Claude Agent SDK & Agentic Orchestration by Michaël Bettan

Definition: The **Claude Agent SDK** (TypeScript `@anthropic-ai/claude-agent-sdk` · Python `claude-agent-sdk`, which replaces `claude-code-sdk`) packages the Claude Code agent harness as a library: the agent loop, built-in tools, permissions, hooks, subagents, sessions and MCP, running inside **your** process. This guide covers the SDK API and the architecture patterns you build with it: loops, coordinator–subagent orchestration, deterministic enforcement, session state and interrupts, custom tools and their errors, permission modes, subagent limits, context compaction and structured results.

Three ways to wire an agent:

Wiring path	Who runs the loop	You own	Pick when
Raw Messages API loop	Your code	Tool execution, `stop_reason` handling, state, retries, compaction	Full control, minimal dependencies
Claude Agent SDK	SDK in your process	Hosting, filesystem isolation, storage	You want Claude Code's tools, hooks, subagents and sessions in your own infrastructure
Claude Managed Agents	Anthropic server-side	Agent config, custom tool handlers, session lifecycle	Long-running hosted sandbox sessions without running containers yourself

MENTAL MODEL

The model *decides* (which tool, which subagent, when to stop); the harness *enforces* (permissions, hooks, budgets). Anything that must hold 100% of the time belongs in code (hooks, tool scoping, prerequisite gates), never only in the prompt.

SCOPE NOTE

Claude Code configuration (permission modes, `CLAUDE.md`, skills, plugins, CLI flags) lives in the Claude Code guide. MCP protocol internals live in the MCP guide.

1. The agentic loop and `stop_reason`

Raw Messages API loop (what the SDK does for you):

1. **Call:** send `messages` + `tools` to Claude.
2. **Inspect:** read `stop_reason` on the response.
3. **If `"tool_use"`:** execute **every** `tool_use` block, append the assistant turn, then append one user turn containing a `tool_result` block per call (matched by `tool_use_id`). Go to 1.
4. **If `"end_turn"`:** the model is done. Use the final text.

`stop_reason`	Meaning	Loop action
`tool_use`	Claude wants tools run	Execute, append `tool_result`s, call again
`end_turn`	Natural completion	Stop; return the answer
`max_tokens`	Hit your `max_tokens` cap mid-response	Raise the cap or continue the response; output is truncated
`pause_turn`	A **server-tool** loop hit its iteration limit	Send the assistant content back unchanged to let Claude continue
`refusal`	Claude declined	Handle explicitly (read `stop_details`, fallback model or human)
`stop_sequence`	One of your `stop_sequences` fired	Read `stop_sequence`
`model_context_window_exceeded`	Response filled the context window	Treat as truncated; compact or split the task

DECISION RULE

The loop's termination signal is `stop_reason`, **not** the text. Continue on `tool_use`; stop on `end_turn`; handle the rest explicitly.

EXAM TRAPS (LOOP ANTI-PATTERNS)

(1) parsing natural language ("I'm done", "task complete") to decide termination; (2) treating "the response contains text" as completion (Claude often writes text *and* calls tools in the same turn); (3) using an arbitrary iteration cap as the **primary** stop condition (keep a cap only as a safety bound); (4) dropping tool results instead of appending them, so the model cannot reason about what happened.

Model-driven vs pre-configured: In an agent loop Claude chooses the next tool from context. A fixed decision tree or hard-coded tool sequence is a *workflow*, not an agent; use it only when the order is known in advance (see §3).

The SDK version of the loop: `query()` runs the loop and streams messages (`AssistantMessage`, `ResultMessage`, ...). Check `ResultMessage.subtype` first:

`ResultMessage.subtype`	What happened	`result` present?
`success`	Finished normally	Yes
`error_max_turns`	Hit `max_turns` / `maxTurns`	No
`error_max_budget_usd`	Hit `max_budget_usd` / `maxBudgetUsd` (subagent spend counts toward the cap)	No
`error_during_execution`	Loop interrupted (cancelled request, crash)	No

`ResultMessage.subtype`	What happened	`result` present?
`error_max_structured_output_retries`	No schema-valid structured output within the retry limit	No

- Every result carries `session_id`, `total_cost_usd`, `usage`, `num_turns`, so you can resume even after an error. The result also has a `stop_reason` (`end_turn`, `max_tokens`, `refusal`, ...).
- A single-shot `query()` **raises after yielding an error result**: wrap the loop in `try` if you need to continue (for example, resume with a higher `max_turns`).

