Agent Workflow Architecture Across Major AI Labs

What Anthropic, OpenAI, and Google Recommend for Building Reliable Agentic Systems — and Why

A Comparative Analysis — February 2026

Prepared for Botplotlib Project Research

1. Introduction

Between late 2024 and early 2026, all three major commercial AI labs—Anthropic, OpenAI, and Google DeepMind—published substantial guidance on how to design, build, and operate agentic AI systems. This report documents what each lab recommends, where their recommendations converge, where they diverge, and the stated reasoning behind their positions.

The analysis draws on primary sources: engineering blog posts, API documentation, technical whitepapers, developer cookbooks, production case studies, and the emerging cross-provider standards (AGENTS.md, Model Context Protocol) that have begun to unify the space. The goal is descriptive, not prescriptive—to map the landscape as it exists so that project-level decisions can be made against the actual state of the art rather than against assumptions about it.

2. Sources Reviewed

Provider	Document	Date	Primary Focus
Anthropic	Building Effective Agents	Dec 2024	Workflow patterns and composability
Anthropic	Writing Effective Tools for Agents	Sep 2025	Tool optimization via evaluation loops
OpenAI	A Practical Guide to Building Agents (PDF)	2025	Agent foundations, guardrails, orchestration
OpenAI	Shell + Skills + Compaction	Feb 2026	Primitives for long-running agent work
OpenAI	Harness Engineering (Codex case study)	Late 2025	Building a million-LOC codebase with agents
OpenAI	Unrolling the Codex Agent Loop	Jan 2026	Agent loop internals, prompt caching
OpenAI	GPT-5.2 Prompting Guide	Dec 2025	Structured output, compaction, reasoning effort
Google	Agents Whitepaper (Wiesinger et al.)	Sep 2024	Cognitive architecture, Extensions vs Functions
Google	Building Agents with Gemini 3	Nov 2025	Thought signatures, reasoning depth control
Google	Real-World Agent Examples with Gemini 3	Dec 2025	Production agent patterns across frameworks
Cross-provider	AGENTS.md specification	2025–2026	Universal repo-level agent context file
Cross-provider	MCP Tool Annotations spec	2025–2026	Standardized tool behavior metadata

3. Convergence: Where All Three Labs Agree

Despite different product strategies and business models, the three labs have arrived at strikingly similar core guidance. These shared positions represent the most reliable signals in the current landscape.

3.1 Definition and Structure of Agents

All three labs define agents as LLMs that iteratively use tools in a loop until a goal is met. Anthropic distinguishes between workflows (LLMs orchestrated through predefined code paths) and agents (LLMs that dynamically direct their own tool usage). OpenAI defines an agent as a system with instructions (what to do), guardrails (what not to do), and tools (what it can do). Google's whitepaper decomposes agents into three components: a model, tools, and an orchestration layer. The orchestration layer governs "a cyclical process" of information intake, reasoning, and action.

The shared structure is: loop(prompt → model reasoning → tool call → observe result → repeat or stop). All three labs note that this loop continues until the agent has reached its goal or a stopping condition is met. OpenAI adds that the agent should be able to halt execution and transfer control back to the user on failure.

3.2 Simplicity as the Default

All three labs recommend starting simple and adding complexity only when measurably justified. Anthropic puts it most directly: the most successful implementations weren't using complex frameworks or specialized libraries, but simple, composable patterns. Their recommendation is to find the simplest solution possible, noting that optimizing single LLM calls with retrieval and in-context examples is usually enough.

OpenAI's practical guide says to validate that a use case genuinely needs an agent before building one, and to maximize a single agent's capabilities before considering multi-agent systems. Google's whitepaper starts from single-agent ReAct loops and introduces multi-agent orchestration only for complex multi-domain tasks.

The reasoning is consistent across all three: agents trade latency and cost for better task performance, and that tradeoff is only justified when simpler approaches demonstrably fall short.

3.3 Tool Design as the Highest-Leverage Work

All three labs identify tool design—not prompt engineering—as the primary lever for agent performance. Anthropic reports that when building their SWE-bench agent, they spent more time optimizing tools than the overall prompt. Their recommendation is to invest as much effort in agent-computer interface (ACI) design as teams typically invest in human-computer interface (HCI) design.

OpenAI's guidance says each tool should have a standardized definition and emphasizes well-documented, thoroughly tested, and reusable tools. Google's whitepaper states that the quality of agent responses can be tied directly to the model's ability to reason about tasks, including the ability to select the right tools and how well those tools have been defined.

