Agent Development Kit (ADK) by Michaël Bettan

Definition: Agent Development Kit (ADK) is Google's open-source, code-first framework (**Python, TypeScript, Go, Java, and Kotlin**) for building, evaluating, governing, and deploying production AI agents. Rather than leaving teams to hand-roll conversation state, tool dispatch, retries, and trace instrumentation around a raw model endpoint, ADK provides the agent harness as modular infrastructure: an event-sourced execution runtime, typed tool and protocol connectors, multi-tier state and memory, multi-agent orchestration, deterministic callback/plugin guardrails, trajectory evaluation, and one-command cloud deployment.

Why ADK? (What a raw model API lacks): A bare model endpoint is **stateless** (retains zero memory across HTTP requests unless using server-side interaction handles), **passive** (emits tool-call intents as JSON text without executing anything), **unstructured in control flow** (no durable loops, fan-out joins, checkpoints, or hard iteration caps), and **ungoverned** (no tenant isolation, approval gates, or tool-boundary PII scrubbing). ADK supplies all four layers cleanly decoupled from model weights.

> MENTAL MODEL
> Treat the foundation model (**Gemini 4 Argon**, **Gemini 3.8 Flash**, **Claude**, or **Gemma 4**) as the **brain**, and ADK as the **nervous system and skeleton**: hands (`Function Tool` , `McpToolset`), short- and long-term memory (`SessionService` , `MemoryService`), spinal reflexes (`before_tool_callback` , `BasePlugin`), and the motor loop (`Runner` / `Workflow`) that coordinates when the brain reasons and when tools act.

End-to-end lifecycle: 1. Define (`LlmAgent` / `Workflow` / `ManagedAgent` + tools + instructions) → **2. Run** (`Runner` loads session state, invokes the model or **Interactions API**, executes tools, and commits each step as an immutable `Event`) → **3. Evaluate** (`adk conformance` + `adk eval` in CI, paired with **Agent Platform Gen AI Evaluation Service** in cloud) → **4. Deploy** (**Agent Runtime**, **Cloud Run**, or **GKE**).

1. When to use ADK, framework choices and multi-language parity

> RULE 0
> Begin with a single grounded model call. Step up to deterministic workflows—and only then to autonomous multi-turn agents—**when benchmark evaluations prove the simpler tier cannot meet accuracy or adaptability targets.**

Architectural tier	Core mechanism	Makes runtime routing decisions?	Executes external actions?
Single LLM call	Stateless prompt → completion	✅ (single step)	❌ (can only emit tool-call JSON)
Deterministic workflow	Predefined DAG or pipeline orchestrating models and tools	❌ (code/edges decide the path)	✅
Autonomous agent	Model-directed loop executing tools, inspecting observations, and updating state until goal completion	✅ (multi-step adaptive)	✅
ADK	Production framework providing both deterministic workflows and autonomous agent loops under one event ledger	n/a	n/a

Workload shape	Recommended pattern	Architectural rationale
Single prompt (+ retrieval) answers reliably	Augmented LLM / RAG call	Adding an agent loop inflates TTFT, token spend, and variance for zero gain
Fixed, predictable multi-step pipeline (e.g., parse → validate → summarize)	**ADK 2.0 `Workflow` graph** (or ADK 1.x `SequentialAgent` / `ParallelAgent` / `Loop Agent`)	Deterministic execution order, bounded cost, straightforward debugging
Read-only Q&A over enterprise SharePoint/Drive corpora with ACLs	Managed **Agent Search** / RAG pipeline	Retrieval grounding solves missing knowledge without open-ended tool loops
Next action depends dynamically on intermediate tool findings	**`LlmAgent`** (ReAct / planner loop)	Control path cannot be hardcoded ahead of time
Long-horizon cloud sandbox execution using Google's hosted harness	**`ManagedAgent`** (backed by **Interactions API**)	Offloads loop, remote container sandbox, and server-side state to Google
Strictly deterministic compliance or actuarial calculation	Plain code / classical ML	Zero stochastic variance, 100% reproducible auditability

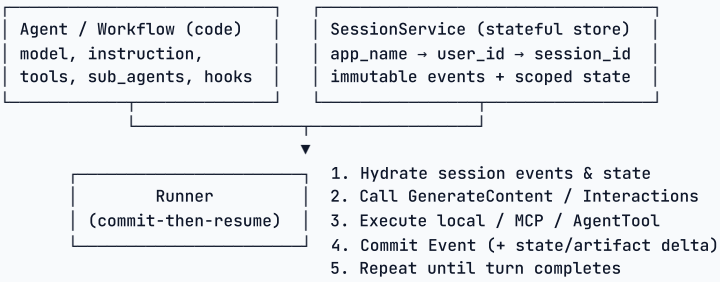
The "agent theater" litmus test: Splitting a system into multiple agents is justified **only when specialists require distinct tool permissions, isolated data scopes, an independent auditor (`include_contents='none'` + verification tools), or explicit human sign-off boundaries.** Breaking a single prompt into four chatty persona agents sharing identical context and zero tools burns ~4× the tokens and ~2× the latency without improving accuracy.

SDK language	Package / module	ADK 2.0 graph **`Workflow`** & task modes	Primary strengths & parity nuances (Oct 2026)	Typical Google Cloud deploy target
Python	`google-adk`	✅ Full support	Reference flagship SDK: earliest access to **`ManagedAgent`** , `SkillToolset` + `GCPSkillRegistry` , `adk eval` / `AgentEvaluator` / `adk conformance` , `VertexAiMemoryBankService` , and `run_live()` . Agent Runtime upload excludes the API server (platform injects it).	**Agent Runtime**, Cloud Run, GKE
TypeScript	`@google/adk`	✅ Full support	Node.js / full-stack web integration, MCP servers, experimental **`RoutedLlm`** dynamic model switcher, and `BuiltInCodeExecutor` without single-tool mixing limits on modern Gemini models (`Agent Search` built-in tool not yet in TS).	**Cloud Run**, GKE
Go	`google.golang.org/adk`	✅ Full support	High-concurrency, low-memory compiled microservices with native goroutine fan-out, MCP, OpenTelemetry, and **Agent Runtime** support (Go deploy payload bundles its own dedicated ADK API server binary).	**Agent Runtime**, Cloud Run, GKE
Java & Kotlin	`com.google.adk`	❌ (ADK 1.x workflow agents)	Enterprise Spring/JVM integration (`LlmAgent.builder()` in Java; type-safe DSL in Kotlin) with `SequentialAgent` / `ParallelAgent` / `LoopAgent` , `FunctionTool` , `McpToolset` , and callbacks; built-in `GoogleSearchTool` + custom tools must be isolated via `AgentTool` .	**Cloud Run**, GKE

> **COMMON TRAP**
> Confusing the **Gen AI SDK** (`google-genai`) with **ADK** (`google-adk`). `google-genai` is the model/interactions client library (supporting `generate_content`, `client.interactions.create`, and basic in-process function calling), whereas ADK provides multi-user `SessionService` isolation, the persistent `Event` ledger, multi-agent transfers, lifecycle guardrails (`BasePlugin`), conformance/trajectory evals, and runtime deployment.

