ChatGPT

Autonomous Multi-Agent System Build-Out Plan

This document outlines a phased technical specification for an autonomous multi-agent system that begins as a developer automation tool and incrementally evolves into a complex organization of agents (research assistants, a crypto analysis swarm, and a social media marketing team). Each phase details the objectives, required tools (MCP servers/APIs), agent roles and responsibilities, coordination model, prompt engineering and memory usage, fault tolerance measures, and execution guidance. The phases build on each other, enabling Claude Code (with internet access, git integration, and MCP support) to **recursively expand and improve itself** into increasingly advanced agent teams.

Phase 1: Developer Automation Foundation

Objective & Scope: Establish a single **Developer Agent** that can handle basic coding tasks autonomously. The agent will set up the project structure and initial config, acting as a baseline for future expansions. Scope is limited to one agent performing coding, editing, and simple testing under user guidance.

Required Tools/APIs: - **Filesystem MCP** – for reading/writing project files (code, configs).
- **Git MCP** – for version control (committing code, creating branches).
- *(No internet or external APIs yet; knowledge is from the model and any provided context.)*

Agent Role: *Developer Agent* – Acts as an AI software engineer. Responsible for writing code, modifying files, and managing the codebase. Initially the sole agent, it plans and implements user requests directly (no sub-agents). It should follow coding best practices and project guidelines.

Coordination Model: Single-agent (no coordination needed). The Developer Agent operates independently. There is no meta-agent yet; the user (or system prompt) provides high-level goals which the agent executes directly. All reasoning is internal to the single agent.

Prompt Engineering & Memory: Provide the agent with a clear persona and project context. For example, include a `CLAUDE.md` with project setup notes and coding standards [1] [2]. The initial prompt should state the agent's role ("You are an autonomous Developer AI tasked with building and improving this project…") and any important constraints (language, style guide, etc.). No long-term memory beyond the session is used in this phase (rely on conversation context). Keep prompts concise and focused.

Fault Tolerance: With only one agent, error handling is manual or sequential. The agent should be encouraged (via prompt) to self-verify its work (e.g. reading back files it edited to ensure correctness) and to handle simple errors (syntax errors, etc.) by itself. If a task fails (e.g. code doesn't run), the agent should catch the failure from tool output and attempt a fix. However, without specialized roles, complex faults may require user intervention or iterative refinement.

Execution Guidance: In this phase, Claude Code should initialize the project environment: 1. **Initialize Repository:** Use Git MCP to create a repo structure (e.g. `git init` and create initial folders/files).

2. **Create Config/Docs:** Write a `README.md` or `CLAUDE.md` describing the project and any known requirements (this will be auto-included in context in future sessions [1]).
3. **Set Permissions:** Adjust Claude Code's allowed tools (via `/permissions` or settings) to always allow file edits and git operations for smooth automation [3].
4. **Verify Workflow:** Have the Developer Agent perform a simple coding task (e.g. create a "Hello World" script and commit it) to confirm it can read/write files and commit to git autonomously.

By the end of Phase 1, the system should have a basic project scaffold and a single autonomous developer agent ready to expand its own capabilities.

Phase 2: Introduce Planning & Task Decomposition

Objective & Scope: Enhance the system with basic planning ability by introducing a **Planner (Meta-Agent)** that can break down high-level objectives and guide the Developer agent. This phase establishes a simple two-agent interaction: one agent plans, the other executes. The scope is still focused on software development tasks, but with better structured thinking.

Required Tools/APIs: (Same as Phase 1: Filesystem and Git; still no external web access). Optionally, a basic **sequential reasoning** mechanism can be introduced (see Phase 5) but here we mainly use the Planner's own logic.

Agent Roles: - *Planner (Meta-Agent):* A top-level agent that receives user goals and formulates a task list or plan. It decides which steps are needed and in what order. It does not directly edit files; instead, it issues instructions to the Developer.
- *Developer Agent:* Performs the coding work as in Phase 1, but now takes guidance from the Planner. It focuses on implementing specific tasks (e.g. "Create module X according to the plan spec") rather than the overall goal.

Coordination Model: Centralized (Hierarchical): The Planner acts as a simple coordinator, delegating tasks to the Developer agent one at a time. This is essentially a manager-worker model [4]. The Planner analyzes the goal, outlines sub-tasks, then communicates each sub-task to the Developer. The Developer reports back when done, and the Planner integrates results or assigns the next task. Communication can be implemented via a shared memory (file or context) or direct messaging if running two Claude instances via a swarm connection.

Prompt Engineering: Define distinct prompts or personas for each role. For example: - Planner's prompt: "You are a **Project Planner AI**. Your job is to break high-level goals into step-by-step technical tasks, and oversee their execution. Provide clear, numbered instructions for the Developer agent. After each step, await confirmation or results."
- Developer's prompt: "You are a **Developer AI** taking instructions from the Planner. Implement the task exactly as described, then report completion or any issues."
Use system or role messages to imbue each agent with its responsibilities and limit their scope. This enforces the *Single Responsibility Principle* (each agent focuses on its own task) [5]. Both agents share access to the project files and codebase.

Memory Use: Still no persistent long-term memory (that comes in Phase 4), but the Planner should log the task plan (perhaps in a plan.md file or in the conversation) so the Developer and future phases can reference it. The Developer can refer to this plan file to stay on track. Maintaining a history of completed tasks (e.g. updating the plan with checkmarks) helps track progress.

Fault Tolerance Patterns: If the Developer encounters an error implementing a task, it should communicate the problem back to the Planner. The Planner can then revise the plan or adjust the instructions. This introduces a simple feedback loop: the Developer's failure triggers re-planning. For example, if a library is missing, the Developer informs the Planner, who adds a sub-task to install that dependency. This **iteration cycle** (Planner -> Developer -> feedback -> re-plan) builds resilience. If the Developer is silent (task done), the Planner assumes success and proceeds. This way, a failure in one agent leads to re-evaluation by the other, improving reliability [6].

Execution Guidance: Claude Code can implement this phase by launching two agent instances (using a tool like Claude Swarm or a custom coordinator logic): 1. Define the two roles in a configuration (e.g. a YAML or script): one instance named "planner", one "developer". Set their descriptions/prompts as above. Ensure the Planner can communicate tasks to Developer (e.g. via a shared file or direct message channel if supported by the framework).
2. **Run Planner-Agent:** The Planner reads the user's high-level request (e.g. *"Implement feature Y"*) and produces a structured task list. The tasks can be saved to `tasks.md` for traceability.
3. **Delegate to Developer:** The Planner either writes a message for the Developer or calls a function to invoke the Developer agent on each task. Claude Swarm natively supports such delegation via MCP connections [7], wherein the main agent can send instructions to connected agents.
4. **Developer Execution:** The Developer agent executes the given task (editing files, running commands as needed). It then marks the task done or reports issues.
5. **Iteration:** The Planner reads the Developer's output or status. If incomplete or error occurred, Planner revises instructions or adds a remediation step (this may loop until success). If completed, move to next task until all tasks in the plan are done, then signal phase completion.

By the end of Phase 2, the system will have an elementary multi-agent framework: a meta-agent planner coordinating a worker agent. This sets the stage for adding more specialized roles.

Phase 3: Add Code Review and Quality Assurance

Objective & Scope: Improve code quality and reliability by introducing a **Reviewer Agent**. This agent will critique and test the Developer's output, providing an internal "quality gate" before changes are finalized. The scope expands the developer team to three roles (Planner, Developer, Reviewer), establishing an **internal review loop** for software tasks.

Required Tools/APIs: - **Testing/Execution Environment:** (optional) If available, the Reviewer can run code or tests via a shell tool. If not, it can at least read the code and logically analyze it.
- Continue using **Filesystem** (to read code files) and **Git** (commit only after review approval). Possibly use a linting or static analysis MCP (e.g. an ESLint MCP for JS projects) for automated feedback, if such tool is accessible.

Agent Roles: - *Planner (Meta-Agent):* Same as Phase 2, but now also routes tasks to the Reviewer when appropriate. The Planner ensures that after the Developer finishes a task, the Reviewer is engaged before considering the task done.
- *Developer Agent:* Same as before – implements tasks according to plan. It now expects that its work will be reviewed and might need to make revisions.
- *Reviewer Agent:* A new role acting as a code auditor and tester. It reviews code changes, checks for bugs, style issues, or mismatches with requirements. It can run the code (if possible) or at least perform static analysis. The Reviewer's goal is to catch mistakes and ensure the deliverable meets the acceptance criteria before informing the Planner that the task is truly complete.

Coordination Model: Centralized coordination with sequential workflow: The Planner remains the coordinator. The task lifecycle becomes: Planner assigns task -> Developer implements -> Reviewer validates -> feedback loop if issues -> Planner finalizes. The Planner can be scripted to always invoke the Reviewer after the Developer signals completion. Alternatively, a chain can be established where the Developer, upon completing code, automatically triggers a review step (this can be done by the Planner or by writing a file that signals the Reviewer to start). All agents are still under a hierarchy directed by the Planner (the Planner might only step in actively at the beginning and end of the cycle, letting Developer and Reviewer trade messages in between).

Prompt Engineering: Define the Reviewer's behavior clearly: - Reviewer's prompt: "You are a **Code Reviewer AI**. Your role is to scrutinize the Developer's output for errors, bugs, or unmet requirements. You have access to the latest code changes. If issues are found, provide clear feedback and request fixes. If the code is good, approve it."
Encourage the Reviewer to be thorough but constructive. Also adjust the Developer's prompt: "After implementing, await feedback from the Reviewer and be prepared to make revisions as needed." The Planner's prompt can instruct it to use the Reviewer's feedback in deciding if a task is done or needs re-assignment. Use **tools** like unit tests or linting by having the Reviewer call them (e.g., run `npm test` via a Bash tool, or open the file to inspect content). The prompts should guide agents to use these tools where appropriate (for example, include in the Reviewer's instructions a hint like "use available tests or linters to verify code quality").

Memory Use: At this stage, still using ephemeral session memory (the conversation and any task log files). However, consider logging review feedback in a persistent place (like a `review_log.md` or commit messages) so the system can learn from past mistakes. This can later be fed into a memory system when introduced. For now, the focus is on immediate feedback rather than long-term learning.

Fault Tolerance Patterns: The addition of a Reviewer inherently increases fault tolerance: if the Developer produces faulty output, the Reviewer catches it and prevents it from moving forward, prompting a fix. This **redundancy** means the system doesn't rely on one agent's perfection [6] [8]. Patterns to implement: - **Feedback Loop:** The Reviewer, upon finding an issue, effectively becomes a "critic" that triggers rework. The Planner or Reviewer can send the Developer a new sub-task: "Fix the bug X identified in review." The Developer should then address it and resubmit to Reviewer. Loop until Reviewer is satisfied.
- **Automated Testing:** If possible, the Reviewer runs automated tests. A failing test triggers a fix task similar to above. This ensures that errors are caught early.
- **Approval Check:** Only once the Reviewer approves does the Planner mark the task done and perhaps merge the code (e.g., commit to main branch). If the Reviewer cannot validate (e.g. lacks domain

knowledge), it can escalate to the Planner or leave a note for human inspection. But since the goal is full autonomy, ideally the Reviewer has sufficient guidance to decide on its own.