Specific tool design principles that appear across two or more providers:

- Clear, unambiguous parameter names. Anthropic gives the example of changing a parameter named “user” to “user_id.” OpenAI recommends thinking about tool design as writing a great docstring for a junior developer.
- Consolidated tools over granular ones. Anthropic recommends implementing tools that handle frequently chained, multi-step tasks in a single call. OpenAI recommends tools that consolidate functionality, handling multiple discrete operations under the hood.
- Token-efficient responses. Anthropic restricts tool responses to 25,000 tokens by default in Claude Code. OpenAI’s GPT-5.2 prompting guide enforces output verbosity constraints. Both recommend tools that return concise, semantically meaningful context rather than raw data dumps.
- Actionable error messages. Anthropic recommends prompt-engineering error responses to clearly communicate specific and actionable improvements, rather than opaque error codes or tracebacks. OpenAI implements the same pattern in Codex CLI, where tool errors include examples of correctly formatted input.

3.4 Evaluation-Driven Development

All three labs treat evaluation as the central engineering practice for agent development, not an afterthought. Anthropic recommends creating comprehensive evaluations that mirror real-world workflow complexity and using held-out test sets to ensure the system does not overfit. They describe an iterative loop: build tools, run evals, analyze transcripts, improve tools, re-evaluate.

OpenAI’s developer platform has formalized this into a “measure → improve → ship” loop with graders, prompt optimizers, and benchmarking pipelines. Their GPT-5.2 prompting guide emphasizes structured evals as a prerequisite before deploying any agent. Google integrates evaluation into its Agent Development Kit (ADK) and recommends framework-level testing for multi-step agent workflows.

The common reasoning: without evaluation, it is impossible to know whether changes to prompts, tools, or models improve or degrade the system. Agent behavior is non-deterministic, so measurement replaces intuition.

4. Anthropic: Workflow Patterns and ACI Design

Anthropic’s distinctive contributions center on a taxonomy of workflow patterns and a philosophy of agent-computer interface design.

4.1 Five Workflow Patterns

Anthropic’s “Building Effective Agents” post documents five composable patterns, presented in order of increasing complexity:

Pattern	Structure	When to Use (per Anthropic)
Prompt chaining	Sequence of LLM calls, each processing the previous output, with optional programmatic gates	Tasks cleanly decomposable into fixed subtasks; trades latency for accuracy

Routing	Classifies input and directs to specialized follow-up task	Complex tasks with distinct categories better handled separately
Parallelization	Multiple LLM calls run simultaneously; outputs aggregated	Independent subtasks, or when multiple perspectives increase confidence
Orchestrator-workers	Central LLM dynamically decomposes tasks and delegates to workers	Tasks where subtask count and nature cannot be predicted in advance
Evaluator-optimizer	One LLM generates a response; another evaluates and provides feedback in a loop	Clear evaluation criteria exist; iterative refinement provides measurable value

Anthropic frames these as composable building blocks that can be shaped and combined, not prescriptive architectures. Their guidance is to add multi-step patterns only when simpler solutions fall short, and to measure performance at each level of complexity.

4.2 Tool Optimization Through Collaborative Evaluation

Anthropic's "Writing Effective Tools" post (September 2025) introduces a methodology where agents are used to optimize the very tools that agents will use. The process involves: (1) building a prototype tool, (2) generating evaluation tasks grounded in real-world use, (3) running evaluations programmatically, (4) analyzing transcripts to identify failure patterns, and (5) using Claude Code to automatically refactor tools based on evaluation results.

Key findings from their internal tool optimization work: merely resolving arbitrary alphanumeric UUIDs to semantically meaningful identifiers significantly improved Claude's precision in retrieval tasks. Namespacing tools by service and resource (e.g., asana_projects_search vs. asana_users_search) had non-trivial effects on evaluation performance. And tool response format (XML, JSON, Markdown) impacts accuracy in ways that vary by task and agent.

They also introduce a response_format enum pattern where tools expose a parameter allowing the agent to request "concise" or "detailed" responses, reducing token consumption by up to two-thirds in their Slack tools example.

5. OpenAI: Skills, Compaction, and Harness Architecture

OpenAI's distinctive contributions address the operational challenges of long-running agents: how to make agent behavior reusable and versioned, how to manage context over extended workflows, and how large-scale agent-driven codebases are actually structured.

5.1 Skills as Versioned Procedure Bundles

OpenAI introduced Skills as a first-class API primitive in early 2026. A skill is a reusable bundle of files—a required SKILL.md manifest plus supporting scripts, templates, and assets—packaged as a folder. The model reads SKILL.md metadata to decide whether to invoke a skill; if it does, it reads the full instructions and executes using available tools.