2. Agent SDK core API

- **`query(prompt, options)` vs `ClaudeSDKClient`:**
- `query()` starts a fresh session per call unless you pass **`continue_conversation=True`** (`continue: true` in TS), **`resume=session_id`** (from `ResultMessage.session_id`), or **`resume=session_id, fork_session=True`** (`forkSession: true`) to branch history into a new session ID. **Fork trap:** `fork_session` branches the conversation transcript **only**, not the filesystem. Pair with `enable_file_checkpointing=True` + `rewind_files(user_message_id)` to restore disk state.
- **`ClaudeSDKClient`** (Python `async with ClaudeSDKClient(options) as client:`): reuses one session across multiple `await client.query(...)` + `client.receive_response()` turns and supports **`await client.interrupt()`** (**drain buffered messages** up to the interrupted turn's `ResultMessage` before sending the next query). TypeScript has no `ClaudeSDKClient` class; call methods on the `Query` object returned by `query()`.

`setting_sources` / `settingSources` (`"user" | "project" | "local"`):

- **Omitted (default)** → loads `user`, `project` and `local` settings, `CLAUDE.md`, skills, agents and hooks, **just like the interactive CLI**. *(Exception: if top-level `skills` is passed while `setting_sources` is omitted, only `user` and `project` load.)*
- **`[]` (empty list)** → disables `user`, `project` and `local` filesystem settings and `CLAUDE.md`. **Multi-tenant trap — what `setting_sources=[]` still loads:** (1) **managed policy settings**, (2) **`~/.claude.json`** global state (isolate via `CLAUDE_CONFIG_DIR` in `env`), (3) **auto memory** `MEMORY.md` (disable with **`CLAUDE_CODE_DISABLE_AUTO_MEMORY="1"`** or `autoMemoryEnabled: false`), and (4) **`claude.ai` MCP connectors** (`mcp_servers={}` does *not* suppress them; pass **`strict_mcp_config=True`** / `strictMcpConfig: true`).

Option (Python / TS)	What it controls & critical SDK traps
`tools`	**Built-in tool availability in context** (`["Read", "Grep"]`, or `[]` to strip all built-ins so Claude sees only MCP tools).
`allowed_tools` / `allowedTools`	**Pre-approves** matching calls; unlisted tools stay in context and fall through to `permission_mode` / `can_use_tool`. No effect in `bypassPermissions`.
`disallowed_tools` / `disallowedTools`	Bare names (`"Bash"`) **remove the tool from context**; scoped rules (`"Bash(rm *)"`) keep the tool and **deny matching calls in every mode**, including `bypassPermissions`. File-write rules must use `Edit(//abs/path)` (`Write(...)` never matches).
`permission_mode` / `permissionMode`	`"default"`, `"acceptEdits"`, `"plan"`, `"auto"`, `"dontAsk"`, `"bypassPermissions"`. In TS, `"bypassPermissions"` throws unless **`allowDangerouslySkipPermissions: true`** is also set.
`can_use_tool` / `canUseTool`	Async callback returning `PermissionResultAllow(updated_input=...)` or `PermissionResultDeny(message=...)`. Fires **only** when the harness would prompt; **never** in `dontAsk`, `bypassPermissions`, or for tools matched by `allowed_tools`.
`hooks`	In-process callbacks (`PreToolUse`, `PostToolUse`, `Stop`, `SubagentStop`, `UserPromptSubmit`, ...) registered with `HookMatcher` (see §5).
`mcp_servers` + `create_sdk_mcp_server`	External or **in-process custom tools** (`@tool` + `create_sdk_mcp_server(name="srv", tools=[fn])`; TS `tool()` + `createSdkMcpServer()`). Pre-approve as **`mcp__<srv>__<tool>`**.
`agents` (`AgentDefinition`)	Programmatic subagents (see §4). **Python quirk:** multi-word fields use **camelCase** (`disallowedTools`, `mcpServers`, `maxTurns`, `permissionMode`, `initialPrompt`).
`max_turns` / `max_budget_usd`	Safety bounds; produce `error_max_turns` / `error_max_budget_usd`.
`output_format` / `outputFormat`	`{"type": "json_schema", "schema": ...}` → validated payload in `ResultMessage.structured_output`.
`env`	**Python merges** `env` onto the process environment; **TypeScript replaces** `process.env` — spread `{ ...process.env, MY_VAR: "1" }` or `PATH` is lost and the CLI fails with `ENOENT`.