2. Runtime architecture: Agent, Runner, SessionService and ManagedAgent

ADK enforces a strict separation between **stateless agent definitions**, **stateful session storage**, and the **event-loop engine**:



Core primitive	Responsibility	Production engineering rule
`Agent` (`Llm Agent`)	Stateless blueprint: model, instructions, tools, sub-agents, planners, callbacks	Store **zero** per-user attributes on `self`; one singleton safely handles thousands of concurrent requests
`Managed Agent`	Data-plane client for Google's **Managed Agents API** via the **Interactions API**	Connects to an existing `agent_id` (`global` endpoint on Agent Platform); runs loop & tools remotely
`BaseLlm`	Model abstraction (`"gemini-4-argon"`, `"gemini-3.8-flash"`, `LiteLlm(...)`, `Claude(...)`, `Gemma(...)`)	Mix model families across coordinator and worker agents without changing session or callback code
`Runner`	Executes turns (`run_async`) or bidirectional audio/video (`run_live`)	**Commit-then-resume:** persists every `Event` to `SessionService` before advancing to the next step
`Session Service`	Manages `Session` objects keyed by `app_name` / `user_id` / `session_id`	`InMemorySessionService` (local dev) → `DatabaseSessionService` (`postgresql+asyncpg` / `sqlite+aiosqlite`) or `VertexAi SessionService` (prod; `app_name` = Reasoning Engine ID)
`Memory Service`	Cross-session long-term recall per user	`InMemoryMemoryService` (unit tests) → `VertexAiMemoryBankService` (consolidated facts) / `VertexAiRagMemory Service`
`Artifact Service`	Versioned binary/file blobs (`"user:filename"` persists across sessions)	`InMemoryArtifactService` / `File ArtifactService` (dev) → `GcsArtifact Service` (Cloud Storage in prod)
`App`	Root application envelope binding `root_agent`, `plugins=[...]`, and runtime policies	Attach `ResumabilityConfig`, `Events CompactionConfig`, and `ContextCache Config` here—never on `LlmAgent`

`LlmAgent` (`GenerateContent`) vs. `ManagedAgent` (`Interactions API`): - **Standard `LlmAgent`:** Your ADK `Runner` owns the orchestration loop, re-sending compacted conversation history (and cryptographic **Thought Signatures**) over `Generate Content` while executing custom Python `FunctionTool`s and `McpToolset` connections inside your runtime environment. - **`ManagedAgent` (Preview):** Invokes a pre-existing hosted agent (`agent_id`, such as `antigravity-preview-...` or a custom agent provisioned via the **Agents API** control plane) over the **Interactions API** (`stream=True`) with `environment={'type': 'remote'}`. Reasoning, built-in tools (`google_search`, `ToolCodeExecution`), and sandbox execution happen inside Google's managed cloud infrastructure while partial and final events stream into your ADK workflow. - **State split in `ManagedAgent`:** ADK never re-transmits prior conversation turns to a `ManagedAgent`. Instead, the local `Session` records only two lightweight pointers on emitted events— **`previous_interaction_id`** and **`environment_id`**—and reuses them on subsequent turns so the server-side `Interaction` and remote container sandbox continue seamlessly. - **Direct `Interactions API` (`client.interactions.create`) for long-horizon tasks:** When calling `gemini-4-argon` or Deep Research directly via `google-genai`, setting `background=True` runs multi-minute tasks asynchronously on the server (preventing ~60s HTTP gateway timeouts and unlocking long decodes up to the unified `1,048,576`-token window). Chaining a follow-up request via `previous_interaction_id` while the prior interaction is still `in_progress` returns **`400 Bad Request`**—poll until `completed` first.

```
# ManagedAgent runs server-side via Interactions API; coordinator wraps it
as AgentTool
web_researcher = ManagedAgent(
    name="managed_web_researcher",
    description="Grounded web synthesis in Google's remote managed
environment.",
    agent_id=os.environ.get("MANAGED_AGENT_ID", "antigravity-preview-09-
2026"),
    environment={"type": "remote"},
    tools=[google_search],  # Server-side tools only; local tools raise
NotImplementedError
)
coordinator = LlmAgent(
    name="portfolio_coordinator",
    model="gemini-4-argon",
    instruction="Combine internal CRM lookups with external web research.",
    tools=[AgentTool(agent=web_researcher)],
)
```

Context hierarchy & Event ledger: `InvocationContext` (full per-turn runtime handle), `ReadonlyContext` (passed to `InstructionProvider` callables for read-only `ctx.state`), `CallbackContext` (agent/model hooks: mutable state + artifacts), and `ToolContext` (tools + tool hooks: state, artifacts, credentials, `search_memory`, confirmation, and `actions`). Every state change, tool call, transfer, or escalation is recorded as an `Event` (`author`, `invocation_id`, `content`, `actions.state_delta`, `actions.artifact_delta`), creating a single source of truth for `adk web` time-travel inspection, Cloud Audit trails, and trajectory grading.

> **COMMON TRAP**
> Passing client-executed Python functions or an `McpToolset` into `Managed Agent(tools=[...])` raises **`NotImplementedError`**, and pointing `ManagedAgent` at a regional Agent Platform endpoint fails because the Managed Agents API is served exclusively from **`global`**. Keep custom/MCP tools on an `LlmAgent` and compose `ManagedAgent` beside it.

3. Agent types and deterministic control flow

Every ADK agent inherits from `BaseAgent`. Only `LlmAgent` (and `ManagedAgent`) consult a model to decide next steps; **workflow agents route deterministically in code with zero LLM routing overhead.** *(In ADK 2.0 for Python, TypeScript,*

and Go, `SequentialAgent`, `ParallelAgent`, `LoopAgent`, and custom `BaseAgent` orchestrators are superseded by unified graph `Workflow`s in §5, though both patterns remain active in Java/Kotlin and exam scenarios.)

