ChatGPT

Scalable Multi-Agent Systems with Claude Agent SDK: Best Practices and Patterns

Abstract: This report outlines proven strategies for building scalable, maintainable multi-agent systems using Anthropic's Claude Agent SDK. We cover gradual agent expansion (dynamic module loading, versioning, changelog practices, and capability-based routing), meta-orchestrator control over subagents' tools (Memory, Context, and Permission management), patterns for dynamically registering new agents with minimal privileges, and examples of agent capability matrices with automated tool provisioning. Code snippets and structured recommendations are provided to guide development of robust orchestrator logic.

Gradual Expansion of Agents

The Claude Agent SDK enables you to start simple and incrementally grow your agent ecosystem. It's important to design your system such that adding or updating agents is straightforward and does not disrupt existing functionality. Key strategies include:

- **Dynamic Module Loading:** Define each subagent as a separate module or configuration file and load them dynamically. Claude supports subagents out-of-the-box – specialized agents stored as Markdown files under `.claude/agents/` (project-level) or `~/.claude/agents/` (user-level). This means you can add a new agent by dropping a file, and the main system will recognize it. Programmatically, you can use the SDK's `agents` option to register subagents at runtime. For example, using the Python SDK you might define agents and assemble them into a registry:

```
from claude_agent_sdk import AgentDefinition

# Define a new subagent
knowledge_builder = AgentDefinition(
    description="Agent that generates Obsidian knowledge base markdown files from research results",
    prompt="You are a knowledge base builder...",
    model="sonnet",
    tools=["Read", "Write", "Search"]  # limit to necessary tools
)
# ... define other agents similarly ...

# Assemble agents into a registry for the orchestrator
agents_registry = {
    "knowledge-builder": knowledge_builder,
    "quality-agent": quality_agent,
    "research-agent": research_agent,
    # etc.
}
# Use this registry when initializing the orchestrator agent (see below)
```

By modularizing agent definitions, you can gradually expand the system by adding new files or modules. This avoids monolithic code and allows testing agents in isolation. Ensure your main script automatically includes all agents (e.g. by scanning the `agents` directory or explicitly importing new agent modules) so that new agents don't get "forgotten" during incremental development [1] [2]. A central **agent registry** (as shown above) helps keep track of all agents in one place.

- **Agent Versioning and Changelogs:** Treat agent definitions as versioned artifacts, much like source code. Anthropic recommends saving agents in your repository and **versioning them like code** so that improvements benefit everyone [3]. Use Git or similar version control to track changes in an agent's prompt, tools, or behavior. This provides an implicit changelog through commit history. For more explicit documentation, consider maintaining a **changelog section** in each agent's Markdown (or in a separate file) describing updates, new capabilities, or behavior changes. This is especially useful as your team expands or when debugging why an agent's behavior changed. When introducing a significantly new agent version, you might keep the old version registered in parallel (e.g. `research-agent-v2` alongside `research-agent-v1`) and gradually route traffic to the new one – this **blue-green deployment** style ensures stability while iterating on agents.

- **Capability-Based Routing:** As your agent pool grows, implement logic to route tasks to the most appropriate agent based on their capabilities or domain expertise. In Anthropic's best practices, this is referred to as a **routing workflow**, where an initial classifier (which can be an LLM itself) directs the input to a specialized follow-up agent [4]. For example, your orchestrator might inspect a user query and decide: "This looks like a deep research question, I should delegate to the `research-agent`" or "This is a request for examples, use the `example-generator`." You can encode simple routing rules (by keyword or metadata), or let the main Claude model decide automatically based on agent descriptions. Claude's Agent SDK supports **automatic delegation**: if an agent's `description` matches the task, Claude can proactively invoke it [5]. To leverage this, make each subagent's description **clear and action-oriented**, specifying trigger keywords or scenarios. For instance, a subagent description might say "Use proactively for mathematical deep-dives" – then Claude may spawn it on a math research query without explicit instructions [5]. In more complex setups, you can have a meta-agent explicitly perform routing by calling subagents (e.g., using a "Task" tool or an API call to launch agents). Anthropic's engineering blog notes that routing enables separation of concerns and ensures each agent's prompt is specialized for its category of input [4]. In practice, combine this with versioning: as new agent versions or types are added, update the routing logic or descriptions so the orchestrator knows when to use them.

