. The frontmatter `name:` defines the slash command name if present [98] [17] (otherwise the filename is used). The frontmatter can also specify which **agent and tools** to use when executing that prompt [99] [100]. For example, a prompt file might specify `agent: agent` (meaning use the default "coding" agent mode) and `tools: ['terminal','search']` to allow the model to use the terminal and code search tools for that prompt. The body of the file is the actual prompt content that gets sent to the model. Using prompt files, one can mimic most Claude slash commands. However, prompt files are static – they don't themselves execute code, they just provide canned AI instructions. Any action (like editing a file) will be done by Copilot if the prompt leads it to do so with its tools.

- *VS Code Extension Commands:* An extension can contribute slash commands programmatically via the Chat Participant API [101] [102]. We saw this with the MSSQL extension, which adds an `@mssql` participant that has its own set of slash commands (like `/runQuery`) [29] [103]. In an extension's `package.json`, under `contributes.chatParticipants[*].commands`, you can list commands with names and descriptions [104] [105]. These become available under that participant's namespace or sometimes even globally (the UI shows them when you type `/`). When invoked, those commands typically trigger extension code or insert a prompt. For example, selecting `/runQuery` might cause the extension to run the query and stream results into chat [106] [107]. This approach is more powerful than prompt files because the extension can do something *immediate* (like actually run a DB query) and then optionally involve the AI (e.g. to explain the results). For our portable plugin, this means if we need slash commands that perform real actions (not just instruct the AI), we'd likely implement them via an extension participant rather than just a prompt file.

- **Custom Chat Participants (Agents)**: VS Code allows creating **chat participants** – essentially custom agents that the user can summon with an `@name` mention [108]. Think of these like having an AI specialist alongside the default Copilot. Microsoft provides built-ins like `@vscode` (which can answer questions about VS Code itself) and `@workspace` (which has knowledge of the workspace context) [109]. Through the *Chat Participant API*, we can create our own (say `@myplugin`). A chat participant extension owns the conversation when invoked – it can decide how to handle the user's prompt, possibly by calling the language model with certain system prompts or using external tools, etc. [110] [111]. In practice, a custom participant often will pipe the query to the OpenAI model but with a custom prompt prefix or using a different knowledge base. For example, an extension could implement `@docs` that, when the user asks it something, the extension searches documentation and then crafts a prompt for the LLM. This is somewhat different from Claude's subagents because participants are explicitly invoked by the user (or auto-routed via detection) and *the extension code can intervene in constructing the answer*, not just the model alone. For portability, however, we might not need a new participant for every plugin – using Copilot's native capabilities (prompt files, custom agents, skills) might suffice. But if we wanted an **autonomous multi-agent workflow** (like one agent planning and another executing without manual user switching), an extension could internally use participants or multiple LLM calls to orchestrate that. For example, the extension could catch a slash command and then internally call the planning agent prompt, then feed its output to the implementing agent prompt, and so on – effectively performing what Claude's multi-agent chain would do. This is complex but possible.

- **Custom Agents (User-defined)**: In addition to extension-defined participants, Copilot recently introduced **user-defined custom agents** via `.agent.md` files. These parallel the Claude subagent concept: in a repo's `.github/agents/` folder, you can create `MyAgent.agent.md` with a YAML frontmatter and instructions. The YAML can specify which tools this agent can use, which model to use, etc., and you give it a name (like "MyAgent") [112] [14]. Once present, this agent appears in the Copilot chat *"Agents" dropdown*, and you can manually switch the chat to that agent persona. When active, the agent's instructions are automatically applied to all prompts [113] [114]. It's like switching the AI's role. Copilot even supports *handoffs* between custom agents (as described earlier) to facilitate moving from one to another in a workflow [115] [116]. Under the hood, custom agents are similar to prompt files but persistent for the session: they prepend their content as system instructions for each user query when that agent is selected [113]. They can restrict available tools: e.g. a planning agent might list only read-only tools in `tools: [...]`, so that while it's active, Copilot's agent won't (or shouldn't) invoke disallowed tools. This is enforced by the Copilot runtime. For our needs, custom agents give us a straightforward way to port Claude subagents: we can take the content of a Claude agent markdown and use it as a `.agent.md` for Copilot. Indeed, both support similar fields (Claude's agent MD supports `name`, `description`, `model`, and optional `tools` limit [47]; Copilot's agent MD supports `name`, `description`, `model`, `tools`, etc. [117] [118]). One minor difference: Claude can *dynamically* call a subagent for one query (like `@agent do X` and then back), whereas in Copilot, switching to an agent tends to persist until you switch back. However, using handoffs or prompt files that specify an agent, we can emulate one-shot invocations.

- **Agent Skills**: As discussed, Copilot supports the same concept of skills, and it even recognizes `.claude/skills/` as legacy location [119] (showing alignment with Claude's implementation). In VS Code, one must enable the preview setting `chat.useAgentSkills` to have Copilot load skills [120], but presumably by 2026 this is standard. Skills are loaded from both project and user locations and are automatically matched to queries [24] [25]. The benefit is we can share skill files between Claude

and Copilot with minimal or no changes. Skills can also include **resources** like scripts or example files which the model can be directed to use (the skill doesn't guarantee the model will use them, but e.g. the skill content could say "I have a template in file X; use it if needed"). Copilot's documentation emphasizes that skills can include such additional files and that Copilot's *terminal tool* has safety mechanisms when executing scripts from skills (auto-approval lists) [121] [122] – implying that if a skill instructs Copilot to run a provided script, the user might need to allow it or pre-configure trust.

- **MCP (Model Context Protocol) Connectors**: Like Claude, Copilot implemented MCP support. In fact, *MCP is an open standard adopted by multiple AI coding tools* (Anthropic Claude, Cursor, VS Code, etc.). VS Code's MCP developer guide confirms that VS Code can interface with any MCP server offering tools/prompts [27] [28]. As a plugin author, one can create a separate MCP server (in Node, Python, etc.) that provides specialized functions. For example, an "npm assistant" MCP server might offer a `/searchPackage` tool for the AI, or even an *auto-complete function* for package names. To use it in VS Code, the user or an extension registers the server. Right now, enabling a custom MCP server in VS Code is a bit dev-focused (via settings or extension activation), but conceptually our portable plugin could include an MCP server component that runs universally. E.g. we package a small Python script that both Claude and Copilot can launch as needed. However, running arbitrary executables in VS Code might require an extension (to place the binaries and spawn them with the right args). VS Code supports local `stdio` MCP servers and even remote HTTP ones [28] [123]. Notably, VS Code can **reuse the user's Copilot API credentials to allow the MCP server to call the model** if needed (the "Sampling" feature) [32] – this could let an external tool itself ask the LLM for something (though that might not be needed if we're just bridging to other APIs).

- **VS Code Extension APIs**: For any gaps in high-level features, a custom VS Code extension can use lower-level APIs to simulate functionality. For example, if we want a "on file save, run tests" behavior (like a Claude PostToolUse hook for Write -> run tests), we could write an extension that catches VS Code's onDidSaveTextDocument event and then uses VS Code's Tasks or Terminal API to run tests, and then possibly informs Copilot of the results. Another example: if Claude's plugin used a PreToolUse hook to prevent deletion of files matching a pattern, in Copilot we might implement a similar policy by intercepting the delete command (maybe via a workspace file watcher or by analyzing Git changes) and warning the user. These kinds of things are more ad-hoc in Copilot's environment – we must implement them in code if needed.

Copilot vs Claude: Capability Matrix

To clarify how Claude Code plugins compare to Copilot's capabilities, here is a **feature matrix** contrasting each aspect and how the proposed portable spec will address it:

Feature / Capability	**Claude Code Plugins** (native)	**GitHub Copilot (VS Code)**	**Portable Plugin Spec** (design)
Packaging & Manifest	Single plugin directory with `plugin.json` manifest listing commands, agents, skills, hooks, etc [124] [125]. Install via marketplace or CLI. Namespaced commands prevent conflicts [126].	No single "plugin" container. Uses config files (`.github/`) and VS Code extensions for distribution. Marketplace (VS Code) can distribute extensions; .github files can be shared via git repos. No built-in namespacing except through participant names or command names.	**Proposal:** A *manifest YAML/JSON* that defines plugin metadata and components. This manifest can be used to generate Claude's `plugin.json` and to coordinate Copilot assets (for example, populating extension config or ensuring unique naming for slash commands). Namespacing: use plugin name as prefix for commands in Claude and perhaps as part of command names or participant name in Copilot to avoid collisions.
Slash Commands (user-triggered actions)	Markdown files under `commands/`. Execute within Claude's session, leveraging the AI to carry out instructions [42]. Can use placeholders for arguments [44]. Runs in context (not an external process).	**(a)** *Prompt Files:* `.prompt.md` files in `.github/prompts` become slash commands (inserting predefined prompt text) [17]. They can be parameterized via placeholders and `${input:}` variables for user input [127] [128]. **(b)** *Extension-provided commands:* An extension can contribute slash commands that run code or generate dynamic prompts [101] [102]. E.g. could run a script then insert output into chat. Copilot also has built-in slash commands (like `/tests`) as examples [96].	**Proposal:** Represent commands in the portable spec with a name, description, and either *static prompt text* or a reference to an action. For Copilot: simple static commands become prompt files (auto-generated `.prompt.md` with the text). Commands that need to run code (e.g. "/format: run prettier on current file") would be mapped to extension commands (the spec could mark them as requiring code execution). The adapter extension will register those accordingly.