Agent class	Routing mechanism	Ideal workload	Critical configuration
`LlmAgent` (`Agent`)	Dynamic model reasoning	Tool selection, open-ended synthesis, triage	`model`, `output_key`, `output_schema`, `planner`, `tools`, `sub_agents`
`Sequential Agent`	Strict linear order `A → B → C`	Staged pipelines where step `N` consumes step `N-1`'s `output_key`	`sub_agents=[...]`
`Parallel Agent`	Concurrent execution (`branch` isolated, shared `session.state`)	Multi-source retrieval, independent compliance checks	Unique `output_key` per parallel branch
`LoopAgent`	Iterative cycle until exit signal or cap	Draft → critique → revise loops, polling	`max_iterations` + tool setting `actions.escalate = True`
Custom (`Base Agent`)	Arbitrary async code	Deterministic rule routers, custom state machines	Override `async def _run_async_impl(self, ctx)`

```
def approve_patch(tool_context: ToolContext) → dict:
    tool_context.actions.escalate = True           # Halts enclosing
LoopAgent immediately
    tool_context.actions.skip_summarization = True  # Suppresses redundant
post-tool LLM turn
    tool_context.state["temp:verified"] = True      # Proves convergence vs
max_iterations cap
    return {"status": "verified"}

patch_loop = LoopAgent(name="patch_and_verify", max_iterations=3,
sub_agents=[
    LlmAgent(name="patcher", model="gemini-3.8-flash", instruction="Write
patch for {cve}.", output_key="diff"),
    LlmAgent(name="auditor", model="gemini-4-argon",
include_contents="none",
             instruction="Verify {diff}. Call approve_patch only if tests
pass.",
             tools=[run_tests, approve_patch], output_key="audit"),
])
root_agent = SequentialAgent(name="pipeline", sub_agents=[gather_agent,
patch_loop])
```

- **State templating, `ParallelAgent` race safety & `InstructionProvider`:** Setting `output_key="cve"` writes the agent's final text to `state["cve"]`, which downstream instructions reference via `{cve}` (use `{audit?}` when optional so turn 1 doesn't raise `KeyError`, or `{artifact.report}` to inline text artifacts). Inside a **`ParallelAgent`**, all branches execute concurrently against the **same shared `session.state`**: if two parallel workers write to the same `output_key` or mutate a shared list key concurrently, they trigger last-write-wins data loss, so assign a **distinct `output_key` per parallel branch** and aggregate in a downstream merger step. Whenever an instruction includes literal curly braces (such as JSON schemas `{"status": "ok"}` or regex quantifiers), supply a callable `def build_instruction(ctx: ReadonlyContext) → str:` so ADK bypasses `{var}` placeholder parsing.
- **Silent `LoopAgent` exhaustion:** Reaching `max_iterations` exits `LoopAgent` **silently without raising an exception**, allowing any subsequent agent in an outer `SequentialAgent` to run on unapproved output. Always write an explicit convergence flag (`state["temp:verified"] = True`) inside your exit tool and check it in the downstream step's `before_agent_callback`.
- **Independent verification:** Prevent self-grading bias in critic loops by isolating the verifier (`include_contents="none"`, dedicated deterministic test/linter tools, and explicit pass/fail criteria).

> **COMMON TRAP**
> Expecting `SequentialAgent` to skip irrelevant steps dynamically or expecting `LoopAgent(max_iterations=3)` to stop automatically when tests pass. `SequentialAgent` always runs every child in sequence, and `LoopAgent` always runs all `max_iterations` unless a tool or callback sets `escalate = True`.

4. Multi-agent delegation: transfer, AgentTool and remote agents

Delegation pattern	Invocation mechanics	Who speaks to the user next?	Session & context boundary	Best architectural fit
`sub_agents` (LLM transfer)	ADK injects `transfer_to_agent(agent_name=...)`; LLM selects target using each child's **`description`**	**Target sub-agent** assumes control	Shares the same `session.events` and `session.state`	Full conversational handoff to a domain specialist (billing vs. tech support)
`AgentTool(agent=...)`	Parent invokes child like a typed function (`skip_summarization=True` returns raw output)	**Parent coordinator** keeps control	Runs in an isolated child session while **merging `state_delta` / `artifact_delta` (`output_key`) back**	Modular sub-task execution without polluting the coordinator's chat history
`RemoteA2aAgent`	Connects to an external service via its A2A Agent Card (`/.well-known/agent-card.json`)	Depends on transfer vs. `AgentTool` wrapping	Strictly opaque network boundary: only A2A messages, tasks, and artifacts cross	Cross-team, multi-language, or cross-organization agent federation

- **Scoping transfers, hop caps & tree rules:** Use `disallow_transfer_to_parent=True` on one-shot triage helpers (blocks upward handoff and routes the user's next turn back to `root_agent`) and `disallow_transfer_to_peers=True` to enforce strict hub-and-spoke governance through a coordinator. Bound coordinator-worker recursion and ping-pong transfers via **`max_hop_count`** (alongside `RunConfig(max_llm_calls=...)`) and keep `sub_agents` nesting shallow (`⩽ 2-3` levels) so latency and context bloat do not compound. Every `Agent` instance obeys the **single-parent rule** (`ValueError` if attached to two parents' `sub_agents`); use factory functions or `agent.clone()` to reuse definitions.
- **ADK 2.0 collaborative sub-agent modes (Python, TS, Go):**
- `mode="chat"` (classic default): injects one shared `transfer_to_agent` tool; the specialist takes over user interaction.
- `mode="task"`: specialist can ask the user clarifying questions across turns, then invokes `finish_task` to hand structured output back to the parent coordinator.
- `mode="single_turn"`: generates a dedicated delegation tool per specialist, runs autonomously with zero direct user prompts, and supports parallel fan-out calls from the coordinator.
- **Cost-efficient routing:** Replace frontier LLM routers with deterministic `Workflow` conditional edges (`Event(route=...)`), embedding similarity, or **`gemini-3.5-flash-lite`** (`thinking_level="minimal"`) when intent buckets are well-defined.

> **COMMON TRAP**
> Writing instructions like `"Call transfer_to_billing()"` in `mode="chat"`. ADK does **not** create per-agent transfer functions in chat mode; it exposes a single `transfer_to_agent(agent_name="billing")` tool (per-specialist tools are generated only in ADK 2.0 `mode="single_turn"` / `"task"` or via `AgentTool`).