Execution Guidance: Claude Code (via the Planner agent) should now orchestrate a review step: 1. **Integrate Reviewer into Workflow:** Update the Phase 2 coordination script or YAML to add a "reviewer" agent instance. For example, in Claude Swarm, define a new instance with role description "QA Reviewer" and connect it such that the Planner (lead) can send it code or instructions after Developer finishes tasks [7].
2. **Task Handoff:** After Developer signals a task is completed, Planner forwards the diff or file list to Reviewer. This can be done by the Planner explicitly opening those files for the Reviewer agent or summarizing the changes.
3. **Reviewer Execution:** The Reviewer reads the changes. If tests are available, it runs them using a Bash tool. If issues are found or tests fail, it writes feedback (which the Planner will relay to Developer). If all is good, it gives approval.
4. **Iterate if Needed:** If feedback was given, the Planner assigns a "fix" sub-task to Developer based on Reviewer's notes, then repeat the review. This may iterate multiple times (the spec should allow loops until resolution).
5. **Finalize Task:** Once Reviewer approves, the Planner can proceed to mark the task as done. At this point, the Planner might instruct the Developer to commit the changes to git (ensuring that only reviewed code is committed to main). The Planner then moves to the next item in its plan.

By the end of Phase 3, the development sub-system is much more robust: for every implemented change, an independent agent verifies the quality. This lays a strong foundation of reliability before scaling up complexity.

Phase 4: Integrate Persistent Memory

Objective & Scope: Introduce a persistent memory store to enable the agents to **remember context and knowledge over long sequences** and across sessions. This phase adds a **Basic Memory MCP server** to the system, allowing agents to record and retrieve important information (plans, decisions, domain knowledge) in a long-term knowledge base. The scope here is not adding new agent roles, but enhancing all existing agents with memory utilization.

Required Tools/APIs: - **Basic-Memory MCP** – A memory server for persistent conversational/contextual memory. For example, the *Basic Memory* plugin stores notes in local Markdown files that both humans and LLM agents can read/write [9]. This will act as a shared knowledge repository.
- Existing tools (Filesystem, Git, etc.) remain in use. Basic Memory may integrate with file storage (e.g. saving notes to a `/memory` directory).

Agent Roles: (No new distinct roles; enhancement of current roles) - *All agents (Planner, Developer, Reviewer)* now leverage memory. For instance, the Planner can record the overall roadmap or architectural decisions in memory, the Developer can log gotchas or APIs learned, and the Reviewer can catalog common mistakes to watch for. Memory acts as an auxiliary tool accessible by any agent that needs context. - Optionally, a dedicated *"Knowledge Manager"* agent could be introduced whose job is to maintain and curate the memory (e.g. summarizing discussions, removing redundant info). This agent would ensure memory doesn't grow unwieldy. (This is an optional specialization if needed, otherwise memory is handled ad-hoc by existing agents.)

Coordination Model: Centralized with shared memory blackboard: The structure remains Planner-led. The memory acts as a **blackboard** in the system – a common area where agents post information and look up previously posted info. This is not exactly an agent, but a resource. All agents need disciplined coordination on memory usage (e.g. Planner might enforce that after each task, key details are written to memory). The presence of memory also enables a form of indirect coordination: e.g. the Developer can leave a note for the Reviewer about a known limitation of the code, etc., although in practice the Planner can intermediate such communication.

Prompt Engineering (Memory Usage): Update prompts to encourage memory interaction: - Tell agents that a memory tool is available: e.g. *"You have access to a persistent knowledge base. Use it to store important information or retrieve past context by asking for notes."* - When an agent learns something significant (e.g. the result of a long debug, or research on a library in later phases), it should commit that to memory: *"Note: X library requires Y – saved to memory."* - The memory can be queried with natural language via the MCP (Basic Memory allows queries like "What do I know about *topic*?" [10]). In prompts, encourage agents to query memory at the start of a task for relevant info: *"Recall relevant knowledge from memory before proceeding."* This ensures continuity across tasks. - Maintain a structure in memory files (e.g. categorize notes by topic or agent). Prompt agents to write concise, linkable notes (Basic Memory builds a Markdown knowledge graph [11]).

Fault Tolerance: Persistent memory improves fault tolerance by preventing regressions and repeated mistakes. If a solution or bug fix was discovered in the past, the agents won't "forget" it – they can look it up rather than making the same error again. Patterns: - **Checkpointing:** After each phase or major milestone, the Planner can save a summary of the state (what's been built, known limitations, next steps) in memory. If the system crashes or a new session starts, the Planner can retrieve this and quickly get up to speed. This is a form of recovery checkpoint.
- **Redundancy in Knowledge:** If an agent fails to recall something from conversation, it can query memory explicitly, increasing the chance that critical details aren't lost.
- **Context Limit Mitigation:** As tasks get complex, not everything fits in prompt context. By offloading details to memory files (which can be fetched on demand), the system can avoid context window overflows. This prevents failures due to forgotten context in long sessions. In case of a memory server failure, agents should default to simpler behavior or request human help, but since Basic Memory is local and simple, failures are unlikely (it's essentially writing to local files).

Execution Guidance: 1. **Install/Enable Memory MCP:** Ensure the Basic Memory server is installed and running. (Claude Code can do this via a command: e.g., `claude mcp add "basic-memory"` or using `uv tool install basic-memory` [12], followed by editing Claude's config to add it [13]). Verify that agents can now call the memory tool (e.g., via a test query).
2. **Define Memory Conventions:** Decide on a directory or file structure for notes (Basic Memory by default stores in `~/basic-memory` with Markdown files [14]). Communicate to all agents (in their prompt or via Planner) how to use this. For example, have a convention like *Project Knowledge Base* with sections for requirements, design decisions, bugs, etc. Optionally, the Planner can create an initial index or table of contents in memory.
3. **Integrate into Workflow:** Going forward, after each meaningful action: - Planner writes an entry summarizing the task outcome (e.g. "Completed module X, used algorithm Y"). - Developer writes an entry for any new API or trick learned while coding. - Reviewer writes an entry if a recurring issue was fixed ("Note: Avoid using global variables as per style guide."). These can be done by calling the memory MCP tool (which will append or create notes). The agents should reference these notes in future planning.

4. **Leverage Memory on New Tasks:** To test, start a new Claude Code session simulating a fresh start. Have the Planner fetch from memory: "What have we built so far?" The memory MCP should return the Markdown summary of Phase 3 work. The Planner can then plan the next task with full context even if the conversation is new. This demonstrates the system's ability to *pick up where it left off* [10].

After Phase 4, the multi-agent developer team has a memory of its own. The groundwork is laid for tackling more complex tasks that require retaining information, such as extended research or multi-session projects.

Phase 5: Enhance Reasoning with Sequential Thinking

Objective & Scope: Integrate advanced reasoning capabilities by using the **Sequential Thinking** MCP server (or a similar chain-of-thought orchestrator). This tool allows the agents (or the LLM behind them) to break down **complex problems into structured steps** and even spawn specialized reasoning sub-agents (Planner, Researcher, Analyzer, etc. internally) for deeper analysis [15]. The scope here is improving cognitive processes rather than adding new domain skills. It will be used across tasks – coding, debugging, planning, etc. – to ensure thorough, stepwise reasoning.

Required Tools/APIs: - **Sequential-Thinking MCP** – an advanced tool that wraps a Multi-Agent System for structured problem solving [15]. When invoked, it coordinates an internal team of specialists (e.g. a Planner, Researcher, Critic, Synthesizer) to examine the query or problem and produce a well-reasoned solution [16] [17]. This effectively upgrades the planner's cognitive strategy.
- Continue using Basic Memory, Filesystem, etc. (These may even be utilized within the sequential thinking steps – e.g., the researcher sub-agent can call the fetch tool if integrated).

Agent Roles: - *No new top-level agents* are added explicitly; instead the **Planner (Meta-Agent)** now has the ability to invoke the Sequential Thinking tool for any task requiring complex reasoning. In effect, the Planner outsources heavy reasoning to the specialized sub-agents inside the Sequential Thinking MCP (Planner, Researcher, Analyzer, Critic, Synthesizer as per its design [15]).
- The existing Developer and Reviewer agents remain, but they too can benefit. For example, the Developer agent could use sequential thinking internally when debugging a tricky error (by calling the tool to reason stepwise through possible causes). - If not using the MCP, an alternative is to explicitly create similar specialized roles in our system (like a *"Solution Analyst"* or *"Critic"* agent). However, using the established MCP server is more efficient since it already implements those roles within its process.

Coordination Model: Centralized with on-demand sub-agent swarm: The Planner still coordinates overall, but when deeper reasoning is needed, it triggers a *decentralized sub-swarm* via the sequential thinking tool. Inside that call, a mini coordinator (Team) delegates to specialist agents (this is handled by the MCP server's internal logic) [16]. The results come back to our Planner, which then proceeds normally. This means our system briefly leverages a *swarm within a swarm*. Coordination remains hierarchical at the top level (our Planner decides when to use the tool and then integrates the outcome). Essentially, we are augmenting the Planner with a powerful problem-solving committee it can summon as needed.

Prompt Engineering: Instruct the Planner agent on *when and how to use sequential reasoning*. For example: - If a task is very complex or ambiguous, the Planner's prompt could have a guideline: "When faced with a complex problem or unclear next step, use the `sequentialthinking` tool to reason it out step-by-step." - Ensure the Planner knows the tool name/trigger words to invoke it. The MCP may be used automatically by

recognizing the word “think step by step” or explicitly via a command. We can include in the Planner’s persona: “Tool: SequentialThinking – use this to break down problems or when needing multi-step reasoning.”
- Additionally, fine-tune how results are used: the Planner should parse the output from the sequential thinking tool (which might be a JSON or structured response with analysis [18]) and then translate that into concrete tasks for the Developer or decisions for the plan. - Example usage in prompt: *Planner:* “I’ll now call the sequential thinking assistant for a detailed analysis.” (The tool runs, returns findings such as potential solutions or a plan refinement.) The Planner then: “Based on the analysis: [incorporate result].” Prompt templates for this interaction might be placed in CLAUDE.md for consistency.

Memory Integration: The sequential steps and conclusions generated should be saved to memory for future reference. For instance, if the system used sequential thinking to debug an issue, the key points of that debug (the root cause, the solution) should be noted in the knowledge base for future agents to recall. This ties Phase 5 back into Phase 4’s memory – ensuring that even though a complex reasoning was done dynamically, its fruits persist. The Planner or the sequential tool’s output handler can be tasked with writing a summary to memory (perhaps tagging it as a **Research Analysis** or **Problem Breakdown** entry).