Orchestrator Control of Memory, Context, and Permissions

In a multi-agent system, a **meta-orchestrator** (lead agent) coordinates subagents. It must manage what tools and information each subagent can access, enforcing safety and efficiency. The Claude Agent SDK provides fine-grained controls for **Memory**, **Context**, and **Permission** tools to implement a least-privilege architecture:

- **Isolated Context and Memory Sharing:** Each subagent has its own context window, separate from the orchestrator and other agents [6]. This prevents irrelevant information bleed-over and keeps each agent focused. The orchestrator should decide what context to share with subagents. For example, the lead agent might pass a specific snippet of text or a summary to a subagent as part of its prompt, rather than giving the subagent the entire user query history. Claude's built-in design uses a shared file system for context – e.g. the orchestrator can save a research plan to a `Memory` file that subagents can read using a tool. In Anthropic's internal research system, the lead agent writes its plan to memory so it isn't lost even if the context window overflows [7]. You can replicate this by giving the orchestrator agent access to a "memory keeper" tool (for example, an MCP server or custom tool that saves and retrieves notes). In code, one might include a memory tool in the main agent's allowed tools, like:

```
options = ClaudeAgentOptions(
    model="sonnet",
    agents=agents_registry,
    allowed_tools=["Task", "Read", "Write", "MemorySave", "MemoryLoad"],  # main agent tools
    ...
)
```

This ensures the orchestrator can persist important context (plans, decisions) and later fetch it, while subagents only see what the orchestrator chooses to give them. **Memory isolation** is generally advisable: let subagents store intermediate results in their own ephemeral context or files, and have the orchestrator pull relevant pieces. This way, one subagent's findings can be passed to another through the orchestrator in a controlled manner (for example, via a summarized result rather than raw context).

- **Tool Permission Policies (Least Privilege):** Only grant each agent the minimum tools it needs to fulfill its role. The SDK allows specifying `allowedTools` or per-agent `tools` lists to enforce this [8] [9]. By default, if you don't specify a subagent's tools, it inherits all tools available to the main agent [10] – **avoid leaving this open** unless necessary. Instead, explicitly enumerate tools for each subagent. For instance, you might configure a web research agent with just the web browsing tool and perhaps a text analysis tool – no file write or shell access. A data-entry agent might have file write permission but no internet access, etc. The official docs emphasize that each subagent can have different access levels, allowing you to **limit powerful tools to specific agent types** [11]. This encapsulation improves security and reliability (e.g., your "quality-agent" can't accidentally execute code, because you never gave it that ability). In practice, implement this by passing the appropriate `tools` array when defining each `AgentDefinition` [12] or by editing the YAML frontmatter in the subagent's markdown file. For example:

```
# .claude/agents/quality-agent.md (frontmatter)
name: quality-agent
description: Validation agent that checks generated files against quality criteria
tools: Read, SchemaCheck  # only read files and validate schemas
model: inherit
---
(Prompt instructions...)
```

Here, the quality agent cannot write or execute code at all. **Permissive vs. restrictive modes:** The SDK provides a `permissionMode` setting to control global behavior of tool use. For most cases, the default mode (which asks for confirmation for risky operations) is fine, but in a fully autonomous setup you might set `permissionMode='acceptEdits'` to auto-approve file edits while still disallowing anything outside the allowed list [13]. Conversely, a `'plan'` mode exists to simulate actions without executing them [13] – useful if you want a planning agent that never actually runs tools. Choose modes per agent or per session to balance safety and autonomy. You can even implement a custom permission function (`CanUseTool`) to dynamically decide on each tool invocation [14] – for example, to log or prevent certain usage at runtime beyond the static allow-list. Summarily, **policy guidelines** are: use allow-lists generously, grant write/exec privileges sparingly, and prefer read-only or "plan" modes for experimental agents until they are well-tested [15].