Choosing the orchestration boundary: Workflow Agents vs. Graph Workflows vs. Remote A2A Agents

The delegation table above decides who speaks next. This table decides where the process boundary sits, which in turn fixes what can be isolated, who can govern it and what each handoff costs.

Pattern	State sharing	Deployment boundary	Handoff latency	Fault and identity isolation	Governance granularity	Choose when
Workflow agents (`SequentialAgent`, `ParallelAgent`, `LoopAgent`)	one `session.state`; children exchange data through `output_key` and `{key}` templates	single process, single deployable unit	in-process call	none: a crash or runaway loop takes down every child, and all children act as the deployment's one principal	ADK callbacks and plugins inside your code	fixed pipeline owned by one team, or Java/Kotlin and ADK 1.x codebases
Graph workflows (`Workflow(edges=[...])`)	typed `node_input` between nodes plus the shared `session.state`	single process, single deployable unit	in-process edge evaluation	same as workflow agents; explicit edges make the failing node easier to locate	callbacks plus explicit routes and human-input nodes (`RequestInput`)	branching, loops and approval pauses inside one service (Python, TypeScript, Go)
Remote A2A agents (`RemoteA2aAgent` to a peer on Agent Runtime)	nothing shared; only A2A messages, task status and artifacts cross the wire	separate service per peer, possibly another project, language or organization	network round trip plus gateway policy evaluation on every call	strong: each peer scales and fails on its own and runs under its own Agent Identity principal	agent policies enforced by Agent Gateway (IAM Unified Access Policies, Principal Access Boundary, Model Armor, semantic governance) on top of each peer's own callbacks	different owners, release cycles, runtimes or privilege tiers, and partner agents

- **Anti-pattern (network for its own sake):** splitting one team's three-step pipeline into three A2A services adds network hops, three deployments and three policy surfaces with no isolation gain; keep it as a `Workflow`.
- **Anti-pattern (privilege mixing in one process):** attaching a read-only researcher and a refund-issuing specialist as `sub_agents` of one deployment gives both the same principal, so no platform policy can tell them apart; move the privileged specialist behind A2A with its own identity.

5. Graph workflows (ADK 2.0), planners and Gemini 4 Argon thinking levels

ADK 2.0 Graph Engine (`Workflow`): Available across **Python, TypeScript, and Go**, `Workflow(edges=[...])` unifies sequential pipelines, parallel fan-out/fan-in, cyclic loops, and conditional routing into one directed graph combining `Llm` `Agent`s, plain code nodes (`FunctionNode`), `ManagedAgent`s, and nested `Workflow`s. Return values pass as typed `node_input`, and function parameters matching `session.state` keys are dependency-injected automatically.

```
def classify_severity(node_input: str) → Event:
    target = "deep_architect" if "architecture" in node_input.lower() else
"fast_responder"
    return Event(output=node_input, route=target)  # Route key for
conditional edge

deep_architect = LlmAgent(name="deep_architect", model="gemini-4-argon",

planner=BuiltInPlanner(thinking_config=types.ThinkingConfig(thinking_level="
high")))
fast_responder = LlmAgent(name="fast_responder", model="gemini-3.8-flash",

planner=BuiltInPlanner(thinking_config=types.ThinkingConfig(thinking_level="
low")))

root_agent = Workflow(name="support_graph", edges=[
    (START, classify_severity),
    (classify_severity, {"deep_architect": "deep_architect",
"fast_responder": "fast_responder"}),
    (("deep_architect", "fast_responder"), END),
])
```

Graph capability	LangGraph equivalent	ADK 2.0 **`Workflow`** (Python / TS / Go)
Conditional branching	`StateGraph` + `add_conditional_edges`	`Workflow(edges=[(router_fn, {"route": node})])` + node returning `Event(output=..., route="target")`
Parallel fan-out & synchronization	Parallel edges + `Annotated[..., operator.add]` reducers	Edge tuple `(START, ("branch_a", "branch_b"), join_node)` + **`JoinNode`** (`dict[node_id, output]`); no state reducers, so use distinct state keys

Graph capability	LangGraph equivalent	ADK 2.0 `Workflow` (Python / TS / Go)
Durable checkpointing	Checkpointer keyed by `thread_id`	`SessionService` keyed by `app_name` / `user_id` / `session_id`
Mid-graph human pause	`interrupt()`	Generator node yielding `RequestInput(message=..., response_schema=...)` or `LongRunningFunctionTool`

- **ADK 2.0 orchestration positioning:** ADK 2.0 steers new stateful branching, cyclic retries, and human-input pauses toward `Workflow(edges=[...])` across Python, TypeScript, and Go, while keeping `SequentialAgent`, `ParallelAgent`, and `LoopAgent` as first-class primitives in Python and Java/Kotlin for fixed pipelines and exam blueprints.
- **Graph node exception & retry boundaries (`BaseException` vs. `RetryConfig`):** Catching broad `BaseException` (or bare `except:`) without re-raising traps `NodeInterruptedError` and breaks `RequestInput` HITL pauses, whereas broad `except Exception:` inside ADK 2.0 tools or nodes masks failures from `RetryConfig(max_attempts=...)` automatic retries. Let transient `Exception`s propagate to the framework, catching only narrow recoverable domain errors when intentionally returning a structured result.

Planners & October 2026 `thinking_level` Matrix: - **`BuiltInPlanner(thinking_config=types.ThinkingConfig(...))`:** Activates native Gemini System-2 reasoning (`part.thought == True` when `include_thoughts=True`) and preserves latent **Thought Signatures** across multi-turn tool loops. - **`PlanReActPlanner()`:** Prompts non-thinking or third-party models to emit explicit `/*PLANNING*/` → `/*ACTION*/` → `/*REASONING*/` → `/*FINAL_ANSWER*/` scaffolding tags. - **`thinking_level` (`low` | `medium` | `high` | `minimal`) vs. legacy `thinking_budget`:** `thinking_level` is **mandatory on Gemini 4 Argon** and **recommended across Gemini 3.x** (whereas integer `thinking_budget` is a legacy Gemini 2.5 parameter).