Fault Tolerance: Sequential thinking inherently improves fault tolerance by **actively revisiting and revising reasoning steps** [19] . Patterns and benefits include: - **Error Anticipation:** The Critic role within the sequential thinking tool will question and evaluate intermediate results, catching logical errors early [15] . This means if an approach is flawed, the system can backtrack or branch out (the tool supports branching alternatives [20]). Our Planner benefits by receiving a well-vetted plan rather than a naive one.
- **Complex Problem Decomposition:** Problems too difficult for one agent to solve in one go are automatically split. Even if one specialist agent fails to find something (e.g., the internal Researcher can’t find info), the Coordinator agent inside sequential-thinking can decide to try a different route or ask for a refinement [21] . This adaptability prevents the whole system from getting stuck on one approach.
- **Resource Usage Safety:** We should note that sequential thinking uses multiple agents and thus more tokens and time [22] . To remain fault-tolerant in resource usage, perhaps configure the tool’s parameters (like limiting depth or iterations for extremely hard problems to avoid infinite loops). If the tool does hit a limit or fails to converge, the Planner agent can detect that (lack of a clear answer) and either retry with different settings or ultimately log an error and proceed with a best effort plan. Essentially, have a timeout or failure-handling for the tool: *If sequential-thinking doesn’t return a useful result, Planner falls back to simpler reasoning or requests human input.*

Execution Guidance: 1. **Enable Sequential Thinking MCP:** Install or activate the `sequentialthinking` MCP server (if not pre-installed). This might involve a command like `uv tool install mcp-server-mas-sequential-thinking` and adding it to Claude’s config with a name (e.g. “mas-sequential-thinking”) [23] . Make sure any required API keys or environment variables for that server are set (if it uses an external LLM or search, configure as needed [24] [25]).
2. **Test the Tool:** Do a quick trial call from an agent to verify the tool works. For instance, prompt the Planner (in a dev conversation) to use sequentialthinking on a simple logical puzzle and observe that it returns a structured breakdown. This helps familiarize how results are formatted.
3. **Integrate into Planning Loop:** Modify the Planner’s logic: when a new high-level task arrives or if the Planner is stuck, it should call the sequential tool. In a Claude Swarm setup, you might allow the Planner to call it as an MCP action (like how it would call any command-line tool). If using a custom orchestrator, embed a step: *if task complexity > threshold or tag requires research, use sequentialthinking.*
4. **Consume Results:** After the tool finishes, the Planner receives something like a JSON containing the

specialists' outputs and the Coordinator's synthesized answer [26]. Parse this: perhaps extract the refined plan steps from it, or the key insight (the tool might return a recommended next step or solution approach). The Planner then continues the normal process – e.g. assigns tasks to Developer as per the refined plan.
5. **Logging:** Ensure that the outcome is logged. Perhaps the Planner writes an entry to memory "Used sequential thinking for X, outcome was Y" for transparency. Also consider printing a console message or comment in the code to trace that this happened, useful for debugging the agent system itself.

By completing Phase 5, the agent system attains a high level of cognitive ability. It can methodically reason through challenges using an embedded multi-agent analytic process. With robust planning, memory, and self-checks in place, the foundation is now strong enough to venture beyond pure development tasks into research and other domains.

Phase 6: Expand into Research Assistant Capabilities

Objective & Scope: Augment the system with an **Information Research** capability. In this phase, we introduce a **Researcher Agent** that can gather data from the internet and provide summaries or analysis for the rest of the team. The system thus evolves from a pure coding assistant to a general *Research Assistant*. The scope includes web searching, data retrieval, and summarizing content relevant to tasks (e.g. finding documentation, market info, user questions, etc.).

Required Tools/APIs: - **Web Fetching MCP** – A tool such as the *Fetch MCP Server* that allows agents to retrieve web content (HTML pages, APIs, etc.) and convert it to text/markdown [27]. This is critical for enabling internet access. The fetch tool can fetch URLs and return site content, with support for truncation and pagination [28].
- **Web Search API** (optional) – If direct URLs are not known, a search MCP or API (e.g. using a search engine) could be integrated to find relevant links. If not available, the user might manually provide some URLs to fetch.
- **basic-memory** – continue using memory to store retrieved knowledge and references for future use (like an internal encyclopedia of findings).

Agent Roles: - *Researcher Agent:* A new specialist role focused on external information gathering. It can take queries (from the Planner or other agents) and use tools (Fetch, etc.) to get answers. For example, if the Developer or Planner asks, "Find the latest best practices for X," the Researcher will perform a web search or fetch relevant pages, then return a summary or the raw content as needed. It should also store key information to memory for later reference.
- *Planner (Meta-Agent):* Now also acts as a **project manager** that knows when to involve the Researcher. If a task requires knowledge beyond the system's current memory (e.g. implementing a feature using a new library, or analyzing a topic), the Planner will prompt the Researcher agent to gather that info. - *Developer & Reviewer:* These remain, but now can indirectly benefit from research. For instance, the Developer can ask the Researcher (via Planner) for documentation on an API it's trying to use. The Reviewer might ask for external best practices if uncertain about a code style decision. These communications would typically route through the Planner to keep things organized.

Coordination Model: Centralized with one more agent: The Planner remains the central coordinator and now has multiple "employees": Developer, Reviewer, and Researcher. The Planner delegates research tasks to the Researcher similar to how it delegates implementation to Developer. This can happen in parallel to development work or as a prerequisite to it (e.g. Planner might pause coding until research is done). The

coordination is still primarily hierarchical (Planner decides when research is needed and integrates the results). However, we could allow more direct queries: e.g., the Developer agent, noticing a knowledge gap, could directly trigger the Researcher via some protocol (in Claude Swarm, possibly Developer could have a connection to Researcher as well). But to avoid chaos, it's cleaner if the Planner mediates: Developer notifies Planner "I need info on X," then Planner tasks Researcher and relays back the answer. This keeps a clear chain of command.

Prompt Engineering: - Researcher's prompt: "You are a **Researcher AI** with access to web tools. Your job is to fetch and summarize information from the internet or documentation upon request. Provide concise, relevant findings, and cite sources if needed." Include guidance for using the fetch tool: e.g. *"Use the* `fetch` *command to retrieve webpages. For example,* `fetch(url=\"http://...\")` *will get page content. Use* `max_length` *and* `start_index` *if content is large."* (This reminds the agent it can paginate large results [28]). Also caution about irrelevant data – the Researcher should filter out noise (maybe using tools or by applying logic like reading HTML via readability mode [29] to get clean text).
- Planner's adjustments: The Planner's instructions should now include when to delegate to the Researcher: *"If a task requires external information (current data, documentation, etc.), formulate a clear query and ask the Researcher agent to gather that info."* Also instruct Planner to incorporate research results into planning (e.g., after Researcher answers, update the task plan accordingly).
- Cross-agent communication prompts: The system should have a protocol for how research queries are phrased and answered. One approach: the Planner sends a natural language query to Researcher (e.g. "Find latest user statistics on topic Y") and the Researcher responds with either a summary or raw content plus summary. The prompt for the Researcher could explicitly mention: *"Output a summary of findings and save details to memory if useful. If multiple sources are needed, fetch them one by one."* Encourage summarization to avoid flooding the Planner with too much data (the Researcher can store raw data in a memory file and just give the highlights to the Planner due to token limits).

Memory Use: Research results should be stored in Basic Memory for future reuse. For example: - After fetching a page or data, the Researcher agent can save the cleaned content or a summary in a memory file labeled by topic. This way, if later a similar query comes up, the agents can search the memory first (avoiding redundant web calls). - Use cross-referencing: if the Researcher finds an important piece of info (say a specification for a crypto algorithm in a later phase), that can be linked in memory so other agents can quickly navigate to it. - The memory also serves as a trace of what sources were consulted (which aids transparency and future auditing). Possibly, maintain a bibliography section in memory listing URLs and key points. - If memory grows large, consider summarizing older entries or categorizing them to keep retrieval efficient (maybe the Knowledge Manager agent from Phase 4, if implemented, could help by condensing older research notes).

Fault Tolerance: Introducing a research agent opens the system to real-world data variability and potential errors (like network issues, misleading data, etc.). We implement patterns to handle these: - **Alternate Source/Redundancy:** If the Researcher cannot find info on the first attempt (e.g., a fetch fails or returns no relevant content), it should have a strategy to try alternative sources. This could mean using a different search query or checking an archived source. The Planner can detect an empty or low-confidence answer and prompt the Researcher to try again or broaden the search query. - **Data Validation:** The Reviewer agent (or possibly the Planner) could cross-verify critical research results. For example, if the Researcher provides a numeric statistic that seems crucial, the Planner might ask the Researcher to fetch a second source for confirmation. This redundancy ensures reliability of info. It's akin to having multiple analysts confirm a finding, improving correctness (and fault tolerance against incorrect info).

- **Tool Error Handling:** If the Fetch MCP fails (e.g., network down or robots.txt blocking as the tool might obey robots by default [30]), the Researcher should handle the error by either waiting and retrying, or using an override (the fetch tool can ignore robots.txt if configured [30]). The agents should be instructed not to give up on first failure. Possibly maintain a retry count and escalate to Planner if all retries fail (Planner might then decide to skip that info or ask human). - **Token Limits on Fetch:** The fetch tool may truncate content to a max length [31]. The Researcher agent should detect if content was cut (perhaps by presence of a marker or by expecting more). If so, it can use the `start_index` parameter to fetch the next portion [28]. This loop continues until the needed content section is retrieved. This ensures the system doesn't silently miss information due to length limits. - **Citing and Tracking:** To avoid confusion and ensure provenance, the Researcher's outputs should include source identifiers (like URLs or titles). This makes it easier later to trace where information came from if something seems wrong. It's also a good practice for the Planner/Developer to include references in code comments or reports they generate, which improves trustworthiness.

Execution Guidance: 1. **Add Researcher to the Team:** Extend the multi-agent configuration to include a "researcher" instance. In a Claude Swarm YAML, for example, add:

```
researcher:
  description: "Internet research specialist, finds and summarizes external
information"
  directory: ./research  # (optional folder for research notes)
  model: <model_type>
  allowed_tools: [WebSearch, Fetch, Read, Write]
# allowing internet access and file ops for notes
```

Connect the researcher to the Planner (main) so the Planner can send queries. Possibly also allow Planner to directly read any files the Researcher writes (so Planner can access fetched content if needed).
2. **Setup Internet Tools:** Ensure the fetch MCP is configured in Claude Code's settings (as per Phase 6 tools). For instance, add to settings:

```
"mcpServers": {
    "fetch": { "command": "uvx", "args": ["mcp-server-fetch"] }
 }
```

and allow the researcher agent to use `mcp.fetch.fetch` or similar. Test that the researcher can call `fetch` by prompting it with a simple URL. If a Search MCP (like a Google Custom Search API integration) is available, configure similarly. If not, the researcher might operate with known URLs or a provided list.
3. **Workflow Integration:** When a new task or question arises that requires outside info, the Planner should create a sub-task for research. This might be manual in the prompt (the user can instruct "now need to research X"), but ideally the Planner decides autonomously. For example, if Developer needs an unfamiliar library, Planner recognizes the lack of info and triggers a research task: "Researcher, find documentation for library Z." The Planner then waits or does other work in parallel. The system can be designed such that the Planner doesn't stall everything; if possible, other agents could continue local tasks while research is ongoing (this would require concurrent execution capabilities – in Claude Swarm multiple agents do run in parallel, communicating via messages). If parallelism is complex, it's fine to do it sequentially: the Planner

pauses new coding tasks until the Researcher replies.
4. **Using Research Results:** Upon receiving the Researcher's findings, the Planner or Developer integrates the knowledge. For instance, if the Researcher returns a summary of an algorithm, the Developer can use that to implement code. The Planner might update the plan based on new insights (maybe a task turns out to be harder or easier given info). All this should happen in a conversation format: Researcher says "Here is info on X...", Planner says "Thanks, I'll proceed with this knowledge...". Optionally, for traceability, the Planner could instruct the Developer to include a comment in code like `// Implemented based on [Source Name]` so that later if something is off, one knows where it came from.
5. **Testing Phase 6:** To validate this integration, try a scenario requiring research. For example: "Add a feature that uses OAuth2 authentication." Likely the system doesn't have all details in memory. The Planner should ask the Researcher to gather documentation on OAuth2. The Researcher fetches, say, *RFC docs or a tutorial*, then summarizes key steps. The Developer then uses those to implement code. The Reviewer checks the code possibly referring back to the summary for verification. If the code is wrong, maybe the Reviewer asks for more info via Planner->Researcher (e.g. "Check if this approach is secure"). This test ensures the loop between coding and researching works seamlessly.