- **Meta-Orchestrator as Gatekeeper:** The orchestrator (lead agent) itself should often have a broader toolset, but you still want to constrain it to its coordination duties. Typically, the orchestrator needs the ability to spawn or invoke subagents and aggregate results. In Claude's ecosystem, this can be done implicitly (Claude will delegate on its own when appropriate) or explicitly via a tool (e.g., the "Task" tool) that the orchestrator can call to initiate a subagent. Make sure the orchestrator's allowed tools include the mechanism for delegation – for example, the built-in `"Task"` tool if using it, or ensure the orchestrator has an API to call `claude_agent_sdk.query()` for subagents. Apart from that, limit the orchestrator's direct tools to those needed for top-level coordination. In our example above, we gave the main agent `Read`/`Write` (to handle any file I/O needed for gathering context or assembling outputs) and some **memory tools** for context management. We did not give it web search or other specialized capabilities – those reside with subagents. This separation means the orchestrator must actively choose to use a subagent (with its distinct tools) for certain tasks, rather than doing everything itself. It enforces the design where "each agent has a specialized domain and toolset" as recommended [16]. The orchestrator can also enforce higher-level policies: e.g., if a subagent returns an unexpected result or tries to step out of bounds, the orchestrator (via Claude's reasoning or via code) can decide not to accept it or to spawn a different agent to double-check. In essence, the meta-agent becomes a **policy governor**, assigning tasks and vetting outcomes.

- **Recommended Criteria & Policies:** In practice, Anthropic's guidance and the community tutorials converge on a few policies for orchestrator-managed multi-agent systems:

- Encourage parallelism for independent tasks: The orchestrator should spawn multiple subagents in parallel if possible to speed up complex tasks [17]. For example, if researching multiple questions, fire off two research subagents concurrently. This requires that the orchestrator (or Claude itself) handle parallel tool calls; the Anthropic blog notes up to 3-5 subagents in parallel yielded ~90% reduction in latency in their research system [17].
- Explicitly define subagent triggers: Use clear language in subagent descriptions such as "Use proactively when X" or "MUST be used for Y" to help the orchestrator auto-delegate correctly [5]. If relying on automatic delegation, fine-tune these descriptions to avoid ambiguity (so agents don't overlap or compete for the same task).
- Memory and state management: Determine whether subagents should carry state between invocations. Often they should not (fresh context each time), but if needed, the orchestrator can pass along necessary state. Keep any long-term memory in one place (like the orchestrator's memory file) rather than scattered.
- Tool quality checks: If you integrate custom MCP tools, be aware that inconsistent or poorly described tools can confuse agents [18]. A meta-pattern used by Anthropic was to create a tool-testing agent that automatically tries out new tools and refines their descriptions [19]. Your orchestrator could similarly restrict new tools to a "sandbox" agent that verifies them before making them available broadly.

Dynamic Registration of New Agents with Limited Privileges

When introducing new capabilities, you often want to plug in a new agent without compromising the system. The Claude Agent SDK allows dynamic agent registration both through config files and programmatically:

- **On-the-Fly Registration:** If using Claude's built-in CLI/IDE integration, the `/agents` command lets you interactively create a new subagent and select its tools. This is great for quick experiments: you can spin up a new agent at runtime. Under the hood, this creates a Markdown file in `.claude/agents/` with the specified config. In a running SDK-based system (your own app or script using Claude API), you can similarly instantiate a new `AgentDefinition` and add it to the `agents` dictionary in your next query. The SDK merges these definitions with any existing ones. Programmatic options like `agents` in the query call always override the static filesystem settings [20], so you have full control to inject agents on the fly. For example, you might maintain a registry and have a function `register_agent(name, AgentDefinition)` that updates the global options for your orchestrator.