3. Coordinator–subagent orchestration and task decomposition

Hub-and-spoke: the main agent is the only node that talks to every subagent. Children report back to it and never to each other, which keeps one place to log, budget and merge results.

Coordinator responsibility	Do	Avoid
Delegate only when it pays	Answer a one-line lookup in the main agent; spawn when the work is verbose or parallel	A subagent per request "just in case": every child costs a fresh context and its own API calls
Non-overlapping briefs	Give two children disjoint directories, services or questions	Two children reading the same files and returning conflicting edits
Route	Write each `AgentDefinition.description` as a "use when..." sentence so Claude can tell agents apart	Two agents with near-identical descriptions, so delegation is a coin toss
Brief	Say what "done" looks like and what to return	Scripting every command, so the child cannot react to what it finds
Bound	Cap spend with `max_budget_usd` and nesting with `CLAUDE_CODE_MAX_SUBAGENT_SPAWN_DEPTH`	An unbounded tree: subagents can spawn their own subagents

Choosing a decomposition pattern:

Pattern	Use when	Example
Fixed workflow (code decides the order)	The steps never change	Lint → type-check → summarize, each as its own `query()` call
Agent-led delegation	What to do next depends on what was found	Incident triage: the agent reads alerts, then decides whether a log-reader or a config-diff subagent is worth spawning
Parallel fan-out	Children do not depend on each other	`style-checker`, `security-scanner` and `test-coverage` subagents on the same change

EXAM TRAP — ONE CONTEXT FOR EVERYTHING

stuffing every log, diff and doc into the main agent fills its window and triggers compaction early. Push verbose reading into subagents so only their short reports land in the parent.

4. Subagent invocation, `AgentDefinition` and context passing

```python
from claude_agent_sdk import query, ClaudeAgentOptions,
AgentDefinition

options = ClaudeAgentOptions(
    allowed_tools=["Read", "Grep", "Glob", "Agent"],  # the
coordinator needs the Agent tool
    agents={
        "code-reviewer": AgentDefinition(
            description="Security and quality review. Use for code
review tasks.",  # WHEN to delegate
            prompt="You are a code review specialist...",  #
subagent system prompt
            tools=["Read", "Grep", "Glob"],  # least privilege
            model="sonnet",
        ),
    },
)
```

- **Spawning mechanism:** Claude spawns subagents through the **`Agent` tool** (legacy name `Task`, still accepted as an alias; older docs and exams say "Task tool"). The coordinator can only delegate if that tool is **in context** (with an explicit `tools` list, include `"Agent"`; a bare `disallowed_tools` entry removes it) **and approved** (docs examples also list `"Agent"` in `allowed_tools`, which only pre-approves calls).
- **`AgentDefinition` fields:** `description` (required; tells the coordinator *when* to use it), `prompt` (required; system prompt), `tools` (omit = inherit all), `disallowedTools`, `model` (`"sonnet"`, `"opus"`, `"haiku"`, `"inherit"`, or full ID), `skills`, `mcpServers`, `maxTurns`, `memory`, `background`, `effort`, `permissionMode`, `omitClaudeMd` (TS).
- **Fan-out limits (documented defaults):** `CLAUDE_CODE_MAX_SUBAGENT_SPAWN_DEPTH` = **3** layers below the main agent (`1` stops subagents from spawning their own); `CLAUDE_CODE_MAX_CONCURRENT_SUBAGENTS` = **20** running at once; `max_budget_usd` has no default and counts subagent spend.

A (non-fork) subagent receives	It does not receive
Its own system prompt (`AgentDefinition.prompt`) + the **Agent tool's prompt string**	The parent's conversation history or tool results
Project `CLAUDE.md` (via `setting_sources`, unless `omitClaudeMd`)	The parent's system prompt
Its tool definitions (inherited or the `tools` subset)	Skill content, unless listed in `skills`

DECISION RULE — CONTEXT PASSING

A fresh subagent knows only what the Agent prompt string tells it. If the parent already found the failing test name, the stack trace or the file that owns a function, that text has to be **copied into the prompt**; a reference such as "the error above" points at nothing. Ask for a fixed-shape final report so the parent can parse it.

- **Return path:** only the subagent's **final message** returns to the parent; its intermediate tool calls stay inside the subagent. Large outputs belong on disk, with the path in the report.
- **Background by default:** a spawned subagent runs in the background (`status: "async_launched"` in the Agent tool result) and its report arrives in a later turn; set `background` on the `AgentDefinition` to make the intent explicit.
- **Resume a subagent:** a custom subagent's Agent tool result ends with `agentId: <id>`. Resume the same session (`resume=session_id`) and name that agent ID in the prompt to continue with its full history instead of re-exploring.
- **Fork vs fresh subagent:** a *fork* inherits the parent conversation; a normal subagent starts fresh. Use fresh subagents for isolation and independent review.