Phase 6 transforms the agent assembly into a true research assistant capable of pulling in fresh information from the outside world [32] [33]. Now the system is prepared to tackle domain-specific analysis, such as the upcoming crypto analysis scenario.

Phase 7: Form a Crypto Research Analyst Swarm

Objective & Scope: Specialize the research capability into a **Crypto Analysis Swarm** – a set of agents focusing on cryptocurrency data and analysis. The objective is to have multiple agents concurrently analyze various facets of the crypto domain (market data, news sentiment, technical indicators, etc.) and aggregate their insights. This phase demonstrates scaling the research team into a domain-specific *swarm* that can provide deep analysis. The scope includes real-time data retrieval (prices, trends), multi-source news scanning, and collaborative analysis, culminating in a comprehensive crypto report or answer.

Required Tools/APIs: - **Crypto Data API Access:** A way to fetch current market data (prices, volumes, charts). If an MCP server exists (e.g., a CoinGecko or Binance API plugin), use it. If not, the **Fetch** tool can retrieve public API endpoints or scrape a website (for example, hitting a REST URL like `api.coingecko.com` for price data, or fetching a coin's page). The system might need to parse JSON – possibly done by the LLM itself or using a parsing tool.
- **News Feeds:** Continue using **Fetch** for news articles or forums (e.g. fetching RSS feeds or specific news sites for crypto). Could integrate a specialized news MCP if available, but not required if fetch works.
- **Data Analysis Tools:** Optionally, if heavy numeric analysis is needed, an MCP like a Python runtime or a small pandas script could be used via a Bash tool to crunch numbers (this is advanced; the LLM can also do basic analysis itself).
- **basic-memory:** crucial for storing historical findings, past market reports, and keeping track of trends observed over time by the agents.

Agent Roles: We will create a *swarm of specialized analyst agents*, for example: - *Market Analyst:* Focuses on quantitative data (prices, metrics). Uses APIs or fetch to get current prices, historical charts, on-chain data if needed. Can compute trends or detect anomalies. - *News Analyst:* Scans recent news headlines, social media (Twitter, forums) to gauge sentiment or major events affecting the market. Summarizes key points (hacks, regulatory news, etc.). - *Technical Analyst:* (optional) Interprets price charts or on-chain metrics in terms of

technical analysis – e.g., identifies patterns like “bullish breakout” or unusual wallet movements. (This agent might rely on Market Analyst’s data or fetch analytic articles). - *Crypto Research Coordinator:* Acts as a mini-Planner for the crypto swarm. This could be either the main Planner itself taking on this role for crypto tasks, or a dedicated agent that aggregates the findings of the analysts. The coordinator’s job is to ask each specialist for their insight on a query (like “Should we invest in coin X now?” or “What is the outlook for Bitcoin this week?”) and then synthesize a coherent answer or report.

In summary, within the crypto domain, we have a *decentralized swarm* of analysts working in parallel, managed by a coordinator agent. This mirrors a team of domain experts collaborating on a problem.

Coordination Model: Hybrid (Swarm with internal coordinator): When a crypto-related query or task arises, the system essentially spins up the crypto swarm. The Planner (global meta-agent) will likely hand off the query to the **Crypto Research Coordinator** agent, effectively delegating responsibility for this domain. The Crypto Coordinator then acts as leader for the subset of agents (Market, News, Technical). This is analogous to a hierarchical multi-level structure: the main Planner delegates to a crypto-team lead (the coordinator), who then runs a decentralized analysis among the specialist agents [34]. The specialists can work largely in parallel, each fetching data and doing analysis simultaneously. They might share intermediate findings amongst themselves if useful (e.g., News Analyst might inform Technical Analyst of a breaking news that could impact technical patterns). The Crypto Coordinator waits for all agents to report back, then compiles their insights into one output for the Planner. This model provides both parallelism and organized synthesis.

Prompt Engineering: - Define each crypto agent’s persona clearly: - Market Analyst prompt: “You are a **Crypto Market Analyst**. You specialize in quantitative data: prices, market cap, trading volumes, historical trends. You have access to crypto APIs. Provide numerical insight (e.g., ‘Bitcoin is up 5% today, breaking a 30-day high’) and interpret what that might mean.” Encourage it to be precise with numbers and possibly use charts (if possible to output ASCII charts or descriptions). Also instruct how to fetch data: e.g. *“Use the fetch tool on URLs like* `https://api.coingecko.com/api/v3/simple/price?ids=bitcoin&vs_currencies=usd` *to get current prices (the response will be JSON). Extract the relevant fields.”* This gives an idea of how to get data.
- News Analyst prompt: “You are a **Crypto News Analyst**. You monitor news sites, social media, and forums for the latest developments in crypto. Use the web to find recent news on relevant coins or the industry. Summarize the top 2-3 news items and assess the overall sentiment (positive/negative).” Guide it to focus on news from the last day or week for freshness. Possibly list a few example sources (CoinDesk, CoinTelegraph, etc.) for it to fetch.
- Technical Analyst prompt: “You are a **Technical Analyst**. You examine charts and blockchain data for patterns. Given price data (from Market Analyst) and news context, infer technical signals (like trend directions or support/resistance levels). If you have price history, you might mention known patterns (e.g., ‘a golden cross formed on the 50-day moving average’). If on-chain data is available (like large transfers or address growth), interpret those too.” This role is tricky for an LLM without built-in numeric computing, but it can qualitatively analyze given data. Possibly instruct it to fetch an analysis from an expert site if needed (like a blog analysis) and then summarize.
- Crypto Coordinator prompt: “You are a **Crypto Research Coordinator**. Your job is to aggregate insights from various crypto analysts (market, news, technical) to answer high-level questions or produce reports. When a question comes in, ask each specialist for their input, wait for their responses, then compile a comprehensive answer. Ensure consistency and that conflicting views are addressed. Your output should read as a cohesive analysis, citing data or news where appropriate.” The coordinator should also instruct

specialists on what exactly they should focus on if the query is specific (e.g., if the question is about a particular coin, the coordinator directs all analysts to focus on that coin).

- Emphasize collaboration: The specialists' prompts can mention, *"Communicate any important findings to the coordinator. If you need data another analyst has, you can ask them via the coordinator."* The coordinator's prompt can include, *"Gather input by explicitly addressing each analyst agent, e.g., '@MarketAnalyst, provide latest price trend; @NewsAnalyst, any breaking news?'"* (This works if the underlying system supports addressing or if using a shared chat - in Claude Swarm, the lead can send messages to specific connections or broadcast).

Memory Use: The crypto swarm should make heavy use of memory for domain knowledge: - Maintain a **Crypto Knowledge Base** file. This could include background info like definitions (what is Bitcoin, Ethereum, etc.), historical notes (e.g., "Last week BTC was at $X, up from $Y a month ago"), and past analyses. The analysts should reference this to avoid repeating explanations and to maintain continuity. - The Market Analyst can log daily snapshots to memory ("On 2025-09-09: BTC \$25k, ETH \$1.6k, volatility high due to event X"). Over time, this builds a historical record the system can use to identify trends without fetching large historical data repeatedly. - News Analyst can similarly keep a running log of major events with dates. - This memory will make the swarm smarter over time – e.g., it might recall "China announced a crypto ban last month" without re-fetching it, when analyzing current market sentiment. - Also, memory helps in collaboration: one agent can write something that another reads. For instance, Market Analyst fetches raw data and stores it in `market_data.json` or memory; Technical Analyst can retrieve that rather than re-fetching. (However, using memory as a real-time communication might be slow; direct messaging via the coordinator is more immediate. Memory is more for persistent knowledge.)

Fault Tolerance: A multi-agent swarm analyzing real-world data needs robust error handling: - **Parallel Redundancy:** Since multiple analysts look at overlapping domains, if one fails or misses something, another might catch it. For example, if the Market Analyst fails to fetch a price due to an API error, the Technical Analyst might still glean price info from a news article (or vice versa). The Coordinator should compare inputs: if one agent's report is blank or clearly flawed, it can prompt that agent to try again or flag that part as uncertain. - **Timeouts and Time Management:** Each analyst's work (especially fetching web data) could take variable time. The Coordinator should be prepared to wait for all agents, but also not wait indefinitely. If one agent is unresponsive (maybe a tool hung), the Coordinator might proceed with available info after a timeout and note that one perspective was missing. Alternatively, implement a mechanism where the Coordinator pings any slow agent: "@MarketAnalyst, are you still gathering data? If facing issues, respond with what you have." This ensures the system doesn't deadlock. - **Data discrepancies:** It's possible two agents provide conflicting information (e.g., Market Analyst says "Price up 5%" and News Analyst found a headline "Market crashes..."). The Coordinator (or possibly a Critic agent if we add one here) should identify contradictions. Fault tolerance here means the system should not produce an incoherent analysis. The Coordinator might ask for clarification: e.g., if such a conflict arises, double-check: "MarketAnalyst, confirm the time frame for +5%. NewsAnalyst, is the 'crash' news from today or old?" By cross-verifying, the agents can resolve whether perhaps the news was about a specific altcoin crashing while overall market is up, etc. This approach of questioning inconsistency acts as a fault tolerance in reasoning. - **Continuous Update Handling:** The crypto domain changes rapidly. If the analysis takes some time, data fetched at start might become stale. To mitigate this, for long-running tasks, maybe have Market Analyst fetch near the end again or monitor changes. However, in practice our tasks are likely short enough. If we were generating a report that takes hours, we'd need periodic refresh. We won't detail that unless needed. - **Safety Checks:** Crypto info can be speculative. If the system is to make decisions (like investment advice), it should include caveats. Fault tolerance in this sense is ensuring the system doesn't output dangerously confident but

possibly wrong advice. The Coordinator or an added *Risk Analyst* agent could ensure to include risk warnings or double-check extremely critical outputs. This is more of a content safety/quality measure.