Feature / Capability	**Claude Code Plugins** (native)	**GitHub Copilot (VS Code)**	**Portable Plugin Spec** (design)
Agents / Roles (multiple AI personas)	Subagents defined by markdown in `agents/`, with custom instructions and limited tools [46] [47]. Invoked via `@name` or tasks, running concurrently with main agent (as separate context). Allows multi-agent collaboration (one can call another) [50] [51].	**Custom Agents:** `.agent.md` files in `.github/agents/` define new agent profiles users can switch to [112] [14]. Only one agent is active at a time in chat (no simultaneous dialogs, but *handoff* allows sequential multi-agent flows [129] [130]). **Chat Participants (ext):** Extensions can add distinct participants `@xyz` that handle prompts with custom logic [108] [109]. In practice similar to custom agents but with extension control (for specialized domains or integrating external knowledge).	**Proposal:** Define agents in spec with a name, role description, and detailed persona prompt. Also specify tool access constraints and if it's primarily a *planner* or *executor*. For Claude: convert to agent markdown files. For Copilot: produce `.agent.md` files for each, preserving name and instructions. Include handoff definitions in frontmatter if the workflow requires (the spec can list which agent should suggest transitioning to which). If an automated multi-agent loop is needed (without user clicking handoff), we might implement it in the extension (the spec could mark an agent as "auto-run then switch to next"). But initial focus will be sequential flows with user approval (which is how Copilot handoffs work).
Auto Skills / Context	`skills/` directory with SKILL.md files (name & description triggers, plus content) [53] [55]. Claude auto-loads skill content into the conversation when relevant [24] [131]. Helps inject project-specific or reusable knowledge. Skills can include code examples or data (and possibly scripts if needed for tools).	**Agent Skills:** `.github/skills/<skill>/SKILL.md` files, same format (name, description, content) [22] [23]. Must enable setting for now. Copilot auto-discovers them and uses description matching to load them during chat [24] [131]. Extra files in skill folder can be referenced if AI chooses (and it can use a tool to open them).	**Proposal:** Skills can be defined identically in the portable spec (since format is essentially the same for both). The adapter can simply copy the skill folder into both plugin outputs (Claude and Copilot) with the same content. Ensure the skill `name` is unique across the ecosystem (the spec might prefix it with plugin name to avoid collision, though since skills are contextual, collisions are less likely). We'll adhere to the *agentskills.io* open standard [21] to maximize compatibility.

Hooks / Event Automation	Hooks in `hooks.json` allow intercepting events (before tool use, after tool use, on stop, etc.) and executing actions or model prompts [9] [10]. This can automate workflows (e.g. run formatter after file edit) or enforce policies (deny certain actions) [63] [64]. Very flexible within Claude's architecture.	No direct hook system exposed to end-users. Some *built-in* Copilot behaviors act like hooks (e.g. auto-fix on error in agent mode), but custom ones aren't declarative. **Alternative approaches:** Use VS Code extension event listeners (on file save, on commit, on terminal output, etc.) to trigger actions. Also, Copilot's presence in editor could be leveraged: for instance, an extension can detect when Copilot writes to a file and then perform an action. Or one can create a *background agent* (Copilot has "Background Agents" concept for running tasks continuously in the background [132]) – not well-documented for custom use yet.	**Proposal:** The portable spec can include hook definitions similar to Claude's (event + action). For Copilot, the adapter will implement what it can: e.g. a **FileSystem hook** (like on file save) can be done via an extension listening to the save event and either executing a command (like run tests) or prompting Copilot (invisibly or via a notification) to do something. **PreToolUse/ Permission hooks:** If the plugin wants to auto-approve certain Copilot actions (like running a trusted script), we might use Copilot's settings for auto-approving terminal commands (with allow-list) [121]. **PostToolUse hooks:** For example, after Copilot runs a terminal command, an extension can capture the terminal output and decide to prompt Copilot further (e.g. tests failed -> prompt to fix). This requires extension logic. *In feasibility terms, implementing hooks in Copilot will rely on the adapter extension and cannot cover every Claude event.* We will document which events can be reasonably mapped (file events, possibly command execution events via intercepting the terminal or tasks) and which are not possible (internal AI events like "Stop" or "SubagentStop" are not exposed – though one could approximate by monitoring chat messages). The spec's hook section will likely be partially supported on Copilot: simple automation (like "run X on save") – yes; complex AI decision hooks – maybe via extension prompting the model indirectly.

Feature / Capability	**Claude Code Plugins** (native)	**GitHub Copilot (VS Code)**	**Portable Plugin Spec** (design)
Tool Integrations	Rich built-in tool set: *Write/Edit* (edit files), *Read* (read file), *Grep*, *Glob* (search), *Bash* (shell), *WebFetch/Search*, etc. [133] [134]. Plugins can add tools via MCP servers (basically any external command/API) [74] [6]. Tools are invoked by the AI when needed (with optional user permission). Claude's UI will prompt if a plugin tool needs credentials or approval.	Similar array of **built-in tools** in VS Code's Copilot agent (not fully public, but implied: e.g. *read file*, *write file*, *open terminal & run command*, *search workspace*, *GitHub repo/issue search*, etc., plus *VS Code UI interactions* like opening a diff). Extensibility: Copilot can gain new tools through **MCP connectors** (via extensions or user config) [28]. Also, **extension-contributed tools** via the Language Model Tool API – essentially the same concept as MCP but tied to VS Code extension host. For example, one could provide a "DatabaseQuery" tool to Copilot. Tools can also be grouped into tool sets. In `.prompt.md` or `.agent.md` frontmatter, one can specify which tools are available for that prompt/agent [135] [136]. If a tool listed isn't present, it's ignored [137]. This provides control similar to Claude's tool permission in agent definitions.	**Proposal:** The portable spec should list any *custom tools* the plugin provides or needs. For instance, if the plugin needs an HTTP API tool, we define it abstractly. The Claude adapter would incorporate it via MCP (perhaps by generating an `.mcp.json` entry or requiring a certain Claude connector), and the Copilot adapter would either instruct users to install an existing extension or include an MCP server implementation as part of our extension. We might also categorize tools as "safe" (no user prompt needed) vs "privileged" (require user permission) and ensure Copilot's corresponding settings (or our extension's logic) reflect that (e.g. if plugin needs to run a shell command, the user must allow Copilot's terminal execution). In general, both platforms support adding arbitrary tools – Claude via MCP, Copilot via MCP – so our spec might include a **tool schema** (name, description, input/output spec). This could potentially allow future model-agnostic generation of OpenAI function definitions or similar as well.

Feature / Capability	**Claude Code Plugins** (native)	**GitHub Copilot (VS Code)**	**Portable Plugin Spec** (design)
State & Memory	Claude can use *persistent conversation memory* in the `.claude/CLAUDE.md` file (essentially user-provided context) and *project file context*. But beyond that, plugin data is mostly in the skill files or agent instructions. Plugins don't maintain long-term state except via files (e.g. a tool could write to a cache file). Claude subagents have separate context windows – effectively short-term memory for that agent only [49].	Copilot has *"custom instructions"* (like a persistent user profile) but for project-specific memory, the closest are *Skills* (which act as memory triggered by context) or possibly using hidden files. Copilot doesn't have a direct equivalent of CLAUDE.md. However, one can imagine using a skill or agent for memory (e.g. a "history skill" that logs important points and re-injects them). Also, since our plugin could run an extension process, we could store state in VS Code's global storage or the file system. For example, an extension could keep track of what was done in step 1 to guide step 2. But the model itself, each turn, largely depends on what's in the conversation context (which can include skill info and previous chat).	State management isn't a first-class part of spec, but we should note if any plugin feature needs to persist data (like an API token or a cache of analysis). For portability, we'd likely store such data in a **sidecar file** or use VS Code's Secret Storage for sensitive info. E.g. plugin manifest could declare "needs API key for X"; Claude would prompt and store it in its config, Copilot extension could prompt user for it and store in VS Code secure storage. As for conversational state, we lean on the AI's context (skills, etc.). Where deterministic state is needed (like which step of a workflow we are in), an orchestrator extension might have to manage that.

Feature / Capability	**Claude Code Plugins** (native)	**GitHub Copilot (VS Code)**	**Portable Plugin Spec** (design)
Security & Permissions	Claude enforces user confirmation for potentially risky actions (running code, internet access, etc.) [89] [90]. Admins can set organization policies limiting certain tools. Claude's plugin system sandboxes file access and doesn't allow network unless via tools that ask for permission. Secrets (API keys) can be provided via settings or environment and used by MCP connectors (usually manually configured).	Copilot in VS Code respects the IDE's security model: running a terminal command via Copilot will usually prompt the user (a notification "Allow Copilot to run this command?" with an allow-list possible) [121] [122]. VS Code also has *Workspace Trust* – if a workspace is untrusted, extensions (and presumably Copilot tools) have limited access. For network calls, if using an extension, we rely on Node's capabilities (with CORS or behind-proxy limitations if any). Enterprise Copilot may allow admins to disable certain features (like no terminal access). Visual Studio (the full IDE) likely has similar prompts.	**Proposal:** The portable plugin spec will include a **permissions section** or flags for sensitive actions. For instance, if the plugin might run system commands or access the internet, we mark that so users know and so that, say, the Copilot adapter extension can explicitly request the needed permissions (VS Code can show a dialog or require enabling a setting). We'll use least privilege: e.g. if a plugin doesn't need the web, we won't request it. For secrets, the spec can declare placeholders (like "API_KEY for ServiceX"). The Claude adapter could inject a skill or config for that, and the Copilot adapter could integrate with VS Code's `settings.json` or secret storage. Documenting these requirements will be part of the plugin package (so that enterprise users can vet what a plugin does). Also, logging and telemetry: our extension can log plugin tool usage or errors in an output channel so users have traceability (Claude provides some logging for plugin actions by default in its UI).

Feature / Capability	**Claude Code Plugins** (native)	**GitHub Copilot (VS Code)**	**Portable Plugin Spec** (design)
Distribution & Installation	Distributed via **Claude marketplaces** (git repos with an index). Users install by `/plugin install repo/name` and it lands in their `~/.claude/plugins` cache [86]. Versioning is via semantic versions in manifest [138]. Alternatively, a user can manually load a plugin from a path (useful for local dev) [84]. Sharing within a team can be done by committing the plugin directory in a repo and adding that repo as a marketplace.	Distribution can be via **Visual Studio Code Marketplace** if we create an extension. That's the formal route for broad sharing (including auto-updates). Alternatively, sharing a plugin for Copilot could mean instructing users to copy a set of files into their repository (the `.github/` folder approach) – this is lightweight but not centralized (each repo needs the files, or one could script copying them). GitHub could potentially allow org-level configuration (there was mention of organization-level custom agents in GitHub settings). For private/internal use, an extension can also be side-loaded or distributed as a VSIX.	We propose a **single repository** that contains the portable plugin spec and perhaps a CLI tool to deploy it to various targets. For example, `npm run deploy-claude` could package the Claude plugin (possibly publishing to a marketplace repo or outputting a zip), and `npm run deploy-copilot-local` could install the VS Code extension or copy files to the current project's `.github` folder. Initially, simplest is to use the file-based approach for Copilot (no installation beyond copying files, which could even be done by a script command). Long-term, if multiple projects should use the plugin, publishing an extension is cleaner – we will also outline how to structure the extension code in the same repo (likely in an `adapter-copilot/` subfolder). Versioning: we keep a version in the manifest; for Copilot's file approach, updates mean updating the files in each repo (perhaps a tool can sync them). For an extension, we follow VS Code extension versioning.