OpenAI's stated rationale: as users scale from single-turn assistants to long-running agents, skills turn "prompt spaghetti" into maintainable, testable, versioned workflows. Skills are recommended for procedures where branching logic and formatting rules matter; for workflows that need code execution and local artifacts; and for keeping system prompts slim by offloading stable procedures into skills.

The skills concept aligns with the AGENTS.md convention. Anthropic has a parallel concept (SKILL.md files in Claude Code, introduced October 2025) with a growing ecosystem of community-contributed skills. Google has not yet standardized a skills format but supports similar behavior through custom MCP tools and ADK configurations.

5.2 Server-Side Compaction

OpenAI introduced server-side compaction as a Responses API primitive to address context window exhaustion in long-running agents. When context crosses a configurable threshold, compaction runs automatically in-stream, performing what OpenAI describes as "loss-aware compression" that preserves task-relevant information while reducing token footprint.

Their guidance is to treat compaction as a default long-run primitive, not an emergency fallback. Specific recommendations: reuse the same container across steps when stable dependencies and cached files are needed; pass previous_response_id so the model continues work in the same thread; and design workflows with continuity in mind from the start.

Anthropic does not currently offer an equivalent compaction primitive, relying instead on tool design that returns concise context and on purpose-built workflows that minimize accumulated state. Google addresses the problem primarily through large context windows (1M+ tokens in Gemini models) and Thought Signatures that preserve reasoning state across tool calls.

5.3 The Harness Engineering Case Study

In late 2025, OpenAI published a detailed case study of building a production application entirely with Codex agents—the first commit to an empty repository, scaffolded by Codex, scaling to over a million lines of code in weeks. Several architectural findings from this effort are notable:

- Repository as sole truth: From the agent's point of view, anything it cannot access in-context while running effectively does not exist. The team found that knowledge in Slack threads, Google Docs, or people's heads had to be pushed into the repo as versioned artifacts (code, markdown, schemas, executable plans) to be usable.
- Enforce invariants, not implementations: Rather than prescribing how agents write code, the team required structural properties (e.g., parsing data shapes at boundaries, following a fixed architectural layer model with validated dependency directions). These invariants were enforced mechanically via custom linters, themselves generated by Codex.
- Documentation for agent legibility: The repository was optimized first for the agent's legibility, not human developers'. The team treated this like improving navigability for new engineering hires—making it possible for an agent to reason about the full business domain directly from the repository.

5.4 Agent Foundations and Guardrails

OpenAI's practical guide introduces a three-type tool taxonomy: data tools (retrieve context), action tools (interact with systems), and orchestration tools (agents serving as tools for other agents). This taxonomy is not present in Anthropic's or Google's documentation.

The guide also documents two multi-agent orchestration patterns: a manager pattern (central agent delegates via tool calls and synthesizes results) and a decentralized pattern (agents hand off execution to peers based on specialization). OpenAI's stated guidelines for when to split agents: when prompts contain many conditional branches that make templates difficult to scale, or when tool overlap causes the agent to select incorrect tools despite clear descriptions.

On guardrails, OpenAI treats them as a distinct architectural layer. Their Agents SDK has built-in support for input guardrails (filtering inputs before generation), output guardrails (validating outputs before returning), and tracing (monitoring tool calls, agent handoffs, and guardrail triggers). The stated benefit: iterate on agents quickly by understanding what happened, not just what the output was.

6. Google: Cognitive Architecture and Reasoning Control

Google's distinctive contributions center on explicit cognitive architecture patterns, a clear separation between proposal and execution in tool use, and fine-grained control over model reasoning depth.

6.1 Cognitive Architecture Patterns

Google's agents whitepaper is the only source that explicitly names and compares cognitive architecture frameworks: ReAct (interleaved reasoning and action), Chain-of-Thought (intermediate reasoning steps), and Tree-of-Thoughts (exploring multiple reasoning paths for strategic lookahead). Their documentation walks through a complete ReAct cycle: Question → Thought → Action → Action Input → Observation → repeat N times → Final Answer.

This explicitness is distinctive. Anthropic's documentation describes similar loops but does not name or compare cognitive frameworks. OpenAI's documentation focuses on implementation patterns (single-agent, manager, decentralized) without prescribing reasoning frameworks.

6.2 Extensions vs. Functions: The Execution Boundary

Google draws a sharp distinction between two tool execution models. Extensions bridge the gap between an agent and an API—the agent calls the API directly and receives results. Functions work differently: the model outputs a function name and its arguments, but does not make a live API call. Execution is delegated to the client application, which decides whether and how to proceed.