Execution Guidance: 1. **Configure Crypto Agents in Swarm:** Expand the swarm YAML or config to include `crypto_coordinator`, `market_analyst`, `news_analyst`, `tech_analyst` as instances. E.g.:

```
crypto_coordinator:
  description: "Lead crypto analyst coordinating market, news, and technical
teams"
  model: <model>
  connections: [market_analyst, news_analyst, tech_analyst]
  allowed_tools: [BasicMemory, Write]
market_analyst:
  description: "Cryptocurrency market data specialist"
  allowed_tools: [Fetch, BasicMemory]  # possibly Bash if using curl for APIs
news_analyst:
  description: "Cryptocurrency news and sentiment specialist"
  allowed_tools: [Fetch, BasicMemory]
tech_analyst:
  description: "Cryptocurrency technical analysis specialist"
  allowed_tools: [Fetch, BasicMemory]
```

Here, `crypto_coordinator` is the main for this sub-swarm. You might integrate this into the overall swarm by making the global Planner connect to `crypto_coordinator` as needed (or simply have the Planner treat it as a callable module via some prompt). In Claude Swarm, one could keep these separate or part of the same YAML with the main lead having a connection to crypto_coordinator when needed.
2. **Domain Knowledge Setup:** Before running a real query, seed the memory with any essential background or ensure the agents have the necessary background. For instance, provide a list of top cryptocurrencies and their symbols, or an explanation of common terms (bullish, bearish, etc.) in the knowledge base. This can be done by writing to `CLAUDE.md` or memory files. The agents will then not waste time explaining basics to each other.
3. **Initiating a Crypto Query:** When the user or system asks a crypto-related question (e.g., "Analyze Ethereum's outlook for next month"), the global Planner should recognize this is a crypto domain query and delegate it. This can be achieved by pattern matching or by user explicitly addressing the crypto team. Assuming Planner knows to hand off, it would forward the query to `crypto_coordinator` agent. (In practice, this might be done by the user message being directed to that agent or the Planner rephrasing it as a request). The Crypto Coordinator then kicks off its internal workflow: it asks Market, News, Tech agents for input. 4. **Parallel Analysis:** Each specialist agent uses `fetch` or tools to gather their info. For example, Market Analyst might call a price API and parse JSON. (The LLM can parse JSON text; instruct it to look for keys in the response and convert to numbers.) News Analyst fetches the latest articles from a news site's RSS or homepage (maybe using `fetch(url="https://news.site/latest-crypto")`). Tech Analyst might fetch a site or simply await data from Market to analyze. Ideally, these happen in parallel in Claude Swarm (since each agent is an independent process once triggered). If the environment doesn't allow true parallel, the Coordinator can simulate it by polling each agent sequentially but conceptually we consider it parallel.
5. **Aggregation:** The Crypto Coordinator collects responses. It should make sure it got something from

each, or note if one failed. Then it composes a report. The Coordinator's output to the Planner could be something like: - "Market: ETH is up 10% in the last week, with volume increasing. News: Positive sentiment due to successful upgrade, but some regulation concerns in US. Technical: Chart shows strong upward trend, but approaching a resistance at \$X. Overall: Outlook is cautiously optimistic... (with caveats)." This final result is what the user or main system sees. The Coordinator should also perhaps write this report to memory (for archival). 6. **Feedback to Specialists (if needed):** If the Planner or user has follow-up questions (like "Why is volume increasing?"), the Coordinator can loop back with the specialists or use memory if that info was captured. The swarm can handle iterative queries by maintaining context. 7. **Testing & Validation:** Run a test scenario fully: e.g., ask "Summarize the current state of the crypto market" or something similar. Ensure each agent produces plausible output and the coordinator merges them without contradictions. Adjust prompts if one agent tends to dominate or if outputs are unbalanced (maybe ensure Market gives short numeric output and doesn't try to analyze news, etc., to keep roles clear). Confirm that if one agent fails (simulate by blocking its tool), the coordinator still returns something and mentions the gap. This fine-tuning ensures the swarm is robust and coherent.

By the end of Phase 7, the system can field complex analytical tasks in the crypto domain by leveraging a swarm of focused agents. This showcases scalability – we have taken the research capability and scaled it horizontally (multiple agents) and vertically (with a sub-coordinator) to handle a specialized, complex domain. The next phases will extend this concept to other domains (like social media) and ensure the entire multi-agent organization functions as a whole.

Phase 8: Establish a Social Media Marketing Team

Objective & Scope: Grow the system into a **Social Media Marketing team** that can create and manage content based on the outputs of the research and development teams. The objective is for the agents to generate social media posts, blogs, or marketing strategies autonomously, using information from previous phases (like code features from dev, insights from research/crypto analysis). The scope includes content planning, copywriting, editing, and scheduling (or at least simulating scheduling) of posts.

Required Tools/APIs: - **Social Media API Access (optional):** If actual posting is desired, an API like Twitter's or a scheduling tool's API could be used via HTTP requests. However, given the complexity and to avoid external dependencies, we might simulate posting by writing content to files or printing them. - **Sentiment/ Tone Analyzer (optional):** Could use an MCP or internal model capability to check tone/quality of content. (Alternatively, the Editor agent can handle this by prompt). - **Image Generation (optional):** For a full marketing team, one might include an image design agent using a DALL-E or Stable Diffusion API. This is advanced and not requested explicitly; we will omit unless easily integrated (most likely omitted in spec for brevity). - **basic-memory:** continue to use for storing content calendars, campaign ideas, brand guidelines, etc. The marketing team will rely on knowledge stored about the product or topic they are promoting.

Agent Roles: Form a mini marketing department: - *Content Strategist:* This agent plans the content: it decides what posts or articles are needed and in what format. For example, given a goal ("promote product X" or "explain research Y to a broad audience"), the Strategist comes up with a campaign idea or schedule (e.g., a series of tweets, a blog post, a LinkedIn article). It sets the tone and target audience. It might create an outline or creative brief for each piece of content. - *Content Writer:* This agent writes the actual content (tweets, blog sections, etc.) following the Strategist's plan. It should be skilled in persuasive and clear writing. If multiple formats are needed (short tweet vs long article), it can adapt style accordingly. - *Editor/ Proofreader:* Reviews and refines the content written by the Writer. Ensures it's error-free, on-message, and

high quality. Also checks that the content aligns with any given guidelines (e.g., word count, tone, compliance). Possibly also optimizes for SEO if relevant (for blog content). - *Social Media Manager:* (optional but could be included) Simulates scheduling and posting. This agent would decide the timing of posts and ensure consistency (e.g., not posting everything at once, choosing best times). It could also simulate engagement by phrasing content in a way likely to attract attention. In an actual deployment, this agent could call the Twitter API to post, but here it might just log the scheduled times or output an example schedule.

In addition, the marketing team will interface with the rest of the system: - They will rely on input from the Developer team (to know product features or technical details worth marketing) and from the Research/Crypto teams (to extract key insights to share). However, instead of direct agent-to-agent communication, likely the Planner or a top-level coordinator will funnel information to the marketing agents. Alternatively, we might designate the Content Strategist as the one who requests needed info (like asking the crypto team for a key takeaway to turn into a tweet).

Coordination Model: Centralized (Marketing Lead) – Similar to the crypto swarm, we can have a *Marketing Coordinator* (the Content Strategist can act as this lead) who receives a marketing goal and coordinates the writer, editor, etc. However, in content creation, a more linear pipeline is typical: Strategist -> Writer -> Editor. So we might implement this as a sequence rather than parallel: the Strategist forms a plan, the Writer drafts content, the Editor then reviews it. These could be done with each as separate agents passing the content along. The Social Media Manager (if separate) would come last to compile final approved content and simulate posting. The overall system's main Planner would initiate this pipeline when marketing output is needed (e.g., "produce a tweetstorm about our new research findings").

Alternatively, for simplicity, the Strategist could itself produce outlines and then directly instruct the Writer to fill them out, etc., playing a dual role as coordinator of the writing pipeline. But since we want distinct roles, treating them as separate agents with handoffs is clearer.

Prompt Engineering: - Content Strategist prompt: "You are a **Marketing Strategist AI**. Your goal is to devise effective social media and content strategies to achieve given objectives. You create campaign ideas, choose platforms (Twitter, blog, etc.), and set the messaging direction. Provide clear briefs for content pieces (like 'Tweet 1 should highlight fact A in a humorous tone', 'Blog post should explain B in depth'). Work with the content writer and editor to bring this to life." - Content Writer prompt: "You are a **Content Writer AI**. You craft engaging and grammatically correct content as per the Strategist's brief. Adjust your tone and style for each platform (friendly for Twitter, informative for blogs, etc.). After drafting, hand it to the Editor for review. Be creative but stay factual (use info provided by research or dev teams accurately)." - Editor prompt: "You are an **Editor AI**. Your task is to review and refine content drafts. Correct any spelling/grammar errors, improve clarity and tone, and ensure it meets the brief. If something is off-message or inaccurate, note it and suggest changes. You have authority to directly edit the text. Once content is polished, approve it for publishing." - Social Media Manager prompt (if used): "You are a **Social Media Manager AI**. Once content is ready, you decide on scheduling and final touches for posting. Ensure posts fit character limits, include relevant hashtags or mentions, and schedule them at optimal times. You don't create content from scratch but you finalize it for publication. Provide a schedule or posting confirmation."

- Also include **brand/persona guidelines** in prompts or the CLAUDE.md if relevant. For example, define the tone (scientific vs playful) depending on context. These guidelines help the writer and editor maintain consistency.

- The Strategist should be prompted to ask for information if needed. If, say, the objective is to promote a research finding, the Strategist might query the research memory or even ask the Researcher agent for a layman explanation of the key point to ensure understanding. In other words, instruct the Strategist: "If needed info isn't readily available (like details about the product or finding), request it from the appropriate source (development notes or research analysis)." The system's Planner can facilitate this data flow (e.g., providing the Strategist with a summary from memory of what needs promoting).
- Use chain-of-thought in writing and editing: e.g., the Writer might generate multiple variations if uncertain and choose the best. The Editor might think step-by-step through a checklist (spelling, tone, factual correctness) to not miss anything. These processes can be encouraged with prompt hints like "Check for: 1) grammar, 2) tone, 3) accuracy."

Memory Use: The marketing team will benefit from memory in several ways: - **Brand and Style Guide:** Store a style guide in memory (or CLAUDE.md) that the writer and editor can reference. For instance, memory can contain the brand voice description: "Our voice is professional yet approachable. We use humor sparingly. Avoid jargon when targeting broad audience," etc. It may also include approved phrases or hashtag strategies. - **Content Calendar:** The Strategist or Social Media Manager can maintain a calendar or list of content produced and scheduled in memory. This prevents duplicate posts and helps long-term planning. - **Material Repository:** If the Developer team produced technical documentation or the Research team produced a report, those can be in memory. The Writer can pull exact quotes or facts from there to ensure accuracy. - **Campaign Retrospective:** After content is "published" (simulated), notes on how it might perform or any lessons can be stored for future improvement. (E.g., "The tweet about X got good engagement – presumably; note to use that format again." This is speculative without real feedback, but could be simulated or assumed knowledge.) - Essentially, memory in marketing ensures consistency and that each piece of content aligns with a unified strategy rather than existing in a vacuum.

Fault Tolerance: A marketing pipeline can fail in quality or coordination; here's how to mitigate: - **Quality Control Loop:** The Editor agent provides a fail-safe that catches mistakes from the Writer. If the Writer misunderstood the brief or inserted incorrect info, the Editor should catch it. If the Editor finds a major issue, it might request clarification from the Strategist (creating a feedback loop). For example, Editor might ask "This tweet references a statistic not provided; can Research team verify this?" This triggers the Strategist or Planner to fetch the needed correction. This ensures erroneous content isn't "published." - **Approval Gate:** Only after Editor approval does content go out. If we include the Social Media Manager, that agent should double-check lengths and format. If a tweet is over 280 chars, the Manager should either trim it or send it back to Writer for revision. This additional gate catches any format-specific errors. - **Deadline Handling:** If multiple content pieces are being produced (say 5 tweets), and one is lagging due to complex info, the Strategist might prioritize and have at least some content ready. Perhaps not critical in an automated context, but the idea is if one agent stalls (Writer stuck on one post), others could still proceed with their tasks or the system can publish what's ready. - **Tool Failures:** If this team were to interface with actual APIs for posting, any failure (like network issue or API error) should be caught by the Social Media Manager. Since we likely simulate, not an issue here. If we did implement posting via fetch (for example, making a POST request to Twitter API), we'd include error-checking and retries in that agent's process. - **Reputation/Safety Check:** Marketing content going out in public should be checked for appropriateness. The Editor should ensure nothing inflammatory or sensitive is present. If the LLM tends to output something that could be misinterpreted, the Editor should revise it. This reduces risk of bad publicity (this is more content safety than technical fault tolerance, but crucial in autonomous marketing). Possibly define a "no-go list" of topics or wording in memory for the Editor to cross-check against.