(Table: Feature comparison between Claude Code plugins, GitHub Copilot capabilities, and how the portable spec will handle each.)

Gaps and Feasibility Notes

From the matrix, the main **gaps** where Copilot cannot fully replicate Claude yet are: **event hooks** (no direct support, must use extension code) and **truly concurrent multi-agent processing** (Copilot won't run two chat agents in parallel on its own). Everything else – slash commands, custom agents, skills, external tools – Copilot supports in some form, though sometimes requiring more effort (extensions).

For multi-agent workflows in Copilot, the recommended approach is to use **sequential agents with handoffs** (the user or extension triggers transitions). For instance, to emulate Claude's Planner->Coder->Tester loop: one could have three custom agents and use either user-triggered handoffs (the chat UI shows a button "Hand off to Tester") [129] [139] or have the extension auto-execute the chain (with or without confirmation at each step). The extension route could be complex to implement reliably, so we might start with the user clicking through handoff suggestions, which already achieves a similar result with minimal custom code.

Another limitation: **Visual Studio 2022/2024** (as opposed to VS Code) has Copilot integration, but it may not yet support all these customizations (especially not the file-based ones, since .github files are more likely picked up by the GitHub Copilot service which is IDE-agnostic). According to GitHub documentation, custom prompt files and agents *do* work in Visual Studio as well [140] [141], stored in a `.github` folder in the solution or repository. We should verify that, but assuming it works, our file-based approach should carry over to VS. The extension approach would differ since VS uses VSIX extensions, but Microsoft might unify Copilot's plugin model. We'll focus primarily on VS Code (since it's the more open platform), with the assumption that the same .github-based assets will function in VS (and if not, a Visual Studio extension might be needed down the line).

Portable Plugin Specification Design

To achieve portability, we propose a **Portable Plugin Specification** that describes all the necessary pieces in a single, technology-agnostic way. This spec will be used by adapters/generators to produce the Claude-specific and Copilot-specific implementations. The spec could be a YAML or JSON file (for human-editable clarity, YAML is friendlier) alongside the accompanying resource files (like markdown content for agents or skills, script files for tools, etc.).

Key elements to include in the spec:

1. **Metadata:** Plugin name, version, description, author, etc. (Much like Claude's plugin.json or an extension manifest). This helps identify the plugin across systems. For instance:

```
name: my-coding-assistant
version: 1.0.0
description: "Assistant that helps plan, implement, and test features"
author: "Acme Corp"
license: MIT
```

2. **Agents (Roles):** Define each agent's persona and permissions. Fields:

3. `id` (identifier/name, e.g. "planner", "implementer"),
4. `description` (when to use it / short info, could serve as Claude subagent description and Copilot detection hints),
5. `instructions` (the full markdown content that defines the agent's behavior; could either be inlined here or point to a `.md` file in the repo for ease of editing),

6. `model` (optional model preference, e.g. use faster model for planner),
7. `tools` (optional list of tool names or categories this agent is allowed to use).
8. `handoffs` (if this agent should suggest moving to another agent after it's done, define target agent and prompt). For example:

```
agents:
  - id: planner
    description: "Generates implementation plans for new features"
    instructions: |
      You are an expert planning assistant...
      (full prompt content)
    tools: [ "read", "search", "terminal/run:test" ]
    handoffs:
      - agent: coder
        label: "Implement Plan"
        prompt: "Please implement the above plan."
        autoSend: false
  - id: coder
    description: "Writes code to fulfill a plan, iterating until tests
pass"
    instructions: |
      You are a coding assistant...
    tools: [ "write", "edit", "terminal/run", "tests" ]
    handoffs:
      - agent: tester
        label: "Review Code"
        prompt: "Review the code for any issues."
  - id: tester
    description: "Reviews code and suggests improvements or confirms
quality"
    instructions: |
      You are a senior engineer reviewing the code...
    tools: [ "read", "grep", "tests" ]
```

(This example defines a three-agent workflow with handoffs. `autoSend: false` *means in Copilot the user must hit send after switching, whereas in Claude this might correspond to just returning control to main agent.)*

These agent definitions will be used to generate Claude's `agents/*.md` files (with YAML frontmatter containing name, description, model, color perhaps) and Copilot's `.agent.md` files (with similar frontmatter). The `handoffs` in spec can translate to Claude hook or just instructions, and to Copilot YAML `handoffs` in the agent's file [130] [142]. (Note: Claude doesn't have an explicit "handoff" feature; one usually implements a handoff by having the agent output a special trigger that the main agent recognizes via a hook or the user manually switches. We might skip formal Claude handoffs and rely on user commands to switch.)

1. **Commands:** Define slash commands that user can invoke:

2. `name`: the command keyword (without plugin namespace; the plugin name will be prefixed for Claude, and used as part of slash name in Copilot if needed).
3. `description`: help text.
4. `action`: This could be one of:
    - a *prompt string* (i.e., content to insert into chat, possibly with placeholders for arguments),
    - an *agent invocation* (e.g. "invoke planner agent with given user request" – though the user could just call `@planner`, having a slash command for it might be convenience),
    - an *external action* (like run a shell command).
5. `argsSchema` (optional): to define if it takes arguments or not, maybe for documentation or future UI generation.

Example:

```
commands:
  - name: plan-feature
    description: "Plan the implementation of the specified feature"
    action: "Use the planner agent to create a plan for `$ARGUMENTS`."
  - name: run-tests
    description: "Run the test suite"
    action: "!npm test"   # '!' denotes this is a shell command to run
  - name: greet
    description: "Just greet the user"
    action: |
      Hello, `$ARGUMENTS`! How can I assist you today?
```

Here, `/myplugin:plan-feature Login system` would effectively prompt the Planner agent to plan "Login system" (the adapter might implement this by switching to planner agent with that query). `/myplugin:run-tests` would execute `npm test` in the project (Claude can do that via Bash tool; Copilot via terminal tool or extension). `/myplugin:greet Alice` would output a greeting.

The adapter logic: - For static prompt commands (like greet), create a `commands/greet.md` for Claude with content and do a `.prompt.md` for Copilot with similar content. - For "agent invocation" commands, we could implement them as just a prompt that says "Switch to X agent and do Y" or directly instruct main agent to delegate – but better might be for Copilot to just serve as a shortcut: e.g. a prompt file that sets `agent: planner` and uses `$ARGUMENTS` in the query. - For external actions (`!npm test` convention above), Claude: we can set up a hook or use the MCP "Task" to run it. Actually, simplest: define a Claude command that says: "Run the tests (shell command `npm test`) and report results." That will cause Claude to likely use its Bash tool. In Copilot, we might implement this command via an extension (because Copilot might not automatically run `!npm test` from a prompt – although Copilot CLI allowed `!` prefix to run shell in some contexts [143]). Probably safer: extension catches `/run-tests` and executes it, then maybe triggers Copilot to summarize results. - So, some commands may require our VS Code extension to implement on Copilot side. The spec can mark them accordingly (e.g. a `shell: true` flag or the bang notation as above).

We will generate documentation for each command (for the user, and for insertion into Copilot's slash command list if needed). In Claude, the commands auto-list under `/help` with their descriptions.

1. **Skills:** Define as needed, likely by referencing skill directories. We can either include them inline or as separate files:

   ```
   skills:
     - id: secure-coding
       description: "Security best practices for web apps. Auto-invoke when
   code dealing with auth or crypto is present."
       content: |
         Guidelines for secure coding:
         1. Validate all inputs...
         2. ...
     - id: performance-tips
       path: "./skills/performance"  # maybe the spec can just point to an
   existing folder with SKILL.md
   ```

   We allow either inline `content` or a `path` to a skill folder (to accommodate large skill texts or those with resources). The adapter will ensure the `SKILL.md` is present and formatted for each environment. Likely the spec `content` would translate directly to a SKILL.md file with frontmatter containing name/description.

2. **Hooks:** This is tricky due to platform differences. We propose an abstraction for a few common scenarios:

3. **File Events:** e.g. on save of certain file types, on open, on close.
4. **Git Events:** on commit, on PR creation (these might be out-of-scope for the coding assistant itself, but an extension could catch them).
5. **Tool invocation events:** e.g. after a command or tool is used. (Claude's Pre/PostToolUse map to Copilot's internal events not accessible; but we could simulate *PostToolUse for specific tools* by monitoring outputs, like test results.)
6. **Conversation events:** e.g. after the assistant answers (Claude's "Stop"). Copilot doesn't allow injection after answer without user input, so probably skip.
7. **Periodic or Idle events:** possibly not needed or could be an extension timer.

We might limit initial support to *File Saved* and *After Tests Run*. The spec might look like:

```
hooks:
  - event: onFileSave
    glob: "*.js"
    actions:
      - type: command
        ref: run-tests  # refer to a defined command or tool
  - event: afterTool
```

```
    tool: "tests"  # after tests tool runs
    actions:
      - type: prompt
        content: |
          Analyze the test results and report any failures in detail.
  - event: onAgentComplete
    agent: planner
    actions:
      - type: autoHandoff
        targetAgent: coder
        prompt: "Implement the plan above."
```

This is illustrative: *onFileSave* for JS files triggers the run-tests command (which we defined above). In Claude, this would be done via a hooks.json PostToolUse on Write events matching *.js that calls* `npm test` [34]*. In Copilot, our extension can listen for save and run tests.* afterTool 'tests' *means when tests finish running, prompt the AI to analyze results – in Claude, possible via PostToolUse on the "Task" or "Bash" if it matches a test command; in Copilot, we can intercept the terminal output and then use the Copilot API to send a message like "Test X failed at line Y".* onAgentComplete (planner) *with autoHandoff basically automates the transition to coder agent with the given prompt – in Claude we might implement via a Stop hook that detects if it was planner and then have Claude ask the next step (though that likely needs user to confirm anyway), in Copilot an extension could catch that the planner finished (maybe by looking at chat turns, not trivial) or we just rely on the* handoff suggestion* mechanism instead (so maybe we don't automate it fully).