- **Limited Privileges for New Agents:** It is wise to start any new agent in a sandboxed mode. By default, consider giving a new agent **no dangerous tools** and possibly running it in `permissionMode='plan'` (so it only plans but doesn't execute actions) until you trust it. For instance, if you add a `db-query-agent` that connects to a database, first allow it only read access and perhaps test queries on a test database. Gradually expand its tool access as it proves reliable. This incremental approach aligns with the least privilege principle and prevents accidents during development. You might maintain a default allow-list for "experimental" agents (e.g., only allow `Read` and a debug print tool) and only elevate permissions after code review or evaluation.

- **Automated Discovery:** In a large system, manually editing a central registry for each new agent could become cumbersome. You can automate registration. One pattern is to use naming conventions or a plugin architecture: e.g., any Markdown file in `.claude/agents/` is auto-loaded, or any Python class that subclasses `AgentDefinition` in a certain directory is automatically imported. If using Python, you could dynamically import all modules in an `agents` package and collect their agent definitions. Another approach is to maintain a JSON/YAML configuration that lists active agents and their settings; the orchestrator on startup reads this and constructs the agents accordingly. The key is to have a **single source of truth** for what agents exist, so that adding a new agent is as simple as adding one file or one config entry – no scattered changes. Our earlier code snippet assembling `agents_registry` could be generated automatically by scanning files. Just be sure to also update any **routing logic** or orchestrator prompts so the new agent is actually utilized.

- **Versioning and Rollback:** Dynamic registration also means you can quickly disable or swap out agents. If an agent is failing or behaving poorly, you can remove it from the registry (or mark it inactive in config) without deleting the code. For example, toggling a feature flag in your config could tell the orchestrator to stop using `example-generator-v1` and start using `example-generator-v2`. Because the orchestrator controls invocation, it can simply choose not to call the faulty agent. Always monitor new agents closely (through logs or evals) – build metrics to track each agent's performance (e.g., success rate, tool usage count) so you have data to decide if a new agent is ready for prime time [21] [22].

- **Example – Updating `main.py` for Dynamic Agents:** Below is a simplified pattern (in pseudocode) illustrating how one might dynamically load agents and enforce limited privileges at startup, using the Claude SDK:

```
import importlib, pkgutil
from claude_agent_sdk import AgentDefinition, ClaudeAgent

agents_registry = {}
# Discover and import all modules in the 'agents' package
for _, module_name, _ in pkgutil.iter_modules(agents.__path__):
    module = importlib.import_module(f"agents.{module_name}")
    # Each agent module is expected to have an `agent_def` object
    agent_def = getattr(module, "agent_def")
    agents_registry[module_name] = agent_def
    # Optionally, downgrade permissions for new/experimental agents
    if module_name.startswith("experimental_"):
        agent_def.tools = ["Read"]  # only allow read for experimental agents

# Instantiate the orchestrator (main agent) with all subagents
orchestrator = ClaudeAgent(
    model="claude-2.0/sonnet",
    prompt=meta_orchestrator_prompt,
    agents=agents_registry,
    allowed_tools=["Task", "Read", "Write", "MemorySave", "MemoryLoad"],  # main agent tools
    permission_mode="acceptEdits"  # auto-approve safe edits, but still constrained by
allowed_tools
)
```

In this sketch, any new file in the `agents` directory will be picked up. We also demonstrate how you might automatically restrict tools for certain agents (here, any agent module prefixed with "experimental_" is given a minimal toolset). Adapting this to your needs ensures that gradual expansion is safe: new agents are easy to add, but start restricted.

Agent Capability Matrix and Tool Provisioning Examples

It's often useful to explicitly map out each agent's capabilities – essentially a **capability matrix** that shows which agent can do what. This helps both in design (to avoid overlaps or gaps) and in implementing automated tool provisioning. Below is an example matrix for a hypothetical system (columns truncated for brevity):

Multi-agent orchestrator architecture from Anthropic's research system. The lead agent (orchestrator) uses specialized tools (e.g. web search, memory management) and spawns subagents in parallel for different subtasks [23] [7].