Execution Guidance: 1. **Add Marketing Agents:** Extend the swarm config with `marketing_strategist`, `content_writer`, `editor`, and optionally `social_manager`. Connect the strategist to writer and editor (and manager if present) in sequence. For example:

```
marketing_strategist:
  description: "Marketing strategist coordinating content creation"
  connections: [content_writer, editor, social_manager]
content_writer:
  description: "Creative writer producing content per strategist's brief"
  # no direct connections, it communicates via strategist
editor:
  description: "Eagle-eyed editor ensuring content quality"
  # connect to writer if we allow direct feedback? or just via strategist
social_manager:
  description: "Social media manager finalizing and scheduling posts"
```

In practice, the strategist will send the draft from writer to editor, and then final copy to manager. We might not need direct writer->editor connection if strategist handles the routing (which is simplest: strategist asks writer for draft, gets it, gives to editor, gets edited version, gives to manager). This again is a hierarchical pipeline, so connections can be linear. Alternatively, the Editor could be connected to Writer to send back corrections directly. But let's assume coordinator manages it for clarity. 2. **Workflow Implementation:** When a marketing task is triggered (either by user or by a need arising from research/dev outputs), the Planner delegates to `marketing_strategist`. For example, user says: "Create a Twitter thread explaining our crypto analysis results." The Planner would package the relevant crypto findings (maybe from memory or by having the crypto coordinator's output) and give that as input to the strategist agent: "Objective: Promote these findings: [summary]. Channels: Twitter thread." Then: - The Strategist formulates a plan: e.g., "We will do a 3-tweet thread. Tweet1: highlight main stat. Tweet2: context. Tweet3: call-to-action." It then sends each piece as a brief to the Writer. - Content Writer drafts the tweets as requested and returns them to Strategist or directly to Editor (depending on configuration; we assume via Strategist). - Editor reviews each tweet, fixes phrasing or length. If Editor sees an issue (like a fact that might need checking), it can communicate that. In a simple pipeline, Editor might loop back to Strategist: "Tweet2 claims '10% increase' but our data said 8%. Please verify." The Strategist (or Planner) can then fetch the correct data (from memory or ask Researcher) and update the brief/draft. Then Writer revises Tweet2. - Once Editor approves, the content goes to Social Manager (or if none, Strategist takes final action). - Social Manager schedules the tweets (in simulation, just output the final thread with numbering and maybe suggested posting times). It might append hashtags or shorten links as needed. - The final output from the marketing team could be a file or message containing the ready-to-post content. 3. **Integration with Other Phases:** The Planner or overarching system should integrate the marketing team's activities with the rest: - For instance, after Phase 7's crypto swarm produces an analysis, the Planner might automatically trigger Phase 8: "Now marketing team, create a summary for social media." If fully autonomous, the Planner decides this as a follow-up task. If user-driven, the user might instruct it explicitly. - Ensure that the marketing team has access to the necessary info: perhaps the Planner provides the key points from the crypto analysis to the Strategist. Or the Strategist can query memory where the analysis was stored. We need to facilitate that data flow so the content is accurate. - Also, if the Developer team built a product or feature, Planner could call on marketing to announce it. The memory would have details of the feature for the Strategist to use. 4. **Test Run:** Example test – "Write a LinkedIn post about the new software feature

developed and its benefits." The Developer team's output (feature description) is in memory, the Planner gives it to Strategist, and the pipeline should produce a polished LinkedIn-style post (maybe longer form than tweets). Test another – "Generate a series of tweets sharing today's crypto analysis." See if it produces coherent, correctly sized tweets with maybe hashtags. Adjust prompt if needed (like remind Content Writer of 280 character limit, which Editor/Manager should enforce too). 5. **Review and Iterate:** Verify that the Editor catches errors. Intentionally introduce a mistake (maybe feed a wrong stat) and see if the Editor queries it. If not, strengthen Editor's prompt to be more vigilant about facts. Also confirm the tone is consistent with guidelines. If not, refine the style guide.

By the end of Phase 8, the system can not only develop solutions and analyze information, but also *communicate* those results to an audience in a polished manner. We have effectively created a multi-agent content creation pipeline that works alongside the dev and research teams.

Phase 9: Unified Multi-Team Orchestration

Objective & Scope: At this phase, we focus on **integrating all the specialized teams (Development, Research/Crypto, Marketing)** under a unified orchestrating intelligence. The objective is to enable the system to handle complex projects that span multiple domains (for example, building a product, analyzing the market for it, and promoting it) with minimal human intervention. The scope includes establishing clear interfaces between teams, possibly introducing an overall *Chief Meta-Agent* or improving the existing Planner's capabilities to manage multiple departments. We also ensure that knowledge and goals are shared appropriately across the system.

Required Tools/APIs: No new tools beyond what's integrated already (Filesystem, Git, Basic Memory, Fetch, etc., plus any domain-specific ones from earlier phases). At this point, the toolset is rich. The emphasis is on orchestrating their use: - The **Basic Memory** now contains a wealth of information (codebase docs, research findings, content guidelines). It serves as a central repository accessible to all teams. - All MCP servers from previous phases remain available. We should confirm they can all run simultaneously without conflict (for example, sequential-thinking, fetch, basic-memory can all be active – ensure resource allocation is fine). If any scaling issue (like many MCP processes eating RAM, as some Reddit discussions warned), consider consolidating where possible or using shared instances for common tools to conserve resources [35] [36]. For instance, maybe one fetch server is enough for all agents to use, rather than each launching its own.

Agent Roles and Hierarchy: Now we have a lot of agents. We need a clear hierarchy or network: - Consider appointing a top-level *"CEO Agent"* or *Chief Coordinator* that oversees everything. This could simply be the original Planner meta-agent elevated to a higher role, or a new agent that specifically manages inter-team coordination (the "architect" role mentioned in the Claude Swarm example [37]). Let's say we formalize: - *Chief Meta-Agent (Architect):* Responsible for overall goal management and high-level planning across domains. It receives the ultimate user goal (or sets one itself if autonomous) and devises a master plan. It then delegates sub-goals to domain-specific leads: - Development Lead (could be the Planner from earlier phases or renamed to Dev Lead) - Research Lead (could be the Crypto Coordinator or a General Research Coordinator if non-crypto research tasks as well) - Marketing Lead (Content Strategist as lead for marketing) - Any other future domain leads. The Chief agent monitors progress and makes sure outputs of one team feed into tasks of another appropriately. - *Domain Team Leads:* As identified: - Dev Lead (Planner from phases 2-5) – coordinates Developer & Reviewer. - Research Lead – we have Crypto Coordinator for crypto-specific; if we want general research beyond crypto, maybe the same or an extended role. Perhaps rename Crypto Coordinator to *Research Coordinator*, and it can handle both general research tasks (by using the

Researcher agent or others) and invoke the crypto swarm when needed. In design, maybe keep them separate: a general Researcher agent for non-crypto queries, and the Crypto sub-swarm specifically for finance domain. But that might complicate – perhaps treat crypto as the main research focus for now since it was explicitly mentioned. - Marketing Lead (Strategist) – coordinates writer, editor, etc. - *Specialist Agents:* As previously: Developer, Reviewer, Researcher, Market Analyst, News Analyst, Technical Analyst, Writer, Editor, etc. - Possibly introduce a *Cross-Domain Critic or QA Agent*: This would be a role that looks at the big picture results from each team and checks for consistency or missed opportunities. For example, after all tasks are done, a Critic agent could review: "Does the marketing message accurately reflect the product the dev team built and the insights the research found?" If it finds discrepancies, it flags them. This is an optional advanced role that can further ensure coherence. If included, the Chief meta-agent would invoke this QA agent after major outputs to get an external evaluation. This agent would use memory (to see what dev built, what research found, what marketing wrote) and then produce feedback.

Coordination Model: Hierarchical multi-level: We have essentially a tree of agents: - Chief at root. - Team leads (Dev Lead, Research Lead, Marketing Lead) as immediate children of Chief. - Specialists under each lead (e.g., Developer under Dev Lead, Market/News under Research Lead, Writer under Marketing Lead, etc.). This is very much like the multi-level example from Claude Swarm documentation [34], where an architect coordinates team leads, and each team lead coordinates specialists. Here our "architect" is the Chief Meta-Agent. This structure is scalable and modular – we can add another team (say a *Customer Support team agent* in future) as another branch and the Chief just delegates to it when needed.

Communication flows mostly vertically: top-down goals and bottom-up results. But horizontal communication can happen indirectly via the Chief or memory: - If Marketing needs something from Dev (say a clarification on a feature), it can either request the Chief to get Dev Lead to respond, or look it up in memory where Dev might have documented it. The Chief can act as a message router if needed. - Similarly, if Research finds something that could alter Dev's plan (e.g., discovered a new technology to use), it should inform the Chief, who can then adjust the Dev team's tasks. We might keep a **shared project status** in memory that all leads update (like a high-level progress board), which also helps coordination.

Prompt Engineering: - Chief Meta-Agent prompt: "You are the **Chief Coordinator AI** (akin to a project manager or CEO). You oversee multiple specialist teams (development, research, marketing). Your role is to take high-level objectives and break them into sub-tasks for each team, coordinate their efforts, and assemble the final output. Ensure that all teams are aligned on goals and timelines. Manage interdependencies: e.g., research findings should inform development and marketing; development output should be correctly understood by marketing, etc. If a conflict or inconsistency arises, resolve it by tasking the relevant team to adjust. Always keep the big picture in mind and drive the project to completion." This prompt sets the tone for holistic oversight. It should be armed with knowledge of each team's existence and capabilities (perhaps list the teams and their responsibilities in the prompt for clarity). - Domain Leads prompts: We have some already (Dev Planner, Crypto Coordinator, Marketing Strategist). They might need a tweak to acknowledge the Chief above them: e.g., "You are the Development Lead AI, reporting to the Chief Coordinator. You receive product requirements and you manage the Developer and Reviewer to implement them. Communicate progress or roadblocks back to the Chief. Work with other leads when required (if research info is needed or if marketing needs technical details, provide assistance)." - Similarly for Research Lead: "… Manage research agents to gather required info. Provide synthesized findings to Chief or directly to other teams as appropriate (e.g., share relevant data with marketing)." - Marketing Lead: "… Coordinate content creation. If you need technical or research info, request it from the Chief or relevant lead. Ensure the promotional content accurately reflects the information from dev/research." - Essentially, prompts

should encourage **inter-team communication via the proper channels**. Possibly instruct leads to use memory for sharing knowledge rather than each directly chatting, to avoid chaos. Or if the system allows direct agent-to-agent messaging beyond the tree structure, it should still be controlled (maybe allowed between certain leads if needed). - Emphasize goal alignment in prompts: All agents ultimately share the same overall goal (set by user/Chief). They should be reminded of that, so they act collaboratively, not at cross purposes.