Given complexity, we'll document that not all hooks will be auto-fulfilled on Copilot; some may degrade to a manual step or require clicking a suggestion.

1. **Tools/Connectors:** If the plugin needs a custom tool that is not readily available in both:

2. We can define it abstractly, e.g.:

   ```
   tools:
     - name: browserAutomation
       description: "Open a headless browser and perform actions"
       requiresMCP: true
       script: "./servers/browser.py"
       api: "http"  # use HTTP or stdio
     - name: weatherApi
       description: "Fetch weather via API"
       functionSpec: ...  # if in future we define an OpenAI function for
   chatGPT
   ```

   This area could become complex. If such tools exist, it might be better to rely on existing ones (e.g. maybe use the `webbrowser` MCP connector by Anthropic, or in Copilot environment, an extension like `wakatime` if it existed). For our blueprint, we can assume the plugin's core tools are the built-in ones plus maybe a small one we implement ourselves as proof of concept (like a dummy "Hello World" MCP server).

The adapter for Claude would generate entries in `.mcp.json` to start these servers [73] [74]. For Copilot, our VS Code extension could launch the same server (if local) and register it via VS Code's API (there are commands to connect an MCP server in development mode). Alternatively, implement the functionality directly in the extension if simpler.

3. **Policies/Permissions:** A section to enumerate what the plugin is allowed to do or not:

```
permissions:
  filesystem: read-write   # or 'read-only' or 'none' outside workspace
  network: external-api    # or 'none'
  requireConfirmation: ["shell", "externalApi"]  # types of actions that
should always ask user
```

   This can guide the runtime: Claude might ignore this (Anthropic's own policies apply), but Copilot extension can use it to set appropriate approvals. For instance, if `filesystem: read-only`, the extension could ensure not to run any write operations without asking. Or if network usage is present, maybe prompt user to configure an API key.

4. **Observability (Logging/Testing):** Not core to spec, but we might include an optional `debug: true` flag to enable verbose logging by adapters. Also, perhaps a `testCases` section that defines some input/expected output scenarios for the plugin (helpful for verifying portability). For example, a test case could be: "User triggers /greet command with name=Alex, expected assistant response contains 'Hello, Alex'." This can be used to validate both in Claude and Copilot yield similar results (given nondeterminism of LLMs, we keep expectations broad, like a contains check).

The **manifest file format** might look like this put together (in YAML for readability):

```
name: my-coding-assistant
version: 0.1.0
description: "Helps plan, implement, and test features with multiple AI agents."
author:
  name: Acme Corp
  email: [email protected]
  url: https://github.com/acme/my-coding-assistant
license: MIT

agents:
  - id: planner
    description: "Planning agent for features (generates plans, no code
editing)."
    instructions: ./agents/planner.md   # external file with prompt content
    model: "Claude Instant"
# preference; Copilot will map to GPT-3.5 perhaps
    tools: ["read", "search"]
```

```
    handoffs:
      - agent: coder
        label: "Start Implementation"
        prompt: "Begin implementing the above plan."
  - id: coder
    description: "Implementation agent (writes code, runs tests)."
    instructions: ./agents/coder.md
    model: "Claude 2"
    tools: ["write", "edit", "terminal", "tests"]   # terminal implies can run
shell
    handoffs:
      - agent: tester
        label: "Review Code"
        prompt: "Review the code and verify it meets the requirements."
  - id: tester
    description: "QA agent (reviews code and ensures quality)."
    instructions: |
      You are a testing expert...
      (inline content can be provided)
    tools: ["read", "grep", "tests"]

commands:
  - name: plan
    description: "Plan the implementation of a given feature."
    action: "Use the planner agent to create an implementation plan for
`$ARGUMENTS`."
  - name: run-tests
    description: "Run all tests."
    action: "!npm test"   # run shell command
  - name: hello
    description: "Greet the user."
    action: "Hello, $ARGUMENTS! This is your AI assistant."

skills:
  - id: security-guidelines
    description: "Security best practices for web development. Auto-use when
discussing auth, encryption, or validations."
    content: |
      **Secure Coding Guidelines:**
      - Always validate user input on server-side...
      - Use parameterized queries to prevent SQL injection...
  - id: performance-guide
    description: "Performance optimization tips. Use when code efficiency or
scaling is mentioned."
    path: "./skills/performance-guide"  # separate folder containing SKILL.md
and maybe resources

hooks:
```

```
  - event: onFileSave
    glob: "*.js"
    actions:
      - type: command
        ref: run-tests
  - event: afterTool
    tool: "tests"
    actions:
      - type: prompt
        content: "All tests completed. If any test failed, analyze the failure
and suggest fixes."
  # (In Copilot, the extension will catch test output; in Claude, we rely on
Claude to automatically show errors anyway and perhaps a Stop hook could trigger
analysis.)

tools:
  - name: httpRequester
    description: "Make HTTP GET requests to retrieve data from URLs."
    implementation:
      type: mcp
      program: "python3"
      args: ["tools/http_requester.py", "--stdio"]
      # This would be a simple MCP server script included in plugin files.

permissions:
  filesystem: write
  network: none
  requireConfirmation: ["shell"]
```

The **adapters** will use this spec as follows:

- **Claude Adapter:** Reads the spec and generates the `plugin.json`. Most fields map directly:
- `name`, `version`, `description`, `author` -> same.
- It will list additional component paths if needed (e.g. if we put agent files not in default location, but we likely will use defaults so not needed).
- Create `commands/` directory and for each command:
  - If action is a prompt, create `name.md` with frontmatter (description) and body from spec (substituting `$ARGUMENTS` usage as needed) [144].
  - If action is `!shell`, we have choices: could create a command file that instructs Claude "Run the following shell command and return output: ...". Or use a hook instead to auto-run it. However, since user explicitly invoked it, simpler is markdown that says "Use the Bash tool to run `npm test`." Claude will then do it (with permission). We should also add something to capture output, but Claude will usually show output of tools by default. So a static prompt might suffice.
  - Alternatively, utilize `.hooks.json` to automatically run after save (we already have that as hook).

- Create `agents/` markdown files: for each agent, YAML frontmatter with `name` (for Claude, the `name` frontmatter likely is the invocation name), `description`, maybe `model` (Claude supports specifying model in subagent file as shown in Shipyard example [145]), and `tools` if possible (the Shipyard snippet hinted at restricting tools optionally [47]). If not supported in frontmatter, we can still embed a note in instructions telling agent not to use certain tools.
- Put the agent's instructions content under the frontmatter. Include example dialogues if needed to guide usage (Claude often includes `<example>` blocks in agent definitions for context switching triggers [46] [146]).
- Create `skills/` directories with each skill's SKILL.md. Use name/description from spec. The body from spec as well.

- Assemble `hooks.json`: map spec events to Claude's events. `onFileSave` is not a Claude hook, but we can approximate it with PostToolUse on Write events for those files (as seen in Claude migration example [34]). So we'd add:

```
"PostToolUse": [
  { "matcher": "Write",
    "hooks": [
      { "type": "command", "command": "npm test", "when": { "matchesFile":
"*.js" } }
    ]
  }
]
```

(Pseudo-syntax: either split by file type or just run for any Write and inside command perhaps use $FILE to filter. The example in docs used a matcher with regex "Write|Edit" and then an action with $FILE [34]. So we would do that.) For `afterTool: tests`, that equates to PostToolUse on the Bash or Task that ran tests. If our `run-tests` command uses Bash, then PostToolUse on Bash with matcher maybe "npm test" could trigger a prompt hook. We might need to get fancy (Claude hooks allow a prompt hook that gets the tool output as $ARGUMENTS [147] [148]). We can implement:

```
"PostToolUse": [
  {
    "matcher": "Bash",
    "hooks": [{
       "type": "prompt",
       "prompt": "Tool output:\n$ARGUMENTS\n\nIf any tests failed, explain
and suggest fixes."
     }]
  }
]
```

That would have Claude analyze any bash output (which includes test results). The planner->coder handoff we spec'd as `onAgentComplete(planner) autoHandoff->coder`. Claude has no direct event for subagent done except `SubagentStop` which triggers when any subagent finishes [149].

We could use that: in `SubagentStop` hook, if the subagent name == planner, then run a command or prompt that engages coder. Possibly just have Claude reply with something like: "Switching to implementation..." which might prompt user to call coder. But we might skip automated handoff in Claude and let the user invoke coder after seeing plan (which is the normal approach).

- Prepare `.mcp.json` if any tool requires a server. In our spec example, we listed `httpRequester` with a script. We would add:

```
{
  "servers": {
     "httpRequester": {
        "command": "python3",
        "args": ["tools/http_requester.py", "--stdio"],
        "cwd": "${CLAUDE_PLUGIN_ROOT}"
     }
  }
}
```

And ensure that tool's functionality is documented so the AI knows how to use it (maybe via skill or the agent prompt).

- The manifest `plugin.json` would reference hooks, possibly mcp config if not separate:

```
{
  "name": "my-coding-assistant",
  "version": "0.1.0",
  "description": "...",
  "commands": "./commands/",
  "agents": "./agents/",
  "skills": "./skills/",
  "hooks": "./hooks/hooks.json",
  "mcpServers": "./.mcp.json"
}
```

(Claude can also inline hooks in plugin.json, but we'll use a file.)

This yields a Claude plugin ready to test with `claude --plugin-dir`.