`thinking_level`	`gemini-4-argon`	`gemini-3.8-flash`	`gemini-3.5-flash-lite`	Architectural sweet spot
`"minimal"`	❌ HTTP 400 Error	❌ HTTP 400 Error	✅ **Supported (Default)**	Ultra-low-latency intent routers, schema extractors, inline classifiers
`"low"`	✅ Supported	✅ Supported	✅ Supported	Routine MCP tool execution, straightforward RAG synthesis, cost-sensitive turns
`"medium"`	✅ **Supported (Default)**	✅ **Supported (Default)**	✅ Supported	Balanced production default for multi-step agentic tool loops
`"high"`	✅ Supported (Dynamic)	✅ Supported (Dynamic)	✅ Supported (Dynamic)	Ambiguous planning, repository-scale code synthesis, conflict resolution (higher TTFT)

> **COMMON TRAP**
> Passing `thinking_level="minimal"` on **`gemini-4-argon`** or **`gemini-3.8-flash`**, OR passing legacy integer `thinking_budget=...` on **`gemini-4-argon`**, fails immediately with **HTTP `400 INVALID_ARGUMENT`**. Note also that `gemini-4-argon` ignores custom `temperature`, `top_p`, and `top_k` values—control reasoning depth and determinism via `thinking_level`, structured schemas, and explicit rubric instructions.

6. State, sessions and memory

Storage tier	Data shape	Lifecycle scope	ADK interface
Session events	Chronological immutable turn & tool transcript	Single conversation thread	`SessionService` (`session.events`)
Scoped state	Exact structured key/value scratchpad	Invocation, session, user, or app (by prefix)	`output_key`, `tool_context.state`, `EventActions(state_delta=...)`
Semantic memory	LLM-distilled, consolidated user facts retrieved by relevance	Cross-session per `user_id`	`MemoryService` (`VertexAiMemoryBankService`) + `load_memory` / `PreloadMemoryTool`
Artifacts	Versioned binary blobs (PDFs, images, CSVs; `"user:..."` = cross-session)	Session or user	`ArtifactService` (`GcsArtifactService` in prod)

State key prefix	Visibility scope	Survives process restart?
(unprefixed)	Current `session_id` only	Only with `DatabaseSessionService` (**Spanner**, **AlloyDB**, **Cloud SQL**) or `VertexAiSessionService`
`user:`	All sessions belonging to `user_id` under `app_name`	Only with `DatabaseSessionService` (**Spanner**, **AlloyDB**, **Cloud SQL**) or `VertexAiSessionService`
`app:`	Global across all users/sessions of `app_name`	Only with `DatabaseSessionService` (**Spanner**, **AlloyDB**, **Cloud SQL**) or `VertexAiSessionService`
`temp:`	Current single invocation (shared across sub-agents in that turn)	**Never persisted** (stripped prior to commit)

- **Memory Bank vs. RAG Memory vs. Enterprise Grounding:**
- **`VertexAiMemoryBankService` (Managed Memory Bank):** Runs LLM fact extraction and **contradiction consolidation** (`await memory_service.add_session_to_memory(session)` → `GenerateMemories`, typically triggered in `after_agent_callback`). If a user changes preferences in Session 5, Memory Bank revises the stored memory rather than returning stale facts (`add_memory` / `CreateMemory` writes verbatim entries unless `enable_consolidation=True`).
- **`VertexAiRagMemoryService`:** Indexes raw session chunks into a RAG corpus (`similarity_top_k`, `vector_distance_threshold` where **smaller = closer**); does **not** reconcile conflicting historical statements.
- **Shared enterprise grounding (not per-user memory):** Use `VertexAiSearchTool` (**Agent Search** with native SharePoint/Drive ACL propagation), `VertexAiRagRetrieval` (**RAG Engine**), or `google_search`.
- **`load_memory` vs. `PreloadMemoryTool`:** `load_memory` lets the model query memory on demand (cheaper, risk of skipped recall); `PreloadMemoryTool` injects relevant memories automatically at the start of every turn (consistent personalization at higher per-turn latency/tokens).

State Hygiene & Backing Store Quotas:

Sensitive or oversized payload	Why it must NEVER enter `session.state` or Memory Bank	Mandated Google Cloud store & ADK pattern
OAuth 3LO / 2LO tokens	Plaintext tokens leak into `session.events`, traces, and model prompts	**Auth Manager** token vault (`GcpAuthProvider` + regional `GcpAuthProviderScheme` on `locations/<REGION>/authProviders/...` emitting `adk_request_credential` with PKCE, `continue_uri`, and auto-generated anti-fixation `consent_nonce` verified on `:finalize`)
Static API keys, DB passwords & SA keys	Triggers Security Command Center secret findings and audit exposure	**Secret Manager** (or Auth Manager API Key provider) injected at the tool/proxy boundary

Sensitive or oversized payload	Why it must NEVER enter `session.state` or Memory Bank	Mandated Google Cloud store & ADK pattern
Unredacted PII / PHI / PCI	Persists across session history and distills into cross-session memories	**Sensitive Data Protection (SDP)** deterministic surrogates (`CryptoDeterministicConfig`) via **Model Armor** or `before_model_callback` / `after_tool_callback`
Multi-MB binary blobs (PDFs, images, CSVs)	Bloats JSON state payloads and hits document/gateway size limits	**`GcsArtifactService`** (`{app}/{user}/{session}/{filename}/{version}` or `"user:..."`); leaving `Runner(artifact_service=None)` raises **`ValueError`** on `save_artifact()`

```
jira_auth = GcpAuthProviderScheme(
    name="projects/my-sec-proj/locations/us-central1/authProviders/jira-3lo",  # Regional-only (never global)
    scopes=["read:jira-work"],  # Minimized read-only scope (never broad admin scopes)
    continue_uri="https://app.internal.example/oauth/callback",  # Handler posts consentNonce to :finalize
)
```

- **Firestore vs. Memorystore persistence boundaries:**
- **Firestore:** 99.999% multi-region strong consistency and document TTLs, with a hard **`1 MiB` max document size** and ~**`1` sustained write/sec per document** (persist turn events in **subcollections**, never by appending unbounded arrays inside a single parent document).
- **Memorystore for Redis / Valkey:** **`< 1 ms`** HNSW/FLAT semantic cache, sliding-window RPM/TPM rate limiters, and **Pub/Sub `messageId` idempotency keys** (scales to 250 nodes / 25+ TB), with mutually exclusive `RDB` snapshot or `AOF` (`appendfsync`) persistence and async replication (still an in-memory tier; never use as a sole compliance audit ledger).

> COMMON TRAP
> Mutating `session.state["user:plan"] = "pro"` directly on the object returned by `await session_service.get_session(...)` . That mutates a detached snapshot, skips `append_event()` , leaves zero audit trail, and drops the write in Spanner / AlloyDB / PostgreSQL / Agent Runtime. Always mutate state through `tool_context.state` , `callback_context.state` , `output_key` , or `Event Actions(state_delta={...})` .