Memory Strategy: With multiple teams, memory management is crucial: - We should organize memory such that it's segmented or at least well-labeled by domain, but still accessible globally. For example, maintain folders or sections: `/memory/dev`, `/memory/research`, `/memory/marketing`, plus a `/memory/global` for project-wide info. - Each team lead/agent writes to their area (Dev team writes design docs, progress updates in dev memory; Research logs sources and findings in research memory; Marketing logs content plans and published content in marketing memory). - The Chief and all leads should have access to the global memory and possibly all sections. If fine control is possible, one could restrict, but probably not necessary since the whole system is cooperating. The Chief can quickly get up to speed by reading memory summaries from each team. - Encourage summarization: To prevent the memory store from growing unwieldy, possibly after a project or major milestone, have each team lead write a concise summary of their outputs to a global summary file. The Chief agent (or a separate summarizer agent) could compile an executive summary for the whole project. This ensures that even if detailed logs are long, there's a high-level snapshot for any new queries or when scaling further. - Memory also serves for retrospective learning: e.g., if a certain approach failed in a past phase, storing that means next time the agents won't repeat it. This is more long-term improvement, but relevant if we consider Phase 10's continuous improvement.

Fault Tolerance: With a large system, faults might be: - **Miscommunication or Lost info:** Mitigated by the Chief keeping an eye and memory logs. If a piece of data doesn't reach a team, the Chief can detect the gap (because it's in memory but not reflected in a team's output). For example, if Research found a crucial insight but Marketing content didn't mention it, the Chief or Critic agent might catch that and loop back. - **Overload or Deadlocks:** Many agents could potentially get into circular waiting (e.g., marketing waiting for research, research waiting for dev, dev waiting for research etc.). The Chief must prevent circular dependencies. It does so by orchestrating sequence: e.g., ensure research tasks happen before dev coding if dev needs that info. Or if parallel, at least make sure each knows what they're waiting for. Possibly maintain a dependency graph internally. If the Chief sees no progress after a while, it should intervene to break deadlock (like ask if someone is waiting unnecessarily and push them to act or drop a dependency). - **Scaling Issues:** More agents mean more token usage and latency. The system should be designed to allow not all agents being active at all times. The Chief should only activate a team when needed. For example, if the current goal is purely research, maybe the dev and marketing agents stay idle (or not invoked) until research finishes. Or if it's coding time, research might pause. This avoids wasteful chatter. Using Claude-swarm, you can start all, but perhaps control by messages. We could implement a simple protocol: the Chief only sends tasks to relevant leads, others remain idle (they could even be configured to only respond when addressed). - **Single Point of Failure (Chief):** If the Chief agent goes wrong, it could misdirect everyone. While we want it autonomous, one idea is to include some self-correction: Maybe the Critic/QA agent or even the team leads collectively could sanity-check the Chief's orders. This is similar to decentralized aspects – we might allow team leads to question the Chief if something seems off. E.g., if Chief tells dev to do something obviously incorrect from a domain perspective, the Dev Lead should respectfully flag it. To implement this, encourage a culture in prompts: "If an instruction from above seems erroneous, politely clarify or double-check." This ensures if the Chief makes a mistake, an agent might catch it (fault tolerance

through **human-in-the-loop analogy, but here an agent is in the loop**). - **Error Recovery:** If one team fails to deliver (say dev team can't implement a feature as specified), the Chief should adjust the plan rather than stall. This might involve scaling down scope or coming up with an alternative path (maybe research an alternative solution). We can design the Chief to be flexible: e.g., include in prompt "If a subtask fails, devise an alternative or consult another team for ideas."

Execution Guidance: 1. **Establish Hierarchy in Config:** If using a unified YAML config for Claude Swarm, structure the instances accordingly:

```
chief:
  description: "Chief coordinator managing development, research, and marketing
teams"
  connections: [dev_lead, research_lead, marketing_lead]
  allowed_tools: [BasicMemory, SequentialThinking]  # can use memory and
advanced reasoning
dev_lead:
  description: "Lead of development team (planner/coordinator for dev agents)"
  connections: [developer, reviewer]  # plus maybe connect to research_lead if
direct queries allowed
research_lead:
  description: "Lead of research team (coordinates researchers/analysts)"
  connections: [researcher, market_analyst, news_analyst, technical_analyst]  #
assuming one team
marketing_lead:
  description: "Lead of marketing team (coordinates content creation agents)"
  connections: [content_writer, editor, social_manager]
# (Other agent definitions as previously set)
```

Ensure all lower agents are connected as needed (some might not need explicit connection if always mediated by leads). This configuration means the Chief can talk to each lead; each lead talks to its sub-agents. If any cross-team comm needed, it can either be Chief relaying or, if we want, we could give leads limited cross-connections (like dev_lead <-> research_lead) to ask directly. But direct cross links might complicate, perhaps better to go through Chief or use memory. 2. **Top-Down Task Delegation:** The process for a given project or query: - The Chief receives the top-level goal from the user (e.g., "Develop feature X and announce it to users, considering market needs"). The Chief should break this into sub-goals: - e.g. Sub-goal1: Research market needs for X (assign to research_lead). - Sub-goal2: Implement feature X (assign to dev_lead, potentially after research). - Sub-goal3: Create marketing content for X (assign to marketing_lead, after dev provides feature details and maybe research provides audience insights). - The Chief then issues these tasks to the appropriate leads, possibly with some sequential order or conditions ("Research team, start on A. Dev team, prepare to do B once A is done."). - The leads then carry out tasks with their teams as per previous phases. - Chief monitors progress via reports (the leads should update the Chief when done or if blocked). This could be done by the Chief polling them or the leads proactively messaging back "Task A completed" with results attached. Possibly using memory as the medium or direct message. - The Chief then triggers the next tasks. For instance, once research is done, the Chief summarizes key findings and passes them to dev and marketing leads: "Dev, here are the insights to consider. Proceed with build. Marketing, begin drafting strategy based on these findings and dev's upcoming features." - The dev lead will

eventually report feature done (maybe with documentation or user benefits described), the Chief forwards that to marketing lead to finalize content. - Finally, the marketing lead reports content ready (maybe with the actual posts). - The Chief compiles an overall deliverable if needed (it might not need to if the end goal was just content or code which are already delivered by teams). If the user expects a single consolidated output, the Chief could produce a summary like: "Project complete. Dev built X (details...), Research found Y (details...), Marketing created Z (see content)." Or deliver the outputs in a structured way (like attachments). 3. **Inter-Team Information Flow:** - Use Basic Memory for handing off information: e.g., after research, the Research Lead could write a summary of findings to a known memory file. The Dev and Marketing leads are prompted to always check memory for relevant updates. The Chief can also explicitly send them the info, but memory ensures nothing is lost. - Example: Research lead writes `research/findings_featureX.md`. Chief (or memory system) notifies Dev lead to read that file for context. Dev lead then knows user requirements or market context to design better. Similarly, dev lead might write `dev/featureX_summary.md` describing what was built, which marketing lead will use. - To enforce, in prompts to leads: "Always read any new notes in the project memory relevant to your task. Likewise, document your outputs in memory for others." - This blackboard approach [38] [39] is reliable and avoids missed messages if an agent was busy. 4. **Testing Integration:** Try a scenario end-to-end: For instance, user request: "We need to create a new analytical dashboard feature for our app, determine if it's viable (market research), implement it, and then announce it on social channels." This touches all teams. The expected sequence: - Chief breaks into research viability (research team), development of feature (dev team), marketing announcement (marketing team). - Research maybe says "Yes it's viable, users want it" plus tips. - Dev builds a simplified version. - Marketing drafts an announcement post. - Chief ensures each step flows. At the end, possibly produce a final brief. - Evaluate if any team seemed idle or confused. If, say, marketing started too early without dev info, maybe Chief should have made it wait or work on preliminary stuff (like start drafting based on assumption, then refine when dev confirms specs). - Adjust prompts or sequence if needed (like ensure Chief explicitly instructs marketing "hold until dev completion, but you can prepare outline" if concurrency is possible). 5. **Multi-Project Scalability:** The system at this point can theoretically handle multiple projects or tasks concurrently by context-switching or with multiple chief instances. We won't detail that, but modular design (teams in isolation except via Chief) allows focusing on one project at a time. If scaling to multiple simultaneous missions, you'd likely spawn separate swarms or have the Chief handle scheduling. But that's beyond scope – just note that adding tasks one after another, memory will keep track of each.

At the end of Phase 9, we have a **fully integrated multi-agent organization**. The developer automation, research swarm, and marketing team operate as parts of one cohesive unit, steered by a central intelligence. The system can autonomously go from problem conception to solution implementation to communication of results, with each expert agent contributing its part.

Phase 10: Continuous Self-Improvement and Scaling

Objective & Scope: In the final phase, the system turns its capabilities inward for **recursive self-improvement**. The goal is for Claude (the AI) to continually refine its own multi-agent processes and incorporate new tools or roles as needed in the future (supporting "current and future MCP servers"). The scope includes implementing feedback loops where the system evaluates its performance, fixes deficiencies, and extends its abilities. We also formalize processes for adding new agents or tools modularly, making the organization truly scalable.

Required Tools/APIs: - Continue using **Sequential Thinking** for reflection and troubleshooting of the system's own workings. The multi-agent analysis tool can be used by a *self-critique agent* to reason about system improvements. - **Git (version control)** – Now not just for code tasks, but for the system to modify its own configuration or code. The developer agent can use git to track changes to the agent framework (like updating YAML configs, prompt files, or even code for custom MCP servers it might create). - **Filesystem** – to allow editing of its own config files or prompt libraries. - **Basic Memory** – to log lessons learned, known issues and resolutions about the agent's performance (an internal "knowledge base" about how the AI operates). - Potentially an **Automated Testing harness** for the AI itself. For instance, have a set of test scenarios saved that the system can run to validate new changes don't break functionality. This could be triggered by a self-improvement cycle (similar to unit tests but for agent behavior). - Any new MCP server can be added when needed. For example, if a future requirement is image creation, one can install an "ImageGen" MCP and add a role for it. The system design should make this straightforward (just a new tool and maybe a new agent role with minimal disruption to others).

Agent Roles: - *Self-Evaluator (Critic) Agent:* Introduce an agent whose job is to monitor and evaluate the performance of the entire agent system. This agent can be triggered after each major project or periodically. It reviews logs, memory entries, and outputs from all phases and identifies inefficiencies, errors, or bottlenecks. For example, it might notice "The Marketing Editor had to correct factual errors frequently – maybe the pipeline could fetch data directly to writer to reduce errors" or "The Research team took too long fetching data – perhaps integrate a faster API or narrow the query." - *Improver (Refactor) Agent:* This agent takes the Self-Evaluator's feedback and implements changes. It could be essentially the Developer agent repurposed to work on the agent system's code/config instead of an external product. For instance, if a new MCP is to be added, the Improver writes the config and updates the YAML, or if prompt tuning is suggested, it edits prompt templates. This agent uses git to safely apply changes (committing new config, possibly branching if needed for testing changes). - *Chief Meta-Agent* (from Phase 9) remains in charge, but now part of its duty is to accept improvement proposals and greenlight the Improver to execute them. Alternatively, we can have a *Governance routine* where the Chief and Self-Evaluator (and possibly other leads) have a short "post-mortem meeting" after each project to decide on changes. This could be simulated by the Chief agent reading the Critic's report and then instructing the Developer (Improver) agent to make certain modifications. - Existing agents might also improve themselves: e.g., each lead could adjust its strategies based on feedback. However, formalizing it through a Critic/Improver pair ensures a more structured approach (like how human organizations have retrospectives and then action items). - *Optional:* A **Learning Agent** that could ingest external research on multi-agent systems (via fetch) to find new techniques. This would be akin to the system researching how to improve itself using outside sources (maybe reading papers or forums for ideas). This might be an advanced extension of the Self-Evaluator's role: not just critique but also find solutions from literature or community.