- **Copilot Adapter:** There are two sub-adapters:
- **File-based (no-code)**: This will generate or update the following in the user's project:
  - `.github/prompts/` for each slash command (except those marked as external actions). Each `name.prompt.md` will have YAML:

```
---
name: plan-feature   # if plugin name is my-coding-assistant, maybe
combine: "my-coding-assistant: plan-feature" to avoid collisions in
slash menu, but Copilot might not allow colon in name. Could use
"myplugin-plan" prefix.
description: Plan the implementation of a given feature.
agent: planner       # so that this prompt uses the planner agent
context
argument-hint: "[feature description]"  # guidance
tools: []            # could specify if we want to restrict or allow
certain tools for this prompt. If agent is set, it might use agent's
default tools unless overridden.
---
Use the planner agent to create an implementation plan for
"$ARGUMENTS".
```

For simple ones like greet, we might not set an agent (so it uses current agent, likely default). For `run-tests` (external action): we probably **should not create a prompt file** because typing `/run-tests` in chat with a prompt file could tell Copilot "run tests" but Copilot itself cannot directly run them unless we allow it tool access and it chooses to. Actually, we could attempt:

```
name: run-tests
description: Run all tests
agent: agent   # use default coding agent
tools: ["terminal"]   # allow terminal tool
---
Run the project's test suite and report the results.
```

This *might* cause Copilot to use its terminal integration to run `npm test` (if it correctly infers the command). The phrase "Run the project's test suite" could lead the AI to decide to use the terminal, type `npm test`, etc. Copilot's agent might ask permission to run it. This could work and then it would output results in chat. It's not guaranteed, but likely given training it might know common build/test commands. So we can try making even shell commands into AI instructions. However, this is less deterministic than doing it via extension code. Alternatively, we **supplement** this by having our extension register a slash command that directly runs tests on `/run-tests`. If the user has the extension installed, it will intercept and run tests immediately (maybe providing output in a chat message). If they don't, the fallback is the prompt approach where the AI tries to run tests. This dual approach can be considered.

- `.github/agents/` for each agent: e.g. `planner.agent.md` with:

```
---
name: planner
description: Planning agent for features (generates plans, no direct
```

```
code edits).
tools: [ "read", "search" ]
model: GPT-4
---
(content from planner.md instructions)
```

We also add any `handoffs:` as per spec – VS Code's format for detection might need adding in `disambiguation` if we want auto-route questions to this agent (we can include some example triggers as in docs example [150] [151], but optional). The handoff suggestions from planner to coder: in the planner's file frontmatter we add:

```
handoffs:
  - agent: coder
    label: Start Implementation
    prompt: Now begin implementing the plan above.
```

This should make Copilot Chat show a "Start Implementation" button after planner responds [152].

- `.github/skills/` for each skill: basically copy the skill content. The same SKILL.md format works [22] [23].
- These files, once committed or placed in the repo, should be picked up by Copilot Chat automatically (assuming Copilot is running with those features on). The user would then see in the chat UI: additional slash commands (our prompt commands) in the autocomplete menu, and additional agents in the agents menu (our custom agents).
- The user would manually switch to an agent if desired, or run slash commands that internally use those agents. Hooks are not handled by this approach; at best, our prompt or agent instructions could mention "(This should be done whenever X happens)" but that's just guidance, not actual automation.

This file-based approach is great for initial feasibility tests: we can ask the user to copy the generated `.github` folder into a sample project and see if Copilot behaves as expected (for example, try `Copilot: Chat -> /plan Feature X` and see if it uses the planner persona, then maybe click *Start Implementation* suggestion to invoke coder, etc.).

One limitation: **Visual Studio** – it also supports custom prompts/agents but uses a different location (likely the same `.github` folder if the solution is in a git repo). If not, possibly something like a settings in VisualStudio. But we will assume using a git-backed repo.

- **VS Code Extension (Copilot Adapter)**: For full functionality (hooks, guaranteed execution of certain commands, etc.), we develop a VS Code extension:

  - It would likely activate on opening a workspace that has our plugin (maybe indicated by presence of the manifest or by user installing it explicitly).
  - It can read the portable spec (either the YAML if present in workspace, or we might bake the plugin logic into the extension ahead of time for distribution).
  - It can then do things such as:

- Registering any **chat slash commands** that need dynamic behavior. In VS Code's API, we can register a Chat kind "service message" or simply use the **Conversation API** to intercept the slash command text. Actually, a simpler approach: define a **Chat Participant** for the plugin that can intercept certain inputs. But that might be overkill.
- Possibly easier: Use VS Code **Commands** and **Quick Pick**: but those wouldn't appear in chat UI's slash list. Alternatively, the extension can contribute to `contributes.chatParticipants.commands` as earlier where we saw in package.json example for `cat` participant [102] [153]. If we go that route, we might set up a hidden participant or piggyback on existing `@workspace` participant (not sure if possible).
- However, since we have already placed prompt files for commands, the extension might not need to add slash commands – they're already there. Instead, it might **override** their behavior by listening to chat messages. For instance, our extension could listen on the Copilot chat output (not sure if an API exists to get the conversation events – possibly not publicly). Another approach: if our slash command triggers an extension *side effect*, we could have the slash command's prompt text include a distinctive marker that our extension looks for. Example: user types `/run-tests`, which inserts "Run the project's test suite now. [ext:run-tests]" or some hidden marker. The extension could detect that prompt (maybe by reading the document that is the chat input? Or via a pre-send hook if provided). VS Code's API might allow a *Chat request provider* where we can intercept the prompt (the Chat Participant API hints at orchestrating the request ourselves).
- Alternatively, simpler: scrap that – just have a separate VS Code command (in command palette or context menu) "Run plugin: run-tests" that the user can trigger. That's not as smooth as slash, but is an option. Or have the extension automatically run tests on file save (that covers one hook).

- For hooks: *File save hook*: use `workspace.onDidSaveTextDocument` and if file matches glob, then do something. In our example, on saving a `.js` file, extension runs `npm test`. We should surface output (maybe in VS Code terminal or output channel). We can optionally send a message to Copilot chat like "Tests are running..." or simply rely on test output and then the user can ask Copilot if needed. *After tests hook*: extension can monitor the terminal where tests run. Once done, it can fetch results. If failures, it could automatically invoke a Copilot chat query like "Some tests failed, please analyze" (programmatically send it to chat). The VS Code API *LanguageModelChatProvider* might allow an extension to send a message to the chat on behalf of user or system. If not, we can at least pop a notification saying "Tests failed. Click to ask Copilot for help." That click could insert a question to Copilot chat. So lots of possibilities, albeit requiring code.

- If we want to coordinate multi-agent steps automatically (e.g. after planner agent finishes, auto-switch to coder agent), we could monitor chat history. There is an API to get the `vscode.chat.sessions` and their messages (for example, for Chat Participants implementation, they provide context to the extension) [154] [155]. Perhaps we can see when a planner agent finishes by looking at last response metadata. However, since Copilot natively supports *handoff suggestions*, maybe we don't need to automate – the user can click.

- **MCP server integration:** If plugin defines a custom MCP, our extension can spawn it. VS Code's MCP dev guide suggests using a command to connect a server for debugging. Possibly we can spawn and register via `vscode.ai.connectToMcpServer()` API (if exists

in the Language Model Tool API). If not, since it's standard, maybe adding an entry in `settings.json` under `copilot.experimental.mcp` might work. But likely there is an extension method. We'll have to consult the VS Code extension API references (the existence of `MCP Dev Guide` implies methods or at least guidance). If complicated, for initial prototype, we might skip custom MCP tools and rely on built-in ones to reduce moving parts.

Essentially, the extension will handle all the heavy-lifting that .github files cannot do: - Implementing automation (hooks), - Ensuring certain commands are executed reliably, - Possibly providing UI or telemetry (like output panels to show logs of plugin actions, which enterprise users might want to audit). - It also serves as a *packaging mechanism* – instead of copying .github files to every repo, an extension (once installed globally in VS Code) can inject the needed participants or commands in any workspace. For example, an extension can register a *Workspace Context Provider* that feeds context to Copilot (similar to what @workspace participant does with code context). But now that user-level skills/agents are available, the .github approach covers multi-repo usage by having user put those files in a VS Code Profile (`~/.copilot/agents/` etc, which is possible for user-level customizations [119] [156]). The spec could even be installed at user level.

We anticipate the **portable plugin spec** to be maintained by the developer in one place, with perhaps markdown files in a subfolder for lengthy texts (agents instructions, etc.). We will provide tooling (maybe a small CLI script or just documentation) on how to *deploy* the plugin: - To test in Claude: run `claude --plugin-dir .` after generating the Claude plugin. - To test in Copilot (file mode): copy the generated `.github` folder into a sample repo (or your current one) and open Copilot chat. - To package for distribution: build the VS Code extension (which bundles perhaps the .github files or generates needed contributions at runtime).

"Hello World" Portable Plugin Example

As a concrete example, let's outline a simple "Hello World" plugin that can run in both Claude Code and Copilot:

Plugin Name: portable-hello

Functionality: - Adds a slash command `/hello-world` that greets the user. - Introduces two agents: *Greeter* (says hello in various styles) and *Reviewer* (just a dummy agent that "reviews" the greeting for friendliness). - A skill that contains a fun fact (maybe auto-invoked if user mentions "fun fact"). - A hook that after using the greeter agent, automatically switches to the reviewer agent.