7. Tools and structured output

Passing a callable in `tools=[lookup_invoice]` wraps it as a `FunctionTool` whose **docstring** defines when the LLM invokes it and whose **type annotations** compile into the JSON Schema. Poor docstrings directly cause tool-selection hallucinations.

Tool design rule	Architectural reason	Implementation pattern
Atomic single responsibility	Monolithic tools hide unobservable logic and bloat failure blast radius	Separate `crm_get_customer` from `crm_issue_refund`
Namespace prefixes	Prevents collisions when combining multiple MCP servers and toolsets	`jira_create_issue` , `bq_execute_sql`
Document constraints & units	Models plan accurately when limits are explicit in the schema description	`"Read-only SQL; max 500 rows; 15 s timeout; dates in ISO-8601."`
Return structured errors, don't raise	Uncaught exceptions abort the turn; error dicts let the LLM self-correct or escalate	`{"status": "error", "error_code": "NOT_FOUND", "retryable": False}`
Enforce security inside tool code	Prompts cannot act as security boundaries against injection	Canonical path `resolve().is_relative_to(ROOT)` + `WriteMode.BLOCKED`
Idempotent mutations	`ResumabilityConfig` and transport retries have at-least-once semantics	Require `idempotency_key` before charging cards or creating tickets
Cap active tools (~16 max)	Tool selection precision degrades sharply when schemas crowd context	Partition across specialist `AgentTool` s or progressive `SkillToolset` s

Enterprise integration layer	ADK class & transport	Target backend & primary strengths	Operational & auth constraint
Integration Connectors + Application Integration (`GA`)	`ApplicationIntegrationToolset`	**100+ SaaS & ERP systems** (SAP, Salesforce, ServiceNow, Workday) with schema mapping	Pair with **Auth Manager 3LO** for user-scoped mutations; higher hop latency/cost than direct DB pooling
MCP Toolbox for Databases (`GA`)	`ToolboxToolset` / `McpToolset` (`googleapis/genai-toolbox`)	**30+ relational, graph & NoSQL DBs** via parameterized `tools.yaml` SQL/Cypher queries	Sub-50 ms connection pooling, IAM auth, and OTel spans; no SaaS/ERP entity adapters
Apigee + API Hub (`GA`)	`APIHubToolset`	**Cataloged OpenAPI REST microservices** with rate/token quotas, mTLS, semantic caching, and Model Armor	Filter operations explicitly rather than importing hundreds of raw endpoints onto one agent
Custom Remote MCP on Cloud Run (`GA`)	`McpToolset` + `StreamableHTTPConnectionParams`	**Bespoke domain tools** deployed with `--functional-type=mcp-server` and `--no-allow-unauthenticated` behind IAP	Send Cloud Run OIDC in **`X-Serverless-Authorization`** when forwarding a user `Authorization` OAuth token, or delegate to **Agent Gateway + Auth Manager**

- **Built-in tool mixing rules across SDK languages:** Gemini server-side tools (`google_search` , `VertexAiSearchTool` , `BuiltInCodeExecutor`) execute inside the model API call.
- In **ADK Python (v1.16+)**, set `bypass_multi_tools_limit=True` on `GoogleSearchTool` or `VertexAiSearchTool` to mix them alongside custom `FunctionTool` s.
- Combining **`BuiltInCodeExecutor`** with custom tools (in Python or TypeScript), OR combining **`GoogleSearchTool` with custom tools in Java/Kotlin**, is rejected unless you isolate each built-in tool inside its own specialist `LlmAgent` wrapped in **`AgentTool(agent=...)`** .
- **Additional native ADK toolsets:** `BigQueryToolset` (`BigQueryToolConfig(write_mode=WriteMode.BLOCKED)`), `SpannerToolset` , `OpenAPIToolset` , and `SkillToolset` .
- **Structured output (`output_schema`):** Pass a Pydantic model (`output_schema=TriageVerdict`) with `output_key="verdict"` for schema-guaranteed JSON. When `output_schema` conflicts with multi-turn tool calling on a given model/SDK configuration, decouple them in a 2-step `SequentialAgent` / `Workflow` : Step 1 runs tools and writes raw findings to state; Step 2 runs a stateless formatter (`output_schema=TriageVerdict` , `include_contents="none"`).

8. MCP, A2A and Agent Registry integration

MCP is vertical (`Agent → Tools/Resources`); A2A is horizontal (`Agent → Peer Agent`). Enterprise architectures combine both under **Agent Registry** governance.

```
# Vertical: remote Streamable HTTP MCP server with explicit allowlist & namespace prefix
erp_mcp = McpToolset(
    connection_params=StreamableHTTPConnectionParams(url="https://erp-mcp.internal.example/mcp"),
    tool_filter=["get_order", "list_shipments"],  # Exact list or ReadonlyContext predicate (never globs)
    tool_name_prefix="erp",
)

# Horizontal: cross-org or cross-language specialist discovered via A2A Agent Card
customs_agent = RemoteA2aAgent(
    name="customs_clearance_agent",
    agent_card="https://partner.example/.well-known/agent-card.json",
)
```

Integration layer	Production rules & Google Cloud integration
MCP Transports & Lifecycle	Use `StdioConnectionParams` only for local dev subprocesses; use `StreamableHTTPConnectionParams` for remote stateless/stateful production services (legacy SSE-only transport is deprecated). Instantiate `McpToolset` **synchronously at module scope** in `agent.py` for `adk deploy`, and call `await toolset.close()` on custom FastAPI shutdown.
Google Cloud Managed MCP & Registry	Discover governed enterprise tools and agents through **Agent Registry** (`AgentRegistry`), first-party remote `StreamableHTTP` MCP servers auto-registered in `global` Registry (`bigquery.googleapis.com/mcp`, `container.googleapis.com/mcp`, `compute.googleapis.com/mcp`, `run.googleapis.com/mcp`, `logging.googleapis.com/mcp`, `developerknowledge.googleapis.com/mcp` via `private.googleapis.com` in VPC-SC, and Agent Retrieval MCP), and **MCP Toolbox for Databases** (`ToolboxToolset`). Authorize calls using **Agent Identity** (`principal://...`) and **Auth Manager** OAuth 2.0 tokens.
A2A Consumption & Publishing	Consume peers via `RemoteA2aAgent` (supports Go/Java/Python/LangGraph interoperability); expose an ADK agent via `to_a2a(agent, port=8001)` or `adk api_server --a2a`. Supports JSON-RPC 2.0 / gRPC / HTTP+JSON, SSE streaming (`message/stream`), and async webhook push notifications across task states (`submitted` → `working` → `input-required` / `auth-required` → `completed` / `failed` / `canceled` / `rejected`).
Adjacent Agentic Protocols	**A2UI:** declarative JSON UI payloads rendered only from the client's trusted component catalog. **AP2:** A2A payment extension enforcing cryptographically signed **Intent → Cart → Payment** mandates.