5. Hooks: lifecycle and deterministic enforcement

Hook lifecycle (Python + TypeScript unless noted):

Event	Fires when	Typical production use
`UserPromptSubmit`	A prompt is submitted	Inject tenant or ticket context
`PreToolUse`	Before a tool runs (can block or rewrite)	Guard shell commands, rewrite paths into a sandbox
`PermissionRequest`	A call needs a permission decision	Route approvals to your own UI
`PostToolUse` / `PostToolUseFailure`	After a tool succeeds / fails	Audit log, redact secrets, normalize formats
`SubagentStart` / `SubagentStop`	A subagent spawns / finishes	Trace fan-out; `SubagentStop` carries `agent_transcript_path`
`PreCompact`	Before history is summarized (`trigger`: `manual` / `auto`)	Archive the full transcript first
`Stop` / `Notification`	Main agent stops / emits a status message	Persist state, page an operator
`SessionStart`, `SessionEnd`, `PostCompact`	**TypeScript only**	Telemetry setup, cleanup, log the summary

Need	Mechanism	Why
A rule that must hold on **every** call (no deletes outside staging, no writes to `.env`)	**`PreToolUse` hook**, scoped deny rule or narrower tool	Prompt instructions are followed most of the time, not all of the time
Clean or annotate tool output before the model reads it	**`PostToolUse` hook** (`updatedToolOutput`, `additionalContext`)	One place to strip secrets or convert units
Style, tone, preferred approach	**System prompt / `CLAUDE.md`**	Guidance; occasional deviation is acceptable

```python
from claude_agent_sdk import ClaudeAgentOptions, HookMatcher

async def guard_kubectl(input_data, tool_use_id, context):
    cmd = input_data["tool_input"].get("command", "")
    if "kubectl delete" in cmd and "-n staging" not in cmd:
        return {"hookSpecificOutput": {
            "hookEventName": "PreToolUse",
            "permissionDecision": "deny",
            "permissionDecisionReason": "Deletes run only with -n staging. "
                "For other namespaces, write the planned change to ops/pending.md."}}
    return {}  # no opinion: the normal permission flow continues

options = ClaudeAgentOptions(
    hooks={"PreToolUse": [HookMatcher(matcher="Bash", hooks=[guard_kubectl])]})
```

- **Callback signature:** `(input_data, tool_use_id, context)`; every input has `session_id`, `cwd`, `hook_event_name`; tool events add `tool_name` / `tool_input` (plus `agent_id` / `agent_type` inside subagents). `tool_use_id` pairs a `PreToolUse` with its `PostToolUse`.
- **`PreToolUse` outputs** (in `hookSpecificOutput`): `permissionDecision` = `"allow"` | `"deny"` | `"ask"` | `"defer"`, `permissionDecisionReason` (sent to Claude), `updatedInput`. `"defer"` ends the turn with `stop_reason: "tool_deferred"`.
- **Rewriting input:** pair `updatedInput` with `"allow"` (auto-approve) or `"ask"`; omit the decision to send it through normal evaluation. **Never pair it with `"defer"`**: the rewrite is dropped.
- **Top-level outputs:** `systemMessage` is shown to the **user**, not the model; `continue` (`continue_` in Python) decides whether the agent keeps running.
- **Precedence:** across hooks, **`deny` > `defer` > `ask` > `allow`**. A hook `allow` does **not** skip deny or ask rules; a hook `deny` holds even in `bypassPermissions`.
- **Matchers:** `"Bash"`, `"Write|Edit"`, or a regex such as `"^mcp__"`. A `Write|Edit` matcher does not see a file written through `Bash`.
- **Async hooks:** return `{"async_": True}` (TS `async: true`) for logging or webhooks; the agent does not wait, so an async hook cannot block or rewrite.

> **EXAM TRAP**
> "Add a stronger instruction in the system prompt (MUST, NEVER)" is the wrong answer whenever a scenario needs a guarantee. The `permissionDecisionReason` on a deny should name the allowed path, so Claude changes course instead of retrying the same call.