Coordination Model: Recursion and Meta-Management: After completing tasks for the user, the system enters a self-management phase. The Chief coordinates this too: - It triggers the Self-Evaluator agent to analyze the run. This agent might act somewhat independently (like it could run a sequential-thinking analysis on the logs to find issues). - The Self-Evaluator then presents a report to the Chief (and possibly to all leads so they are aware). - The Chief, acting like a CEO with a report from QA, then prioritizes which improvements to act on. It forms a plan: e.g., "Improve prompt for X, add Y tool, retrain memory on Z format." The Chief then instructs the Improver agent to carry out these tasks. - The Improver (Developer) then makes the changes. This could involve: - Editing config files (e.g., adding a new MCP server entry for a new tool, updating the YAML to include a new agent). - Adjusting prompts in CLAUDE.md or other prompt templates to fix issues (like adding a reminder for the Writer to check facts). - Possibly writing new code if a

custom integration is needed (for example, writing a Python script to handle some repetitive task or a new agent). - Running tests: After changes, the Improver can run a quick simulation or a set of predefined scenarios to ensure things still work. If any test fails, it might roll back or adjust. - Committing changes to Git for versioning, so we have a history of improvements and can revert if something went wrong.

This meta-level operation is essentially the system using its own dev team on itself. The coordination is hierarchical: Chief decides what to improve, Developer executes, Critic verifies if the change resolved the issue in the next run.

Prompt Engineering: - Self-Evaluator (Critic) prompt: "You are a **System Critic AI** tasked with evaluating the performance and coordination of the multi-agent system itself. Analyze logs, outputs, and outcomes of the last task. Identify any mistakes, inefficiencies, or rule violations. Be specific: cite instances where an agent got stuck, or a better tool could have been used, etc. Also suggest possible improvements or new tools/agents that could help. Your output is a report for the Chief Coordinator to consider." This focuses the agent on introspection. The sequential-thinking tool can be very useful here for thorough step-by-step analysis of the process [15] [16]. - Improver (Refactor Developer) prompt: "You are an **Improvement Engineer AI**. You implement changes to the agent system according to the improvement plan. This may involve editing configuration files, adding new agent roles or tools, and adjusting prompts or memory contents. Ensure not to break existing functionality. Use Git to version control your changes. Test the system after changes. Document what you changed." This ensures the agent knows the scope (the codebase is the agent system itself now). - Chief prompt add-on: The Chief should be prompted to actually utilize the critic. Possibly: "After each project, ask the System Critic for a review and then decide on improvements." So the Chief doesn't forget to do self-improvement when appropriate. - All agents: foster a culture of learning. For instance, memory could include a section "Known Issues & Fixes" which all agents can read to avoid repeating known pitfalls. The Editor might check that list to not allow a known banned phrase to slip, etc. We can direct the Improver to update that memory section whenever a new lesson is learned (so it's a living document). - Encourage conservative changes: The improver should perhaps make one change at a time and test, rather than many at once, to isolate effects. We can encode that principle in prompt: "Make incremental improvements and verify them."

Memory Use: - The memory now also stores **meta-knowledge about the system**. For example, a memory file `system_evaluation.md` could accumulate all the Critic's past findings and what was done. This is valuable so the system doesn't flip-flop on decisions or re-attempt old fixes. It's also a track record to see improvement over time. - All prompt changes or config changes can be logged in memory or in a CHANGELOG. Possibly have the Improver agent update a `CHANGELOG.md` file with each change description (since it's using git, commit messages can double as this). - The Basic Memory can also store reference material for improvement, e.g., relevant parts of documentation or forum knowledge on using new MCP servers. If, say, the Critic suggests "maybe integrate an OCR tool," the system could fetch instructions for that and keep it for the Improver to use. - In essence, memory ensures continuous learning: the system builds its own user manual as it evolves.

Fault Tolerance: Self-improvement phase should also be fault-tolerant: - **Safe Mode for Self-Edits:** There's a risk the system could "improve" itself incorrectly and break. To mitigate, one could implement a fallback: keep a stable config and only use the new config after testing. In practice, the Improver can write changes to a branch or separate file, test with a dry run (maybe running a small scenario), and only then merge to main. If the test fails, revert. This process can be instructed. - **Human Oversight Option:** Although goal is fully autonomous, it's wise to allow a manual checkpoint before major changes. The system could output

the Critic's report and intended changes to the user, giving an option to intervene if something looks off. But since the prompt says fully autonomous, perhaps skip unless user specifically configured that. However, building in that capability is not bad: e.g., memory could have a setting "autonomous_improvement=true" which if false, the system would pause and ask for confirmation from a human. By default, we assume true here. - **Prevent Regressions:** The testing step catches obvious errors. Also, the Critic can check after an improvement if it indeed solved the targeted issue. If not, it might rollback or try another solution. This is a feedback loop: Critic says "problem X," Improver changes something, next run Critic sees if problem X happened again. If it still does, maybe the fix didn't work; try something else. This loop continues until resolved (or deemed too costly, then possibly escalate to human or accept a minor flaw). - **Scaling to New Domains:** The system should handle adding a new agent or tool gracefully. For example, if tomorrow we want an "UI Design Team" agent, the Improver can follow a documented pattern: set up a new role, integrate its tools, connect to Chief. The existing structure supports it because of modular design. We should test adding a dummy agent to ensure the config approach works (maybe as a simulation, e.g., add a simple agent that prints "Hello" to verify new addition doesn't break others). - **Tool Updates:** If an MCP server updates or breaks, the system should catch it. For instance, if the fetch API changes and fetch tool stops working as before, the agents will get errors. The Critic should see repeated fetch failures in logs and highlight it. Then the Improver might update the usage or switch to a different fetch alternative (maybe switch to a backup server like Apify RAG as listed in PulseMCP [40]). The key is the system can adapt to tool changes by having that feedback mechanism.

Execution Guidance: 1. **Automate Critique After Tasks:** Modify the Chief's workflow such that after completing a user's request (or on a schedule), it triggers the Self-Evaluator. Concretely, in the config or code, after finishing phase 9 tasks, call something like `critic.evaluate(project_id)` where the Critic agent reads through memory/log of that project. 2. **Critic's Analysis:** The Critic agent should be given access to transcripts or summaries of interactions. If the conversation logs are accessible via the filesystem or memory (maybe we log all agent communications to a file), the Critic reads those. If not, the Critic can rely on memory notes and outputs. The more data, the better for analysis. In a Claude environment, perhaps each message is logged; we might need to explicitly save some of that. The Developer/Improver can set up logging in earlier phases (like ensuring important steps are written to memory for exactly this purpose). 3. **Improvement Planning:** The Critic outputs a report. Possibly, it could be structured (like listing issues and suggested fixes). The Chief reads it and converts into a to-do list. For example: - Issue: "Fetch tool was slow due to large content." Suggestion: "Use `start_index` to paginate." Plan: Instruct Researcher agent to use that parameter when needed (update prompt). - Issue: "Developer agent sometimes overwritten memory notes incorrectly." Suggestion: "Add lock or prefix to avoid collisions." Plan: Implement a better file naming or lock mechanism. - New Opportunity: "New MCP 'imagegen' available that could benefit marketing." Plan: Install and add an Image Designer agent in marketing team. - The Chief then prioritizes (maybe all trivial ones now, bigger ones later). - If many, the Chief could ask the Developer agent to implement sequentially and test each. 4. **Implement Changes:** The Developer (Improver) goes through each planned change: - For prompt tweaks: open the relevant prompt file or CLAUDE.md and edit (Filesystem tool). - For adding a new MCP: call `claude mcp install ...` if available, or update `.claude/config.json` using an Edit tool to include the new server, then ensure it's running. - For adding new agent role: update YAML config (if using Claude Swarm, maybe edit the YAML to include new agent, or if dynamic, maybe the system can launch new instance on the fly). - Use Git: after each cohesive set of changes, commit with message. This provides a safety net (we can revert to previous commit if something breaks). - Possibly run the scenario tests using a script or instruct the system manually: e.g., the Developer might simulate a quick run of a representative task in a sandbox. If writing actual automated tests is possible (maybe using the system in a dummy mode), that would be sophisticated. Otherwise, a quick

sanity check via the Critic agent again might suffice. 5. **Validation:** After improvements, run a new project or re-run a problematic scenario to verify the issues are resolved. The Critic agent can be run again specifically focusing on the prior issues. If resolved, it should note improvement (maybe store that positive note in memory or remove the item from known issues). If not, try another iteration. 6. **Future-Proofing:** Document how to extend the system: - Write a guide (for human developers or for the AI itself in memory) on how the architecture is structured and how to add new capabilities. This can be a structured file, e.g., `architecture.md` summarizing each agent's role and connections. Having this documented ensures that if a future agent (or a future AI developer) joins, they can quickly integrate. - Keep the system updated with external developments: perhaps occasionally the Researcher agent could be tasked with scanning news about "new MCP servers" or "updates to Claude Code" and feed that to the system. This way, if a new powerful tool appears, the system can adopt it. For example, if a new "Database MCP" comes that would help the dev team, the Critic could flag that after reading about it. - This approach makes the system **modular and evolving**: any component can be upgraded or swapped with minimal effect on others, and the system can evolve by adding new modules rather than monolithic changes.

With Phase 10 implemented, Claude's multi-agent organization becomes self-improving and truly autonomous. It can analyze its own behavior, learn from mistakes [6], and adapt to new requirements or tools without needing explicit reprogramming. This concludes the roadmap, resulting in a robust, modular system that **"scales itself"** by creating or refining its agents in response to challenges.

Throughout these phases, we've included diagrams of agent relationships and tables of roles in the design documents (for brevity, not explicitly drawn here, but one can imagine an org chart with Chief at top branching to Dev/Research/Marketing leads, and those branching to specialists). The final system is akin to a multi-department organization contained within Claude, capable of tackling complex, multi-faceted objectives in a coordinated, autonomous fashion [34].

---

1 2 3 Claude Code Best Practices \ Anthropic
https://www.anthropic.com/engineering/claude-code-best-practices

4 5 6 8 32 33 Understanding LLM-Based Agents and their Multi-Agent Architecture | by Pallavi Sinha | Medium
https://medium.com/@pallavisinha12/understanding-llm-based-agents-and-their-multi-agent-architecture-299cf54ebae4

7 34 37 GitHub - parruda/claude-swarm: Easily launch a Claude Code session that is connected to a swarm of Claude Code Agents
https://github.com/parruda/claude-swarm

9 10 11 12 13 14 GitHub - basicmachines-co/basic-memory: AI conversations that actually remember. Never re-explain your project to Claude again. Local-first, integrates with Obsidian. Join our Discord: https://discord.gg/tyvKNccgqN
https://github.com/basicmachines-co/basic-memory

15 16 17 18 19 20 21 22 23 24 25 26 38 39 GitHub - FradSer/mcp-server-mas-sequential-thinking: An advanced sequential thinking process using a Multi-Agent System (MAS) built with the Agno framework and served via MCP.
https://github.com/FradSer/mcp-server-mas-sequential-thinking

27 28 30 31 mcp-server-fetch · PyPI
https://pypi.org/project/mcp-server-fetch/

29 40 Fetch MCP Server by Anthropic | PulseMCP
https://www.pulsemcp.com/servers/modelcontextprotocol-fetch

35 36 How to configure MCP servers when running multiple Claude Code agents? : r/ClaudeAI
https://www.reddit.com/r/ClaudeAI/comments/1mpfsdq/how_to_configure_mcp_servers_when_running/