> **COMMON TRAP**
> Assuming local ADK `before_tool_callback` hooks inspect a `RemoteA2aAgent`'s internal database queries. Remote A2A agents are **opaque peers** executing outside your process—never expose internal prompts/tools over A2A, authenticate via the card's `securitySchemes`, and validate every returned A2A artifact at your boundary.

Platform path: expose, deploy, register and discover

Step	Action	Documented mechanism
1. Expose	Serve the ADK agent over A2A with an Agent Card	`to_a2a(root_agent, port=...)` builds the card from the agent's metadata and skills and publishes it at `/.well-known/agent-card.json` once served with `uvicorn`; pass `agent_card=` (an `AgentCard` object or a JSON file path) to control what is advertised. `adk api_server --a2a` exposes only agent folders that contain an `agent.json` card.
2. Deploy	Host on Agent Runtime with a per-agent identity	`agents-cli deploy --agent-identity`; or `adk deploy agent_engine` with `.agent_engine_config.json` set to `{"identity_type": "AGENT_IDENTITY"}`; or the Agent Platform SDK `client.runtimes.create(...)` with `identity_type` in the config. A2A-native hosting on Agent Runtime uses the `A2aAgent` template (`Agent Card` + `AgentExecutor`), currently Preview.
3. Register	Make the agent discoverable in Agent Registry	Automatic or manual registration (next table).
4. Discover	Resolve peers at runtime instead of hardcoding URLs	ADK `AgentRegistry` client (`google-adk[a2a,agent-identity,mcp]`, version `1.29.0` or later): `get_remote_a2a_agent()` returns a remote agent you can attach as a sub-agent; `get_mcp_toolset()` returns a registered MCP server as a toolset.
5. Authorize	Allow only approved caller-to-peer paths	Agent policies at Agent Gateway (see the next subsection).

Where the agent runs	Registration mode	How it happens	What gets indexed
Agent Runtime	automatic, on by default	deploying registers it; later updates and deletions sync	A2A skills read from the Agent Card when the agent speaks A2A
Cloud Run	automatic, opt-in	deploy with `--functional-type=agent`, which requires `--identity-type=agent-identity`	agent name and `AGENT` type, plus card skills for A2A agents
GKE	automatic, opt-in	label `registry.gke.io/functional-type: "AGENT"` plus an `a2a-protocol.org/agent-card` annotation on the Deployment	card skills from the annotated endpoint
Built-in Google agents (Google Workspace, Gemini Enterprise)	automatic	no action	ingested by the platform
External clouds, on-premises, unsupported runtimes, other projects outside a management boundary	manual	console, `gcloud agent-registry services create --agent-spec-type=a2a-agent-card --agent-spec-content=agent-card.json`, or Terraform `google_agent_registry_service`; needs `roles/agentregistry.editor` on the governance project	`A2A_AGENT_CARD` agents get their card synced; plain REST agents use `NO_SPEC` and carry no skills

- **Discovery scope:** automatic registration only sees the governance project unless you configure a management boundary across projects; everything else must be registered manually.
- **Non-A2A agents:** the registry records them but cannot capture endpoint, identity or skills on its own; add those through manual registration or skill registration.
- **Location limit:** manual registration is not available in the `us` and `eu` multi-regions; use a single region or `global`.

```
from google.adk.a2a.utils.agent_to_a2a import to_a2a

# Card is generated from the agent; port must match the uvicorn --port value
# or the card advertises an unreachable URL
a2a_app = to_a2a(root_agent, port=8001)
```

```
from google.adk.integrations.agent_registry import AgentRegistry

registry = AgentRegistry(project_id=GOVERNANCE_PROJECT, location="global")
# Resolve once at startup, not on every invocation
fx_agent = registry.get_remote_a2a_agent(
    agent_name="agents/AGENT_ID",       # short form inherits project and
location
    httpx_client=authed_httpx_client,   # attaches the caller's credentials
)
coordinator = LlmAgent(
    name="treasury_coordinator",
    model="gemini-3.8-flash",
    instruction="Send currency questions to the FX specialist.",
    sub_agents=[fx_agent],              # fx_agent.clone() for a second
parent
)
```

- **Startup resolution & `.clone()` across branches:** resolve `AgentRegistry.get_remote_a2a_agent()` and `get_mcp_toolset()` once at startup rather than on every turn; because every ADK agent instance obeys the single-parent rule, call `.clone()` whenever reusing a resolved remote A2A agent across multiple workflow branches or parent orchestrators.
- **A2A large binary payload handling:** pass Cloud Storage `gs://` URIs via `Gcs ArtifactService` rather than inline base64 A2A `Part` blobs, which bloat JSON-RPC payloads and choke Agent Gateway `CONTENT_AUTHZ` inspection.
- **Anti-pattern (hardcoded peer URLs):** pasting a partner's card URL into `RemoteA2aAgent` skips the registry, so endpoint changes break silently and the gateway cannot apply per-resource rules to that peer.

Cross-agent authorization with agent policies

Agent policies are the controls that Agent Gateway enforces on Agent-to-Anywhere (egress) traffic. Each control answers a different question, and a call proceeds only when all of them pass.

```
coordinator on Agent Runtime (principal A)
   | A2A request, mTLS + certificate-bound token
   ▼
Agent Gateway, Agent-to-Anywhere mode
   1. IAP: UAP grants A egress to the target? PAB includes the target?
   2. Agent Registry: target registered, or an explicit rule for its URL?
   3. Inspection: Model Armor, semantic governance, custom authz engine
   ▼
specialist on Agent Runtime (principal B, its own egress policies)
```