6. Session state: resume, fork and interrupt

Operation	SDK	Claude Code CLI	Effect
Continue	`continue_conversation=True`	`claude --continue` (`-c`)	Most recent session in the current directory; same session ID
Resume	`resume=session_id`	`claude --resume <id-or-name>` (`-r`), `/resume`	Specific session; same ID; full prior context
Fork	`resume=id, fork_session=True` (`forkSession`)	`--resume <id> --fork-session`, `/branch`	**New** session ID with a copy of the history; original unchanged
Name	`rename_session()` (TS `renameSession()`)	`claude -n <name>` / `--name`, `/rename <name>`	Resume later with `claude --resume <name>`

- **Fork use case:** branch when you want to try another direction and keep the way back, e.g. after mapping an auth module, fork once to sketch an OAuth flow while the original thread stays untouched.
- **Fork is transcript-only:** the filesystem is shared. Forks that edit files collide unless isolated (worktrees, checkpoints).
- **Interrupt:** `await client.interrupt()` stops the running turn (streaming mode only). It does **not** clear the buffer: drain the interrupted turn's messages, including its `ResultMessage`, with `receive_response()` before reading the reply to your next query. Cancelling a tool this way does not fire `PostToolUseFailure`.
- **Resume after a crash:** persist `ResultMessage.session_id` as soon as you see it; after a process restart, `resume=session_id` (or `continue_conversation=True` for the latest session in the directory) restores the conversation. Disk changes are not rolled back.

> **DECISION RULE — RESUME OR NOT**
> a resumed session trusts every file it read earlier. If the code moved on, either name the changed paths in your first message so it re-reads them, or open a new session seeded with a short brief of what still holds; old tool output that contradicts the disk is worse than none.

7. Custom tools and tool errors

In-process tools: decorate an async handler with `@tool(name, description, input_schema)` (TS `tool()` with a Zod schema), wrap it with `create_sdk_mcp_server(name=..., version=..., tools=[...])`, pass the server in `mcp_servers`, and reference each tool as `mcp__<server>__<tool>` in `allowed_tools` and hook matchers. The handler returns `{"content": [{"type": "text", "text": ...}]}`.

```python
from claude_agent_sdk import tool, create_sdk_mcp_server, ToolAnnotations

@tool("stock_level", "Units on hand for one SKU in one warehouse",
      {"sku": str, "warehouse": str}, annotations=ToolAnnotations(readOnlyHint=True))
async def stock_level(args):
    try:
        units = await inventory.count(args["sku"], args["warehouse"])
    except TimeoutError:
        return {"content": [{"type": "text", "text":
                "Inventory API timed out: count UNKNOWN, not zero. Retry in about 30s."}],
                "is_error": True}
    return {"content": [{"type": "text", "text": f"{units} units"}]}

inv = create_sdk_mcp_server(name="inv", version="1.0.0", tools=[stock_level])
# ClaudeAgentOptions(mcp_servers={"inv": inv}, allowed_tools=["mcp__inv__stock_level"])
```

- **Parallelism:** custom tools run **sequentially by default**. Set `readOnlyHint=True` so Claude can batch the tool with other read-only calls (`Read`, `Grep`, `Glob`); `Edit`, `Write` and `Bash` stay sequential. Annotations are metadata, not enforcement.
- **Approval policy (TypeScript):** a tool-level `permission_policy` of `always_ask` stops `allowedTools` from approving it, so calls reach `canUseTool`; `always_deny` denies it at the deny-rule step.
- **Handler errors never stop the loop:** an uncaught exception becomes an error result carrying the raw exception text. Catching the error and returning `is_error: True` (TS `isError: true`) lets you write the message Claude actually needs.

Write the error for the next decision. Ask "who can fix this?" and say so in the message:

Who can fix it	Example	What the message should tell Claude
Claude, by changing arguments	SKU format wrong, unknown warehouse code	The accepted format or valid values
The user	Missing account number	Exactly which detail to ask for (`AskUserQuestion`)
Time	Upstream 503, timeout	That the value is unknown, plus when to retry
Nobody in this session	Caller not authorized, action outside policy	To stop this path and report it, not retry

> EXAM TRAP — UNKNOWN IS NOT EMPTY
> a tool that answers `0` or `[]` when its backend is unreachable gives Claude a confident wrong fact (it may reorder stock that exists). Return `is_error` with "unknown". A vague message has the same flaw in a different form: Claude has no way to know whether waiting would help.