Spec (summarized):

```
name: portable-hello
version: 0.1.0
description: "A cross-copilot hello world plugin."
agents:
  - id: greeter
    description: "Enthusiastic greeting agent."
```

```
    instructions: |
      You are a friendly greeter bot. Always greet with enthusiasm and maybe an
emoji.
    handoffs:
      - agent: reviewer
        label: "Review Greeting"
        prompt: "Review if the greeting is friendly enough."
  - id: reviewer
    description: "Politeness reviewer agent."
    instructions: |
      You analyze messages for politeness and friendliness.
commands:
  - name: hello-world
    description: "Greets the user enthusiastically."
    action: "Hello, world!   I'm your coding assistant."
  - name: agent-greet
    description: "Ask the Greeter agent to greet someone by name."
    action: "Use the greeter agent to greet `$ARGUMENTS`."
skills:
  - id: fun-fact
    description: "Fun fact about Hello World. Auto-load when user asks for fun
fact."
    content: "FunFact: The first known instance of \"Hello, World!\" in a
programming context was in the 1972 Kernighan and Ritchie C tutorial."
hooks:
  - event: onAgentComplete
    agent: greeter
    actions:
      - type: autoHandoff
        targetAgent: reviewer
        prompt: "Please review the above greeting."
```

Claude side: This becomes a plugin with: - `commands/hello-world.md` (simple static greeting), `commands/agent-greet.md` (which might instruct Claude "Use @greeter to greet $ARGUMENTS" or just "As Greeter: greet $ARGUMENTS" – since Claude doesn't have a built-in concept to call a subagent from a command, the command might just directly produce a greeting, or it could explicitly yield a plan like: it could respond with something that triggers the greeter. Possibly easier: we make `/agent-greet` simply output: "@greeter, $ARGUMENTS" – which Claude will interpret as the user invoking the greeter agent. Not sure if that works in one turn; if not, user might have to copy it. Alternatively, we skip the command and let user directly invoke @greeter.) - `agents/greeter.md`, `agents/reviewer.md`. - `skills/fun-fact/SKILL.md`. - `hooks/hooks.json` with SubagentStop or Stop hook to handle greeter -> reviewer. (Claude could use `SubagentStop` event with matcher for greeter's name to then run reviewer automatically, but that may involve calling the reviewer agent autonomously, which Claude might not support without user input – it can with hooks: a SubagentStop *prompt* hook could ask "Should I switch to reviewer?" but that's again a model decision. Possibly we can have the hook of type command that just outputs something like triggering reviewer, but there's no command to swap agent except user context. So maybe skip actual auto-

run in Claude, just prompt user "You can now use reviewer." In any case, this part isn't straightforward.) - However, if user triggered via slash command or manually, it's fine.

Copilot side: - `.github/prompts/hello-world.prompt.md` (with the greeting text; trivial). - `.github/prompts/agent-greet.prompt.md` (set `agent: greeter` in YAML and body "Greet ${input:name} enthusiastically." – then when user uses it, Copilot will activate greeter agent context just for that prompt, hopefully). - `.github/agents/greeter.agent.md`, `.github/agents/reviewer.agent.md` (with content, plus in greeter's YAML we put a handoff to reviewer as specified). - `.github/skills/fun-fact/SKILL.md`. - Hooks: on Copilot, we can rely on the handoff suggestion to switch to reviewer after greeter responds (which is built-in from YAML). We don't have auto execution (unless we code extension). We might not implement an extension at all for "Hello World" since it's not needed (no external actions or real events).

Testing: - In Claude: `/portable-hello:hello-world` yields a greeting. `/portable-hello:agent-greet Alice` hopefully triggers the greeter persona or at least a greeting containing "Alice" with emoji. Or maybe user would do `@greeter, please greet Alice` (the command could serve just as an example). - In Copilot: In chat, type `/hello-world` (should output greeting). Type `/agent-greet Bob` (should run using greeter agent context, giving an enthusiastic greeting as the greeter's persona directs). After that, Copilot should show a suggestion "Review Greeting" as a follow-up (from greeter's handoff), and clicking that would switch to reviewer agent and send the "Please review the above greeting." prompt, resulting in a review message.

This demonstrates multi-agent workflow and skill usage in a minimal way.

Adapter Architecture & Repo Layout

We propose structuring the repository (and runtime) roughly as follows:

```
portable-plugin-project/
├── portable-plugin.yaml        # The manifest/spec as defined above.
├── agents/                     # Agent instruction files (optional, can embed
in YAML too).
│   ├── agent1.md
│   └── agent2.md
├── skills/                     # Skill content (if large or with resources).
│   ├── skillA/
│   │   ├── SKILL.md
│   │   └── ... (maybe example files)
│   └── skillB/...
├── scripts/                    # Any scripts or MCP server implementations.
│   └── http_requester.py
├── adapters/
│   ├── claude/
│   │   └── build_claude_plugin.py   # or a script to generate plugin dir
│   ├── copilot-files/
│   │   └── build_github_files.py   # generates .github/ structure
```

```
│   └── copilot-extension/
│       ├── package.json       # VSCode extension manifest, referencing chat
commands etc.
│       ├── src/               # extension TypeScript source code
│       └── ...
└── dist/
    ├── my-coding-assistant-claude.zip   # output for Claude plugin
    ├── portable-hello.vsix              # packaged VSCode extension (if any)
    └── ...
```

In this layout: - The developer mainly edits `portable-plugin.yaml` plus any separate content files (agents, skills, etc.). We keep them in a structured way for clarity. - We provide scripts under `adapters/` to transform the spec: - `build_claude_plugin.py`: reads YAML, writes out to `dist/claude-plugin/<pluginName>/...` all the needed files (or directly zip it). Possibly we implement this in a simple Python or Node script. Alternatively, we can integrate it as part of the VS Code extension build, but having separate is fine. - `build_github_files.py`: creates a `dist/github-files/` output with the `.github/` folder structure containing prompts, agents, skills, etc. The user can copy this to a repo to test. (We could even include a script that automatically copies into the current repo if run, but better to leave manual or clearly instruct the user.) - The `copilot-extension/` will be a full VS Code extension project if we decide to make one. It will likely depend on the YAML to know what to do (we could have it load the YAML at runtime from the workspace or include a compiled version of it). There's a design choice: - *Per-workspace extension:* one extension that is generic and reads any `portable-plugin.yaml` in the workspace and sets up accordingly. This would be flexible (one extension can handle multiple plugin projects). But that means if multiple plugins are in one project, does it handle both? Possibly yes. However, typically you might use one plugin at a time. It also adds complexity and potential security issues (executing arbitrary hooks from any YAML). - *Specific extension per plugin:* where the extension is basically a hard-coded implementation of this one plugin's logic. That is more like how a typical VS Code extension is written (if someone were to port the plugin manually, they'd write an extension for it). This is easier to implement for known logic, but then every plugin would require building and installing a separate extension, which might be fine if publishing to Marketplace. - We can choose an approach in between: provide a *framework extension* that can load plugin definitions (like a runtime engine reading the YAML), so end-users only install one "Portable Copilot Plugin Runner" extension, and drop plugin spec files in their project to use them. This would be quite powerful but more ambitious. For now, focusing on one plugin's adapter extension is sufficient as a reference.

Given time constraints, we might not fully implement such an extension in the blueprint, but we will outline how to do it. For example, in code: - On extension activation: read `portable-plugin.yaml` from workspace (if present). - Register chat slash commands: In package.json, we actually can't dynamically add slash commands after installation; they must be declared ahead (unless using proposed API to contribute chat items on the fly). But since our extension knows the plugin's commands, we could generate the package.json at build time from the spec (which means a build step for the extension). - Alternatively, we skip using VS Code's built-in slash command contributions and just watch chat input. However, no public API to intercept chat mid-flight is documented (the Chat Participant API is oriented around writing a full participant). - Possibly, we *could* create a chat participant for the plugin: say `@assistant` or `@plugin`, and have commands under it. But that means user would have to @mention it, which is not as nice as slash. So maybe we rely on slash commands via prompt files and extension just does the behind-the-scenes work.

- The extension can still do the event hooking (no problem there, just normal VS Code events). - And it can manage MCP servers or any persistent processes.

We will suggest that in early prototypes, one might avoid writing the extension and see how far `.github` files can go. The user in their instruction indeed said: *"do file-based first to verify feasibility, then if needed do the extension."* So that aligns with our plan: 1. **Prototype Phase 1**: Use `.github` prompts/agents/skills to mimic the plugin in Copilot. Evaluate what works and what doesn't (e.g., Copilot might not spontaneously run tests on /run-tests command's request, or it might not know to do something). 2. **Phase 2**: Implement VS Code extension for the missing pieces: - Possibly needed to reliably run shell commands or handle outputs. - Possibly needed to achieve automation without user always clicking (if desired). - Packaging for shareability.

Security & Compliance considerations: When porting plugins, we must ensure we don't break enterprise policies. A few points: - If an enterprise disallows Copilot's "web access" or "terminal access", our plugin's functionality needing those will be limited. For example, if `network: none` in policies, our plugin should detect that (maybe via an environment variable or Copilot settings if available) and not attempt those actions or provide a graceful fallback (like instruct user to do it manually). - Secrets: We should never hard-code secrets in a plugin spec. If a plugin needs an API key, ideally the user provides it via environment or a command. Claude Code might allow storing an API key in `~/.claude/credentials.json` or similar (not sure, but maybe as part of MCP config you can reference env vars). In VS Code, an extension can use `vscode.secretStorage` to securely store user tokens. We'd document that "upon first use, the plugin will ask for your API key for service X and store it." - Logging: On Claude, plugin actions are logged in the chat or debug logs. On VS Code, our extension can log to an Output Channel for transparency (especially for hooks that run automatically). - PII: If plugin deals with PII (say it has a tool to fetch customer data), both Claude and Copilot need to be handled carefully because sending PII to the LLM could violate policies. We might advise filtering such data or requiring user confirmation. - Legal: If distributing externally, obviously abide by terms (for instance, using the Copilot name or APIs properly; our extension should not violate OpenAI's terms like sending data outside allowed scope, etc.). But since we're targeting official APIs, we're fine.

Implementation Plan (4-Week Timeline)

Assuming we want to build a working prototype and then a robust version:

Week 1: Prototype and Basic Architecture - *Goal:* Have a basic portable plugin working in both environments with core features. - **Tasks:** - Finalize the portable spec schema in a small example (like the Hello World plugin above). - Write a quick conversion script to generate: - Claude plugin files (manifest, one command, one agent, one skill). - Copilot `.github` files for the same. - Manually test the example: - Load in Claude: run the command/agent and verify output [84] [85]. - In VS Code, enable Copilot skills/agents, copy in the .github folder: - See if the slash command appears and works [19] [157]. - Try switching to the custom agent and see if it follows instructions. - This will reveal if Copilot indeed picks up the files (check Copilot output for signs it loaded the skill, etc.). - Identify issues: e.g., maybe the prompt command didn't cause the model to use a tool as expected, etc. If so, adjust approach (maybe need extension sooner for certain parts). - **Deliverable:** A basic "hello world" plugin spec and its generated outputs, plus a short report of what worked and what didn't in Copilot.