Google's stated reasoning for the distinction: Functions give the developer more control over the execution boundary. The model proposes; the client disposes. This allows for human-in-the-loop patterns, validation of proposed actions before execution, and integration with systems where direct API access would be inappropriate.

This pattern has implications for governance. A system where the agent proposes actions but cannot execute them autonomously creates a natural permission boundary—the deterministic

layer between proposal and execution can enforce invariants, log provenance, and require approval for sensitive operations.

6.3 Reasoning Depth and State Management

Gemini 3 introduced two mechanisms for controlling agent behavior at the model level:

- Thinking level: A per-request parameter (high, medium, low) that adjusts the depth of the model's internal reasoning. High for deep planning and bug finding; low for high-throughput tasks with latency comparable to previous-generation models. Google frames this as giving developers granular control over the cost-latency-quality tradeoff.
- Thought Signatures: Encrypted representations of the model's internal reasoning generated before each tool call. Passing these signatures back in conversation history allows the agent to retain its exact train of thought across multi-step execution, preventing what Google calls "reasoning drift" in long sessions.

OpenAI has a parallel reasoning effort parameter in GPT-5.2, described in their prompting guide as a control for production agents that need to balance reliability with throughput. Anthropic does not currently expose a per-request reasoning depth control.

7. Divergences

The providers disagree or emphasize differently on several dimensions.

7.1 Autonomy and Control

Dimension	Anthropic	OpenAI	Google
Default posture	Workflows first; agents only when justified	Agents as trusted delegates within guardrails	Agents as autonomous goal-seekers with human checkpoints
Framework guidance	Start with raw API calls; frameworks obscure prompts and responses	Agents SDK provided; code-first approach preferred over declarative graphs	Heavy framework ecosystem (LangChain, CrewAI, LlamaIndex, ADK)
Multi-agent stance	Skeptical; maximize single agent before splitting	Documented patterns but general recommendation is single agent first	Enthusiastic; multi-agent collaboration highlighted across case studies
Stated reasoning	Complexity trades latency and cost for performance; justify the tradeoff	More agents can help but introduce overhead; split only on evidence	Specialized agents with distinct roles collaborate more effectively

7.2 Context Management Strategy

The three labs address context window exhaustion differently:

Strategy	Provider	Mechanism	Tradeoff
Tool-level efficiency	Anthropic	Tools return concise, semantic output; 25k token cap per tool response; response_format enum for verbosity control	Requires careful tool design; no help for accumulated conversation state
Server-side compaction	OpenAI	Automatic compression when context crosses threshold; loss-aware; preserves task-relevant information	Potential information loss; adds a dependency on the provider's compression quality
Large context + state tokens	Google	1M+ token context windows; Thought Signatures preserve reasoning state across tool calls	Higher cost per request; does not solve the problem, only delays it

These are not mutually exclusive. A system could combine tool-level efficiency (reducing what goes into context), compaction (compressing what accumulates), and large context windows (providing headroom). But each lab leads with a different primary strategy, reflecting their respective model capabilities and API designs.

7.3 Tool Execution Model

The providers differ on who executes tool calls:

- Anthropic: The agent directly executes tool functions defined by the developer. The focus is on making those functions well-designed and token-efficient.
- OpenAI: Tools can execute in hosted shell environments managed by OpenAI (sandboxed, with controlled internet access) or locally. Skills bundle execution environment with instructions.
- Google: The Extensions model has the agent call APIs directly. The Functions model has the model propose a call and the client execute it. The developer chooses which model to use per tool.

The practical implication: Google's Functions pattern and OpenAI's approval modes both create explicit boundaries between agent proposal and system execution. Anthropic achieves similar boundaries through tool design (hard-separating code actions from public speech actions) rather than through a distinct API-level execution model.

8. Emerging Cross-Provider Standards

8.1 AGENTS.md

AGENTS.md is a plain Markdown file placed at the repository root that provides coding agents with project-specific context: build commands, conventions, testing expectations, and explicit boundaries. It was introduced in mid-2025 through collaborative efforts including OpenAI Codex, Google Jules, Cursor, and others. It is now stewarded by the Agentic AI Foundation under the Linux Foundation.

Adoption data from early 2026: over 60,000 repositories contain AGENTS.md files. Research by Mohsenimofidi et al. (2026) found that the files serve as persistent configuration mechanisms encoding architectural constraints, build commands, and workflow conventions. Chatlatanagulchai et al. (2025) found that context files are actively maintained, structurally consistent, and focus on functional development instructions. Community reports suggest projects with detailed AGENTS.md files see 35–55% fewer agent-generated bugs, though this figure has not been independently verified.