8. Tool scoping and `tool_choice`

- **Three different levers:** `tools` decides what is **in context**; `allowed_tools` only **pre-approves**; a bare `disallowed_tools` entry **removes** a tool. Per subagent, use `AgentDefinition.tools` / `disallowedTools`.
- **Fewer, sharper tools:** each extra schema costs tokens and adds a chance of picking the wrong tool. A subagent that summarizes CI failures needs `Read` and an in-process `get_build_log(job_id)`, not `Bash` with network access.
- **Narrow beats general:** expose `get_build_log(job_id)` or `list_open_incidents(team)` instead of a raw shell or SQL tool when misuse is the risk; the handler enforces the scope in code.
- **`tools=[]` + MCP:** strips every built-in so Claude sees only your in-process tools, a common shape for embedded product agents.

`tool_choice` (Messages API):

Value	Behavior	Use for
`{"type": "auto"}`	Claude decides whether to call a tool (default when `tools` are provided)	Normal agent turns
`{"type": "any"}`	Must call **some** tool, Claude picks which	Guarantee a structured tool call when several output schemas exist
`{"type": "tool", "name": "classify_ticket"}`	Must call **that** tool	Force one specific call on this turn, then return to `auto`
`{"type": "none"}`	No tool calls (default when no `tools`)	Pure text turns

> EXAM TRAPS
> Forcing a tool every turn turns an agent into a fixed pipeline (it can never finish with `end_turn` text). Forced calls skip thinking. Manual extended thinking does not support `any` / `tool`, and some newer models reject them with a 400: use `auto` with strict tool definitions there (verify per model).

9. Permission modes, approvals and clarifying questions

Evaluation order for every tool call:

Step	Check	Notes
1	**Hooks**	Can deny outright; a hook `allow` does not skip steps 2–3
2	**Deny rules** (`disallowed_tools`, settings)	Deny even in `bypassPermissions` and even after a hook `allow`
3	**Ask rules**	Fall through to `canUseTool`, even in `bypassPermissions`
4	**Permission mode**	`bypassPermissions` approves; `acceptEdits` approves file operations; `plan` sends edits to the callback
5	**Allow rules** (`allowed_tools`, settings)	Approve matching calls

Step	Check	Notes
6	**`canUseTool` callback**	Skipped in `dontAsk` (the call is denied)

Mode	Behavior	Fits
`default`	No mode-based approvals; unmatched calls go to `canUseTool`	Interactive apps with an approval UI
`dontAsk`	Anything that would prompt is denied	Locked-down workers with an explicit `allowed_tools` list
`acceptEdits`	File edits and filesystem operations auto-approved	Trusted coding tasks in a sandbox
`plan`	Explores and plans; source edits never auto-approved	"Show me the plan first" flows
`auto`	A model classifier approves or blocks reviewed actions	Fewer prompts where available
`bypassPermissions`	Calls reaching step 4 run, apart from a few documented exceptions (TS also needs `allowDangerouslySkipPermissions`)	Disposable sandboxes only

- **Switch mid-session:** `await client.set_permission_mode("acceptEdits")` (TS `setPermissionMode`) applies to the next tool request, e.g. start in `plan`, then switch once a human approves the plan.
- **`canUseTool` results:** `PermissionResultAllow(updated_input=...)` or `PermissionResultDeny(message=...)` (TS `{ behavior: "allow" | "deny" }`). The deny message goes back to Claude.
- **Clarifying questions:** Claude asks through the `AskUserQuestion` tool, which also arrives in `canUseTool`. With an explicit `tools` list, include `"AskUserQuestion"`; return `answers` keyed by each question's text. It is not available inside subagents.
- **Subagent inheritance:** a parent in `plan`, `acceptEdits`, `auto` or `bypassPermissions` imposes that mode on subagents; an `AgentDefinition.permissionMode` of `"bypassPermissions"` is never applied.

> **EXAM TRAPS**
> `allowed_tools=["Read"]` with `bypassPermissions` still approves `Bash`, because unlisted tools fall through to the mode. A bare `allowed_tools` entry pre-approves that tool, so `canUseTool` never sees its calls. Checks that must see **every** call belong in a `PreToolUse` hook.

10. Subagent failures, limits and partial results

What the parent can observe: only the subagent's final message (plus `parent_tool_use_id` on streamed messages from inside it), the `SubagentStart` / `SubagentStop` hooks, and limit messages returned as Agent tool results.