Week 2: Expand Multi-Agent Orchestration and Tools - *Goal:* Implement the multi-agent workflow (planner -> coder -> tester) in the portable plugin, and integrate an external tool usage. - **Tasks:** - Expand the spec to include multiple agents and a more complex command (like "implementFeature" that triggers plan -> code -> test, either via sequential slash commands or as a scenario). - Add a dummy MCP tool (e.g., a `calculator` tool as a simple Python MCP server) and a skill or command to use it, just to prove the concept. - Update conversion scripts to handle multiple agents, etc. - **Claude side:** Test that one agent can call another – likely through user invocation or a hook. Might need to write a Claude hook (SubagentStop) that automatically prints a suggestion "Use X agent now." Ensure that works. - **Copilot side:** Test handoffs – see that after using agent1, the follow-up for agent2 appears [152]. If not automatic, see if detection can route a question to the right agent by examples. - Experiment with making Copilot run something in terminal via a slash command prompt (like we want it to run tests). Possibly deliberately ask Copilot in chat "run tests" to see what it does. If unsatisfactory, plan extension approach for that. - **Deliverable:** Updated plugin spec (with multi-agents, etc.), demonstration of it working (maybe a short video or log of usage in both envs). List of remaining limitations (e.g., "Copilot doesn't auto-run tests on command X").

Week 3: Develop VS Code Extension Adapter & Hooks - *Goal:* Build the VS Code extension to fill gaps (especially automation hooks or reliable tool calls). - **Tasks:** - Scaffold a VS Code extension (`yo code` or similar). Add necessary permissions in manifest (if we need to run terminal commands or use proposed APIs). - Implement file-save and test-run watchers if hooks require them: - E.g. on file save, run tests (using VS Code Tasks API or spawn a process). - On test complete, collect results and either automatically send to Copilot (explore `vscode.chat.sendMessage` if exists in API – the docs mention *ChatProvider* which might be used to intercept user queries or to provide responses, not sure about injecting new user messages programmatically). - If direct injection isn't possible, have extension pop a button "Analyze with Copilot" linking to a prompt file or performing an action. - Integrate with the spec: - Possibly have the extension include the plugin spec (maybe as JSON) so it knows what to do. Or have it read the YAML at runtime from workspace. - If multiple plugins could exist, ensure it identifies which one to load (maybe by file name or a setting). - *Tool integration:* If an MCP server is defined, the extension should start it. We may use Node's `child_process` to spawn e.g. Python script. Then, how to notify Copilot's agent of this tool? According to MCP guide, if a server is started on stdio and outputs the proper registration info, VS Code might auto-detect it in dev mode. Possibly we need to configure `dev.environmentVariable` or call a command. This part might require reading VS Code's internal or using the `LanguageModel.connectTo` API if exists. For now, we might simulate by using a simpler route: implement that same logic within the extension (like if it's just a calculator, handle it in extension code when AI asks). - Test the extension: - Load it in VS Code (as "development extension"), open a project. - Ensure slash commands still show (coming from prompt files). - Trigger scenarios: e.g. Save a file -> extension runs tests -> after test output, extension posts a prompt or we manually ask Copilot and see if it has the info (some extension could possibly push context via *Workspace Context Variables* – Copilot has variables like ${file}, etc. There's `vscode.ChatContext` that might allow adding context out-of-band). - It's possible none of the extension <-> Copilot Chat integration is officially allowed beyond what we used (prompt files, skills). If so, our extension might have to be creative: for example, after running tests, it could write the results to a file and then set a user-level *skill* containing those results so that Copilot sees them. That's hacky but could work (e.g. update a `last-test-results.md` skill and the description triggers if it finds "Test run results" in conversation). - Alternatively, extension could call OpenAI API directly (if allowed by user's credentials), but then it's separate from Copilot service and likely not allowed by terms if using Copilot subscription. - Essentially, find any path to feed info to Copilot. The *MCP resource* feature might let extension supply data that the model can retrieve. E.g., extension could add a *Resource* of type "testResults" that Copilot can see (the MCP guide mentions providing *Resources* and *Elicitation* [32] [158]). - Focus on at least demonstrating one hook working: e.g. after save, tests

run, and maybe we manually copy the output to chat to simulate the model analyzing it. - **Deliverable:** A working VS Code extension (in dev mode) that implements at least one automation from the plugin. Documentation on how it interacts with Copilot and any API usage/tricks done. Also, updated spec if needed to accommodate differences.

Week 4: Stabilization, Documentation, and Tests - *Goal:* Polish the system, ensure reproducibility, write docs. - **Tasks:** - Write **user documentation**: how to install the Claude plugin (point to marketplace or how to use `claude --plugin-dir`), how to use the Copilot side (if extension is published, how to install it; if file-based, how to integrate the .github folder). - Write **developer documentation**: how to create a new portable plugin using our blueprint. Possibly provide a template repository or CLI (could be simple: cookiecutter to create spec and directories, and instructions for building). - Setup some **automated tests**: - Unit tests for spec conversion scripts (verify that given a spec input, the expected Claude files and Copilot files are generated). - If possible, integration test: maybe using Claude's CLI in headless mode to run a slash command and capture output (Claude Code might have a headless mode where we can send a command and get the result – not sure if `claude` CLI can be scripted; if not, a manual step). - For Copilot, integration tests are harder (no CLI for chat). Perhaps we skip or do a semi-manual test script (e.g. instruct QA to verify certain behaviors). - Address any found bugs: e.g. adjust prompts if the AI doesn't respond correctly. Often, tuning might be needed (like adding more explicit instruction in an agent's prompt if it misbehaves). - Consider edge cases: no network environment (make sure plugin doesn't hang or try to call unreachable service), large projects (skills injection might be heavy – maybe our skills should be concise). - Package the VS Code extension (vsix) if we have one, and test installation on a fresh environment. - Prepare for publishing: e.g. create a marketplace listing for the extension (or instruct user to sideload if private). - Summarize risks and mitigations in documentation.

- **Deliverable:**
- Repository with `portable-plugin.yaml`, conversion scripts, and VS Code extension code.
- A sample portable plugin (possibly the multi-agent example we built) fully functioning in both Claude and Copilot.
- Documentation including an **Executive Summary** (for decision makers), a **User Guide** (for end-users of the plugin), and a **Developer Guide** (for plugin authors to use the system for their own plugins).
- A checklist for porting existing Claude plugins: e.g., "Ensure your Claude plugin doesn't use unsupported hook events X, or if it does, plan an extension workaround. Check if all needed tools exist in Copilot (if not, might implement via MCP or skip)." etc.
- Identify any features that simply can't be done in Copilot yet (for instance, if something absolutely requires a synchronous model-in-the-loop hook which Copilot can't do, note that as limitation).

Risks, Limitations, and Alternatives

Current Limitations in Copilot: Through this research, it's clear that not every Claude capability maps cleanly: - *Autonomous action without user approval:* Copilot by design keeps the developer in control more. For example, Claude can auto-apply edits and run tests in a loop (with permission granted once). Copilot's agent mode does some of that, but any custom automation we add might need user clicks or at least pre-configuration. Our plugin might not achieve the same fully hands-free operation on Copilot, especially due to missing hook injection. We mitigate by using Copilot's **hand-off UI** and by nudging the user (notifications). - *Parallel agent reasoning:* Claude could theoretically have subagents plan while main agent waits. Copilot doesn't expose a way to have two LLM contexts in parallel – it's one at a time. So our design

uses sequential steps (which, to be fair, Claude's use of subagents is also sequential for a given conversation – it's just orchestrated by main agent). - *Complex Hook logic:* For things like "decide intelligently if all tasks are done" (Claude's Stop hook with prompt decision) [147] [148], Copilot offers no such hook. If needed, one could code this logic into the extension or into the agent's own behavior. E.g., make the agent check conditions itself. That might bloat the agent's prompt though. Another approach: use an *external orchestrator* – i.e., instead of relying on Copilot's built-in loop, have our extension orchestrate OpenAI API calls (basically write our own simplified agent chain outside of Copilot). But that defeats using Copilot's subscription/IDE integration and is against the "in Copilot" goal. So we consider that an alternative path only if necessary for advanced automation: essentially implementing a custom chain-of-thought using the Copilot API if it becomes available (like how someone might use the OpenAI functions with VS Code context to do their own agent). - *Stability of Copilot internal APIs:* We had to rely on certain preview features (skills, etc.) and perhaps will use proposed VS Code API (which may change). There is a risk that future updates to Copilot or VS Code break our adapter (for example, if the format for prompt files changes or if new security measures block reading .github files). We should keep the design flexible – for instance, if prompt files were deprecated in favor of a new official plugin system, we should be ready to adopt that. (Microsoft has hinted at more *"Copilot extensibility"* coming – possibly they will have an official plugin mechanism similar to ChatGPT's plugins, in which case our portable spec could directly target that as another adapter.) - *Enterprise restrictions:* Some companies might disable user-added custom agents or prompt files for compliance (since those can instruct the AI arbitrarily). If so, our Copilot side might not work at all unless they enable it. If extension is allowed, they might still restrict running commands. In worst case, the plugin's more sensitive aspects would be locked down, and only the informational parts (like skills) work. As a workaround, an enterprise could explicitly vet and allow a specific extension (which can enforce policies in code). - *Alternatives if Copilot falls short:* If certain workflows can't be embedded in Copilot, an alternative is to use **GitHub Actions or external orchestration**: for example, if a plugin's goal is to handle PR reviews with multiple steps, one could create a GitHub Action that triggers the Claude plugin or ChatGPT to do a review, and then output results. That's not in-editor interactive, but it's a fallback. Similarly, using **Anthropic's Claude API or OpenAI API with a custom script** to mimic the plugin might be done. However, since the question specifically wants it in Copilot (IDE context), we stick to that scope. - *Gemini Pro / Other LLMs:* The design is model-agnostic in principle. If Google's Gemini (or others integrated into IDEs) support similar constructs (agents, skills), we could adapt. For now, focusing on Claude and Copilot covers two major ecosystems. But we note that our spec aligns with an **open standard trend**: agent skills, MCP, etc., which suggests future platforms (e.g., perhaps Visual Studio, JetBrains, etc.) might converge on these concepts too. So hopefully our portable spec would map to those with minimal changes.