Anthropic supports the convention through CLAUDE.md (the original context file for Claude Code), with symlinks to AGENTS.md as the common pattern. OpenAI co-created the standard. Google supports it through Gemini CLI. The convention is also supported by Cursor, Windsurf, Aider, RooCode, and others.

8.2 MCP Tool Annotations

The Model Context Protocol, originally created by Anthropic and now broadly adopted, includes a standardized annotation vocabulary for tools. Four annotations are defined:

Annotation	Type	Meaning
readOnlyHint	boolean	Tool does not modify its environment
destructiveHint	boolean	Tool may perform destructive updates (delete, overwrite)
idempotentHint	boolean	Repeated calls with same arguments have no additional effect
openWorldHint	boolean	Tool interacts with external entities outside the user's system

These annotations are hints, not guarantees—MCP clients may choose how to act on them. Common implementations: auto-approve tools annotated as readOnly; require human confirmation for destructive or openWorld tools; optimize caching for idempotent tools.

The annotations provide a standardized mechanism for encoding governance constraints at the tool level. A tool marked destructiveHint: true, openWorldHint: true (e.g., publishing content or opening a pull request) can be automatically gated by any MCP-compatible client, regardless of which AI lab's model powers the agent.

9. Feature Comparison Matrix

Capability	Anthropic	OpenAI	Google
Workflow pattern taxonomy	5 patterns (chaining, routing, parallelization, orchestrator-workers, evaluator-optimizer)	Single-agent, manager, decentralized; prompt templates for complexity management	ReAct, Chain-of-Thought, Tree-of-Thoughts; framework-level orchestration
Skills / versioned procedures	SKILL.md in Claude Code	Skills API with SKILL.md manifest, shell execution, versioning	No standardized format; uses MCP tools and ADK configs

Context compaction	Tool-level token efficiency; no explicit compaction primitive	Server-side compaction in Responses API; previous_response_id for stateful continuation	1M+ token context windows; Thought Signatures for state preservation
Reasoning effort control	Not exposed as per-request parameter	reasoning_effort parameter in GPT-5.2 API	thinking_level parameter (high/medium/low) in Gemini 3
Tool execution model	Agent executes tool functions directly	Hosted shell (server-side) or local execution; sandboxed with network controls	Extensions (direct) vs Functions (propose-only); developer chooses per tool
MCP support	Creator of MCP; full support	Adopted MCP standard; Apps SDK integration	Partial MCP support; migration from legacy Tool Calling API by March 2026
AGENTS.md support	Via CLAUDE.md symlink	Co-created; native Codex support	Gemini CLI support
Guardrails architecture	Hard permission boundaries; separate code actions from speech actions	Distinct architectural layer; input guardrails, output guardrails, tracing in SDK	Confirmation before critical actions; framework-level safety controls
Multi-agent orchestration	Discouraged unless single agent demonstrably insufficient	Manager and decentralized patterns documented; guidelines for when to split	Enthusiastic support via CrewAI, ADK, LangGraph integrations
Production eval integration	pytest-mpl ecosystem; eval-driven tool optimization methodology	Evals platform with graders, prompt optimizer, benchmarking pipelines	ADK testing; framework-level evaluation support

10. Summary of Findings

The state of the art in AI agent architecture, as documented by the three major labs, has converged on several stable principles while diverging on operational details.

Stable consensus: Agents are tool-using loops. Start simple and justify complexity with measurement. Invest more in tool design than prompt engineering. Build evaluation harnesses before building the system. Use standardized context files (AGENTS.md) and tool metadata (MCP annotations) for cross-platform compatibility.

Anthropic's emphasis: Composable workflow patterns over autonomous agents. Tool optimization as a collaborative, evaluation-driven practice. Agent-computer interface design as a first-class engineering discipline.

OpenAI's emphasis: Skills as versioned, portable procedure bundles for long-running agents. Server-side compaction as an infrastructure primitive. Repository-as-truth and invariant

enforcement for agent-generated codebases. Guardrails as a distinct architectural layer with tracing.

Google's emphasis: Explicit cognitive architecture patterns (ReAct, CoT, ToT). A sharp distinction between tool proposal and tool execution (Functions vs Extensions). Per-request reasoning depth control and state preservation via Thought Signatures.

Cross-provider standards: AGENTS.md (60,000+ repos, Linux Foundation stewardship) and MCP tool annotations (readOnlyHint, destructiveHint, idempotentHint, openWorldHint) are converging into a portable governance layer that works regardless of which model powers the agent.

End of Report