Situation	What happens	Design response
Subagent hits its `maxTurns`	Output comes back **marked partial**	Resume it by `agentId` so it keeps its history, rather than spawning a fresh one
Spawn refused at the concurrency cap	Tool result: `Concurrent subagent limit reached`	Wait for running agents, or batch the work
Spend cap reached	`Budget limit reached`; background subagents stop; the query ends	Read `total_cost_usd`; finalize with what returned
Subagent prose says "mostly done"	Parent must guess what is missing	Require a fixed status block in every report

```
{ "status": "partial",
  "done": ["services/auth", "services/billing"],
  "not_done": ["services/search"],
  "reason": "maxTurns reached while tracing search indexer",
  "resume_hint": "agentId a1b2: continue from indexer/worker.py" }
```

- **Ask for the status block in `AgentDefinition.prompt`:** `complete | partial | blocked`, what was done, what was not, and why. The parent can then resume, re-scope or report honestly.
- **Trace with hooks:** `SubagentStop` gives `agent_id`, `agent_type` and `agent_transcript_path`, so you can log every child run without parsing the stream.
- **Treat reports as data:** recent Claude Code versions scan a subagent's final message for instruction-shaped text before the parent reads it; still never let a report trigger privileged actions on its own.

11. Long sessions and context compaction

Automatic compaction: near the context limit the SDK summarizes older history and emits a `SystemMessage` with `subtype: "compact_boundary"`. Instructions given early in the conversation may not survive the summary.

Technique	How
Persistent rules in `CLAUDE.md`	Rules that must survive compaction live in `CLAUDE.md` (loaded via `setting_sources`), not in the first user message
Steer the summary	Add a `CLAUDE.md` section telling the compactor what to keep (open TODOs, file paths, exact IDs)
`PreCompact` hook	Archive the full transcript before summarizing; `trigger` is `manual` or `auto`, `custom_instructions` carries any focus text
Manual `/compact`	Send `/compact` as the prompt string to compact on demand, e.g. between phases
Pinned facts file	Agent writes exact values (IDs, versions, thresholds) to a file and re-reads it; summaries blur numbers
Subagent delegation	Verbose exploration ("find every caller of `parse_token`") runs in a subagent; only the report enters the parent context
Lower a subagent's window	TS `AgentDefinition.autoCompactWindow` compacts a long-running subagent earlier
Trim tool output	Return only the fields Claude needs from custom tools

Symptoms of context degradation: inconsistent answers, generic advice instead of the specific files found earlier, forgotten constraints. Fix the cause (summary lost the detail) rather than repeating the question.

12. Structured results from agents

- **Top-level structured output:** pass `output_format={"type": "json_schema", "schema": ...}` and read the validated object from `ResultMessage.structured_output`; repeated schema failures end with `error_max_structured_output_retries`.
- **Subagent reports:** subagents return free text, so put the report shape in `AgentDefinition.prompt` and validate it in the parent.

- **Validate in the parent:** parse each report against the shape you asked for; a report that does not parse is a failed run, not a partial success.
- **Point at evidence:** a code-audit finding should carry the file and line it is about, so a reviewer can open it directly.

```
{ "finding": "Token refresh skips expiry check", "severity":
"high",
  "evidence": { "file": "auth/refresh.py", "line": 88, "observed":
"2026-03-02" },
  "confidence_basis": "read code path; no test covers it" }
```

13. Managed Agents

All Managed Agents endpoints require the beta header `managed-agents-2026-04-01` (set automatically by `client.beta.agents` / `environments` / `sessions`).

Primitive	Endpoint & mechanics
1. Agent	`POST /v1/agents` — versioned definition (`model` , `system` string — no `CLAUDE.md` hierarchy, `tools` , `mcp_servers` , `skills`). Built-in sandbox tools use **`{"type": "agent_toolset_20260401"}`** (in `/workspace`); custom tools (`{"type": "custom", ...}`) pause the session and emit **`agent.custom_tool_use`** until your backend posts **`user.custom_tool_result`** . Per-tool `permission_policy` (`always_allow` , `always_ask` , `auto`) gates execution via **`user.tool_confirmation`** .
2. Environment	`POST /v1/environments` — Anthropic-managed cloud container (`type: "cloud"` , `networking: {"type": "limited"}` allowlists) or self-hosted runner.
3. Session	`POST /v1/sessions` — binds a pinned `agent` version + `environment_id` , mounted `resources` (`file` , `github_repository` , `memory_store`), **`vault_ids`** (Vaults inject MCP/API credentials into outbound requests outside the container so raw secrets never enter the sandbox), and a spend `budget` .
4. Events	`POST /v1/sessions/{id}/events` & SSE `/events/stream` — `user.message` , `agent.message` , `agent.custom_tool_use` → `user.custom_tool_result` , `user.tool_confirmation` , `session.status_idle` with `stop_reason.type == "end_turn"` .