Agent Name	Role/Purpose	Allowed Tools	Notes
Meta-Orchestrator (lead)	Coordinates all subagents; planning and synthesis of final result.	`Task` (invoke subagent), `Read/Write` (filesystem I/O), `MemorySave/Load` (persist context), no direct internet	Uses **Task** tool to delegate work to subagents. Holds global context/memory [7]. No domain-specific logic – focuses on orchestration.
Research-Agent	Deep web research on topics.	Web search tool (e.g. BraveSearch MCP), `Read` (to parse results), no file write	Searches the web and extracts information [24]. Runs in parallel if multiple research tasks [17]. Cannot modify files, only returns findings.
Knowledge-Builder	Creates knowledge base entries (Markdown files).	`Read`, `Write` (file I/O), perhaps local search (grep)	Takes research outputs and composes Markdown. Only file system tools so it can write notes, plus read for context. No internet access (relies on Research-Agent's data).
Quality-Agent	Validates generated content against criteria.	`Read` (open files), `Validate` (custom tool for schema/standards)	Checks output files for errors, style, or schema compliance. Read-only access by design [25]. Reports issues to orchestrator; cannot alter content itself.
Example-Generator	Produces example problems/ solutions for a topic.	`Compute` (math API or Python tool), `Read` (if needed)	Generates new content (e.g., math exercises). Has computational tools (like a Python execution tool or math library MCP). No file write unless we want it to save outputs.
Socratic-Planner	Asks clarifying questions for ambiguous requests.	`Chat` (to user, if interactive), `SequentialThinking` (MCP tool for step-by-step prompting)	Engages in a dialogue to refine requirements. Limited to conversational tools and reasoning – no file or external access.
Dependency-Mapper	Analyzes and maps prerequisite relationships.	`NLPAnalysis` (language processing tool), `GraphPlot` (to output dependency graph)	Processes text (maybe from Knowledge-Builder) to find concept dependencies. Could output a graph. No external access.

Table: Example capability matrix for agents, detailing their roles and tool access. Each agent's tool set is limited to its needs (principle of least privilege) [11] .

In this matrix, each agent's allowed tools are configured to suit its function. Such a table can guide how you configure `tools` in each AgentDefinition or YAML file. It also informs the **automated tool provisioning**: if you categorize tools (e.g. "internet", "filesystem", "computation", "validation"), you could automatically assign tools based on an agent's role tags. For instance, label `research-agent` with category "internet" and your initialization code grants it the web search tool; label `quality-agent` with "validation" and grant it the validator tool. In code, this might look like:

```
tool_by_category = {
    "internet": ["BraveSearch"],
    "filesystem": ["Read", "Write"],
    "validation": ["SchemaCheck"],
    "compute": ["PythonExec"]
}
# Pseudocode: assign tools based on tags
for name, agent in agents_registry.items():
    tools = []
    for tag in agent.tags:   # assume we defined some tags for the agent
        tools += tool_by_category.get(tag, [])
    agent.tools = list(set(tools))
```

This way, adding a new agent with certain tags automatically gives it the standard tools for that capability. Regardless of automation, **review each agent's tool list carefully** before deploying. The anecdote from Anthropic's blog is clear: using the wrong tool or having too broad access can derail an agent's performance [18] . For example, if an agent meant to search a local knowledge base is mistakenly allowed web search, it might waste time or get irrelevant info. Thus, building a capability matrix and enforcing it not only improves security but also aligns each agent's actions with its intended scope.

Finally, note that **capability restrictions can evolve**: you might start an agent with very limited tools and later expand if needed. Monitor agent behavior – if you find an agent struggling because it lacks a tool (e.g., your Example-Generator needs internet access to fetch up-to-date data), you can then decide to grant that new capability in a controlled manner (and update your matrix/documentation accordingly).