Security Considerations Recap: - We will *not* allow arbitrary code execution in the Copilot adapter without explicit user opt-in (the extension will use VS Code's `executeInTerminal` which by default prompts for permission unless auto-approve is configured [122]). - Our extension will abide by VS Code's security (marking itself as not running in untrusted workspaces if it runs code). - The Claude plugin side is as safe as any Claude plugin (if following Anthropic's guidelines and requiring permissions). - Logging: The extension can log important actions (like "Ran tests via plugin at 3:00pm"). - If the plugin accesses external APIs or sensitive data, ensure compliance (e.g., don't accidentally send sensitive code to an external service through a plugin tool – both Claude and Copilot caution about that). - The spec should maybe include a field to declare data handling categories (like "this plugin might transmit code to service X" so users know). But that's beyond our scope for now.

Checklist to validate portability (for a plugin author who has a Claude plugin and wants to port): 1. **Commands**: Do all slash commands make sense in Copilot? (Yes if they are just AI prompts or file ops. If a

command runs complex logic via Python script in Claude, that won't port unless we provide an equivalent tool/extension). 2. **Agents**: Are the Claude subagents fairly self-contained as prompts? (If they rely on Claude-specific tools not in Copilot, we must create or drop those tools). For example, Claude's `Search` tool vs Copilot's `search` – conceptually same, just ensure naming. 3. **Skills**: If the Claude plugin uses skills, easy port. If it uses them heavily to inject context, check Copilot context window (currently GPT-4 8k or 32k for some, vs Claude up to 100k). Copilot might drop or truncate some skill content if too large, whereas Claude could handle bigger – need to design around that (progressive disclosure helps anyway). 4. **Hooks**: Mark down each hook and determine: can extension do this? If not, is it critical? If critical and can't do, maybe not portable yet. Possibly ask user to perform that step manually as a compromise. 5. **External dependencies**: Does the plugin rely on local dev environment (like running `make` or `docker`)? If so, Copilot extension can run those, but ensure the user environment is set up (the extension will just invoke commands, errors if not installed). 6. **Testing**: Verify the Copilot version of the plugin in a safe environment first – some mis-configuration could cause unwanted file changes. Always test on sample project.

Decision Tree: Extension vs No Extension: - If your plugin only adds new *AI guidance* (commands that are basically prompts, agents with instructions, read-only analysis) – **no extension needed**. Use .github files approach. - If your plugin needs to **perform actions in the environment** (run builds, modify files automatically beyond what the AI normally does) – attempt to see if the AI can do it on its own via its tools. If the AI reliably can, maybe still no extension. If not, then **extension is needed**. - If needed, decide: - Can you reuse a **generic adapter extension**? (If we manage to open-source a generic one that reads spec – that would be ideal future, not immediate.) - Or build a **specific extension** for this plugin? (Likely yes for now). - If building extension: weigh the complexity vs benefit. Perhaps for an internal team, copying .github config and educating team to click a few extra buttons might be sufficient without an extension, which is less maintenance. - Also consider update frequency: If plugin logic will evolve often, updating an extension frequently is overhead; using config files might be easier (just pull latest .github changes). - If your devs use primarily VS Code, an extension is fine; if some use IntelliJ or others, maybe the file-based approach could be extended to them (though currently only VS Code and Visual Studio read those config files; other IDEs won't). - If **enterprise constraints** block one approach (e.g. extension installation is easier through internal marketplace than telling everyone to copy files), go with extension.

In conclusion, this research demonstrates that a large portion of Claude Code's plugin functionality can be reproduced in GitHub Copilot's ecosystem. By defining a clear portable specification and using the techniques above, we can implement a single plugin that augments both AI coding assistants. We provided a capability matrix and example to show the one-to-one correspondences and workarounds. With these building blocks, teams can start creating **AI assistant plugins that are platform-agnostic**, ensuring that whether a developer is using Claude in a cloud IDE or Copilot in VS Code, they have access to the same custom "AI teammate" capabilities [159] [5] .

Going forward, as tooling APIs standardize (MCP, agent skills, etc.), we anticipate this will become even easier – potentially a future where you write a plugin spec and all AI IDEs can import it. This blueprint is a step in that direction.

Appendix: Key Sources

- Anthropic Claude Code Documentation:
- *Claude Code Plugins Intro – "Create plugins"* (Anthropic docs) [160] [161]
- *Claude Code Plugins Reference* – manifest schema, structure, etc. [162] [79]

- *Claude Code Hooks Reference* – events like PreToolUse, examples [9] [10]
- *Claude Code Subagents Guide (Shipyard blog)* – subagent YAML format and usage [46] [47]
- *John Conneely's blog on Claude Extensibility* – mental model comparing skills, subagents, commands, plugins [163] [164]
- GitHub Copilot and VS Code Docs:
- *VS Code Chat Extensibility (Chat Participant & MCP)* – VS Code API guides [108] [28]
- *GitHub Copilot User docs* – custom prompts and agents:
  - *Prompt files in .github/prompts* [17] [18]
  - *Custom agents in .github/agents* [117] [118]
  - *Agent skills in .github/skills* [119] [22]
- *Microsoft Learn – Copilot slash commands (MSSQL extension example)* [29] [30]
- *GitHub Community Discussion on custom slash commands* – confirming extension needed for custom commands [165] [166] .
- *VS Code Copilot Chat README (GitHub)* – mentions agent mode, participants, etc. [95] [167] .
- Cross-platform Standards:
- *AgentSkills standard* – noted as open and used by both Claude and Copilot [21] [26] .
- *Model Context Protocol (Anthropic & VSCode blogs)* – introduction of MCP connectors [168] [27] .
- Example Community Plugins:
- *BMAD Method* – large multi-agent framework for Claude (GitHub) [169] .
- *Claude Flow / Autogen* – though not cited above, background of multi-agent orchestration attempts.
- These informed our understanding but not directly cited due to focusing on official sources.

All in all, the above sources provided the foundation for reconstructing Claude's plugin architecture and mapping it to Copilot's capabilities, enabling us to propose this unified approach.

---

[1] [2] [4] [5] [6] [11] [12] [41] [59] [76] [159] A complete overview of the Claude Code plugin ecosystem
https://www.eesel.ai/blog/claude-code-plugin

[3] [50] [51] [52] [57] [58] [60] [163] [164] Understanding Claude Code: Skills vs Commands vs Subagents vs Plugins | #95
https://www.youngleaders.tech/p/claude-skills-commands-subagents-plugins

[7] [8] [36] [37] [38] [39] [73] [74] [75] [77] [78] [79] [80] [83] [86] [87] [88] [92] [93] [124] [125] [162] Plugins reference - Claude Code Docs
https://code.claude.com/docs/en/plugins-reference

[9] [10] [35] [63] [64] [65] [66] [67] [68] [69] [70] [71] [72] [89] [90] [91] [133] [134] [147] [148] [149] Hooks reference - Claude Code Docs
https://code.claude.com/docs/en/hooks

[13] [14] [15] [16] [112] [113] [114] [115] [116] [117] [118] [129] [130] [136] [137] [139] [142] Custom agents in VS Code
https://code.visualstudio.com/docs/copilot/customization/custom-agents

[17] [18] [19] [20] [97] [98] [99] [100] [127] [128] [135] [157] Use prompt files in VS Code
https://code.visualstudio.com/docs/copilot/customization/prompt-files

[21] [22] [23] [24] [25] [26] [119] [120] [121] [122] [131] [156] Use Agent Skills in VS Code
https://code.visualstudio.com/docs/copilot/customization/agent-skills

27 28 31 32 123 158 MCP developer guide | Visual Studio Code Extension API
https://code.visualstudio.com/api/extension-guides/ai/mcp

29 30 103 106 107 Quickstart: Use GitHub Copilot Slash Commands - MSSQL Extension for Visual Studio Code | Microsoft Learn
https://learn.microsoft.com/en-us/sql/tools/visual-studio-code-extensions/github-copilot/slash-commands?view=sql-server-ver17

33 34 40 42 43 44 45 53 54 55 56 61 62 81 82 84 85 126 138 144 160 161 Create plugins - Claude Code Docs
https://code.claude.com/docs/en/plugins

46 47 48 49 145 146 Shipyard | Claude Code Subagents Quickstart: what they are + how to use them
https://shipyard.build/blog/claude-code-subagents-guide/

94 95 167 GitHub - microsoft/vscode-copilot-chat: Copilot Chat extension for VS Code
https://github.com/microsoft/vscode-copilot-chat

96 Getting started with prompts for GitHub Copilot Chat
https://docs.github.com/en/enterprise-cloud@latest/copilot/how-tos/chat-with-copilot/get-started-with-chat

101 102 104 105 108 109 110 111 150 151 152 153 154 155 Chat Participant API | Visual Studio Code Extension API
https://code.visualstudio.com/api/extension-guides/ai/chat

132 Get started with chat in VS Code
https://code.visualstudio.com/docs/copilot/chat/copilot-chat

140 Customize chat responses - Visual Studio (Windows) - Microsoft Learn
https://learn.microsoft.com/en-us/visualstudio/ide/copilot-chat-context?view=visualstudio

141 Slash Commands - Invoke Prompts and Manage Chats
https://developercommunity.visualstudio.com/t/Slash-Commands-%E2%80%94-Unified-and-Extensible/10992996

143 Using GitHub Copilot CLI
https://docs.github.com/en/copilot/how-tos/use-copilot-agents/use-copilot-cli

165 166 Custom slash commands for Copilot Chat · community · Discussion #74564 · GitHub
https://github.com/orgs/community/discussions/74564

168 Introducing the Model Context Protocol - Anthropic
https://www.anthropic.com/news/model-context-protocol

169 bmad-code-org/BMAD-METHOD: Breakthrough Method for ... - GitHub
https://github.com/bmad-code-org/BMAD-METHOD