> **COMPLIANCE TRAP**
> **Managed Agents is stateful by design** (conversation history, container state and outputs stored server-side) and **cannot be used under a HIPAA BAA or Zero Data Retention (ZDR) agreement**. For regulated PHI or ZDR-governed data, run the loop yourself (Agent SDK or Messages API) on an eligible configuration.

14. Failure modes and exam traps

Symptom / scenario	Root cause → correct fix	Tempting wrong answer
Loop exits early whenever Claude says "Let me check..."	Termination keyed on text → loop on `stop_reason == "tool_use"` , stop on `end_turn`	Raising the iteration cap
Coordinator never delegates	`Agent` (alias `Task`) tool not in context/approved → add `"Agent"` to the explicit `tools` list (and `allowed_tools`)	Rewriting subagent descriptions only
Debug subagent re-runs the whole test suite to find the failure the parent already saw	Fresh subagents get only the Agent prompt → paste the failing test and stack trace into it	Telling it to "check the error above"
Four calls to a custom read-only tool run one at a time	Custom tools default to sequential → set `readOnlyHint=True`	Splitting the tool into four tools
`kubectl delete` ran in production despite a prompt rule	Prompt rules are guidance → `PreToolUse` hook on `Bash` denies with a redirecting reason	Repeating the rule in capitals
Agent reorders stock after the inventory API timed out	Tool returned `0` on timeout → `is_error` saying the count is unknown, with a retry hint	Retrying forever inside the handler
Hook path rewrite silently ignored	`updatedInput` paired with `"defer"` → pair with `"allow"` or `"ask"`	Moving the rewrite into the prompt
Resumed session calls a function a teammate deleted yesterday	History holds old file reads → name the changed paths so it re-reads them, or seed a new session with a short brief	`/compact` the old session
`canUseTool` never fires in production	Mode is `dontAsk` / `bypassPermissions` , or the tool has a bare `allowed_tools` entry → use `default` and drop the entry, or move the check to a hook	Adding a second callback
Agent "forgets" a rule after a long run	Rule lived in the first prompt and was summarized away at `compact_boundary` → move it to `CLAUDE.md`	Raising `max_turns`
Multi-tenant SDK app leaks memory across tenants or fails with `ENOENT` in TS	`setting_sources=[]` still loads auto memory, `~/.claude.json` , claude.ai MCP; TS `env` replaces `process.env`	`mcp_servers={}` without `strict_mcp_config=True`
PHI workload on Managed Agents "with deletion after each run"	Managed Agents is not BAA/ZDR eligible → Agent SDK / Messages API on eligible setup	Picking a US region

15. Glossary

Term	Definition
`stop_reason`	Why a response ended: `tool_use` (run tools, continue), `end_turn` (done), `max_tokens` , `pause_turn` , `refusal` , `stop_sequence` , `model_context_window_exceeded`
`ResultMessage.subtype`	SDK loop outcome: `success` , `error_max_turns` , `error_max_budget_usd` , `error_during_execution` , `error_max_structured_output_retries`
Hub-and-spoke	Coordinator routes all communication between isolated subagents
`Agent` tool (alias `Task`)	Tool Claude uses to spawn a subagent; must be available/approved for the coordinator
`AgentDefinition`	Programmatic subagent: `description` , `prompt` , `tools` , `model` , plus optional limits
`HookMatcher`	Registers SDK hook callbacks for an event and tool-name matcher

Term	Definition
`permissionDecision`	`PreToolUse` hook verdict: `deny` > `defer` > `ask` > `allow`
`fork_session`	Copies a session's history into a new session ID; original unchanged; filesystem not forked
`isError`	MCP tool-result flag marking a failed call (`is_error` in Python SDK tools)
`readOnlyHint`	Tool annotation that lets a custom tool run in parallel with other read-only calls

Term	Definition
`canUseTool` / **`set_permission_mode`**	Runtime approval callback (step 6) / switch permission mode mid-session
`tool_choice`	`auto` / `any` / `tool` (forced) / `none`
`compact_boundary` / **`PreCompact`**	System message emitted after automatic compaction / hook that runs before it
`interrupt()`	Stops the running turn in streaming mode; drain buffered messages afterwards
`agent_toolset_20260401` & Vaults	Managed Agents built-in toolset (`/workspace`) and server-side credential injection (`vault_ids`)