Conclusion

Designing a scalable multi-agent system with the Claude Agent SDK involves balancing flexibility with control. By modularizing agents and using dynamic loading, you can expand the system gradually without major refactoring. Employing version control and clear changelogs for agent definitions ensures that improvements are tracked and can be rolled out or rolled back safely [3] . A capability-based routing approach, where an orchestrator or classifier directs tasks to the right agent, leads to cleaner separation of concerns and better overall performance on complex tasks [4] .

Critically, the orchestrator should enforce strict boundaries on what each subagent can do – leveraging SDK features like `allowedTools`, `permissionMode`, and isolated contexts [11] [13] . The principle of least privilege not only safeguards your system but also makes agent behaviors more predictable (since each agent has a narrow, well-defined toolset). As demonstrated, creating a capability matrix and provisioning tools accordingly is a practical way to implement these boundaries.

When adding new agents, start small: register them easily via the SDK's agent registry, but give them minimal powers until they earn trust. Use parallelism for independent subtasks and let the orchestrator synthesize results, much like Anthropic's orchestrator-worker pattern where a lead Claude agent with Claude Opus spawns Claude Sonnet subagents for research [26] [23]. Always evaluate and iterate – monitor token usage, success rates, and errors for each agent. This evaluation-driven refinement was key in Anthropic's own multi-agent research system (which saw a 90% performance boost over single-agent by effectively distributing work) [26].

In summary, **building a maintainable multi-agent system** with Claude involves: clear agent roles, carefully scoped tools, an orchestrator that smartly delegates (and nothing more), and a disciplined approach to growth (with versioned, documented changes). Following these patterns and the SDK best practices [27] [16], you can construct a robust agent ensemble that scales in capability while remaining manageable and safe. By giving each agent "a computer" with just the right tools and guidance [28] [29], and by orchestrating their cooperation, you unlock complex workflows that single agents alone could not easily handle – all while retaining the control needed for production-quality systems.

Sources:

1. Anthropic, Claude Agent SDK Documentation – Subagents [11] [8]
2. Anthropic Engineering Blog, How we built our multi-agent research system (Jun 2025) [23] [7]
3. Anthropic Engineering Blog, Building agents with the Claude Agent SDK (Sep 2025) [30] [5]
4. Anthropic Engineering Blog, Building effective AI agents (Dec 2024) [4]
5. Medium (Mazaharul Huq), Mastering Claude Code's Sub-agents (Jul 2025) [15]
6. Claude SDK Reference (TypeScript/Python) – AgentDefinition and Permission Options [12] [13]
7. Anthropic Documentation – Model Context Protocol (MCP) and Tools Reference [18] [19]
8. Internal Analysis – Meta-Orchestrator Improvement Plan [31] [32] (summarizing official guidelines)

---

[1] [2] [16] [21] [22] [27] [31] [32] meta-orchestrator-improvement-analysis.md
file://file_00000000311861f4bcfde30d905575bb

[3] [15] Practical guide to mastering Claude Code's main agent and Sub-agents | by Md Mazaharul Huq | Medium
https://jewelhuq.medium.com/practical-guide-to-mastering-claude-codes-main-agent-and-sub-agents-fd52952dcf00

[4] Building Effective AI Agents \ Anthropic
https://www.anthropic.com/engineering/building-effective-agents

[5] [6] [8] [9] [10] [11] [25] Subagents - Claude Docs
https://docs.claude.com/en/docs/claude-code/sub-agents

[7] [17] [18] [19] [23] [24] [26] How we built our multi-agent research system \ Anthropic
https://www.anthropic.com/engineering/multi-agent-research-system

[12] [13] [14] [20] Agent SDK reference - TypeScript - Claude Docs
https://docs.claude.com/en/api/agent-sdk/typescript

[28] [29] [30] Building agents with the Claude Agent SDK \ Anthropic
https://www.anthropic.com/engineering/building-agents-with-the-claude-agent-sdk