Control	Question it answers	How it is enforced	Exam detail & quotas
Agent Identity (GA)	Who is calling?	SPIFFE-based `principal://` identifier per agent, tied to the Agent Runtime resource; X.509 certificate and certificate-bound tokens (~24 h rotation) checked with mTLS and DPoP under a Google-managed Context-Aware Access policy	not shared across workloads by default, cannot be impersonated, no long-lived keys
Agent Registry check	Is the destination approved?	gateway confirms the target is registered (`destination.agent_registry.agent.name` / `.mcp_server.name`); an unregistered target needs an explicit access rule on `destination.unregistered.host` / `.path`	no matching policy means the request is denied (`roles/agentregistry.viewer` grants discovery only, not invocation)
IAM Unified Access Policies (via IAP, `REQUEST_AUTHZ`)	May principal A reach this agent, tool or endpoint?	allow/deny rules (`roles/iap.egressor` / `iap.googleapis.com/resources.egressViaIAP`) plus CEL keys `api.getAttribute('iap.googleapis.com/mcp.toolName', '')` , `api.getAttribute('iap.googleapis.com/mcp.tool.isReadOnly', false)` , and `destination.agent_registry.*`	**max `4` custom `Authz Policy` per egress gateway**; deny overrides allow; dry-run via `iam EnforcementMode: "DRY_RUN"` (enforce by removing field); match `global` vs. regional registry location to avoid `403 IAP_AUTHZ = DENIED`
Principal Access Boundary (GA)	What is the most A could ever reach?	PAB policy bound to the Agent Identity principal set (**max `500` rules per policy**, **`500` resources per rule**), evaluated before allow policies	acts strictly as an identity ceiling (never grants access alone); an over-granted allow rule still cannot reach resources outside the boundary
Model Armor (`CONTENT_AUTHZ`)	Is the payload safe?	inline `FULL_DUPLEX_STREAMED` content filter on prompts, tool responses, and the **first `256` URIs** per payload (consumes quota in the template's project)	dry-run via `INSPECT_ONLY` → enforce via `INSPECT_AND_BLOCK` ; org/folder Floor Settings enforce a minimum floor that project templates cannot weaken
Semantic governance policies (Preview, `CONTENT_AUTHZ`)	Does this tool call match the user's intent and business rules?	intent-aware LLM policy firewall evaluating chat history, tool calls, and `load_skill` actions against Natural Language Constraints (**max `5,000` chars per NLC**; **max `1,000` chars per tool description**)	**strict deny-overrides** (`any DENY` blocks); dry-run via `sgpEnforcementMode: "DRY_RUN"` (enforce by removing flag); scope CEL predicates away from raw gRPC health traffic; supplements IAM, never replaces it

- **Availability:** the Google-managed gateway covers Agent Runtime and Gemini Enterprise; GKE, Cloud Run and cross-cloud deployment models are listed as early preview.
- **Anti-pattern (enforcing on day one):** enforcing a new egress policy without a dry run can cut off working handoffs in production; validate in `DRY_RUN` against audit logs first.
- **Anti-pattern (semantic governance as the only gate):** relying on natural-language constraints for access control leaves a probabilistic check as the sole barrier; keep deterministic UAP deny rules and PAB underneath it.

Handoff controls and approval pauses by pattern

The orchestration-boundary table in section 4 picks the boundary. This table shows what each choice gives you at the handoff itself: whose identity the next agent uses, whether the registry knows about it, what can sit between the two agents and where a person can stop the flow.

Pattern	Where it runs	Sub-agent identity	Agent Registry discovery	Controls between agents	Where a human approval pauses it
Sequential (`SequentialAgent`)	inside the parent's process, children one after another in the same invocation	the deployment's single principal, shared by every child	children are not registered; only the deployed agent as a whole is	`output_key` and `{key}` templates carry data; `before_agent_callback` can gate the next child	ADK tool confirmation inside a child's tool (`require_confirmation` or `tool_context.request_confirmation(hint=..., payload=...)`)
Parallel (`ParallelAgent`)	same process; branches start concurrently and see no history or state from sibling branches while running	same single principal	same as sequential	one distinct `output_key` per branch, then a merge step that reads them after all branches finish	same tool-level confirmation, raised inside a branch's tool
Graph workflow (`Workflow(edges=[...])`)	same process (Python, TypeScript, Go)	same single principal	nodes are not registered	typed `node_input` , route keys on edges, and plain function nodes that validate or redact before the next agent node	a node yields `RequestInput(message=..., payload=..., response_schema=...)` (from `google.adk.events`); the reply becomes the next node's input, and `rerun_on_resume` decides whether the pausing node runs again

Pattern	Where it runs	Sub-agent identity	Agent Registry discovery	Controls between agents	Where a human approval pauses it
Remote A2A (`RemoteA2aAgent` to a peer on Agent Runtime)	separate service, possibly another project or organization	the peer's own Agent Identity principal	`AgentRegistry.get_remote_a2a_agent()` ; Agent Runtime peers register automatically, others manually from their card	Agent Gateway policies (registry check, UAP, PAB, Model Armor, semantic governance) plus artifact validation at your boundary	the peer returns its task in `TASK_STATE_INPUT_REQUIRED` (v0.3 JSON: `input-required`), an interrupted state that is not terminal; the caller answers with another message on the same task

- **Tool confirmation limit:** ADK documents that tool confirmation does not support `DatabaseSessionService` or `VertexAiSessionService` ; for durable approvals use `LongRunningFunctionTool` or a graph `RequestInput` node (section 11).
- **Confirmation with resume:** when resumability is on, the confirmation response must carry the same `invocation_id` that raised the request, otherwise ADK starts a new invocation.
- **Confirmation vs. RequestInput:** tool confirmation lives inside an `LlmAgent` 's tool call and asks for a yes/no answer or a small payload; `RequestInput` is a standalone graph node that can ask for structured data against a schema.
- **Detective control after the handoff:** Agent Anomaly Detection (Preview, allowlisted access) reads OpenTelemetry logs and traces asynchronously, so it adds no latency and never blocks a call. Its categories include agentic cascading failures (OWASP ASI08: fault propagation, infinite loops, oscillating retries), tool misuse (ASI02), identity privilege abuse (ASI03) and rogue agents (ASI10), with findings sent to Security Command Center.
- **Anti-pattern (anomaly detection as the loop breaker):** an asynchronous detector reports a ping-pong handoff after it happened; the hard stops remain `RunConfig(max_llm_calls=...)` , `LoopAgent(max_iterations=...)` and hub-and-spoke transfer rules.

9. Skills, Skill Registry and Agents CLI

A **tool** executes an action; a **skill** packages domain procedural knowledge (`SKILL.md` instructions + reference docs + helper scripts) using **3-tier progressive disclosure** so a 50-skill agent doesn't burn 80,000 prompt tokens on turn 1.

Disclosure tier	Payload loaded into context	Triggered by ADK tool
L1: Routing metadata	YAML frontmatter `name` (kebab-case matching folder) + trigger-phrased `description` (~60 tokens)	`list_skills` / `search_skills` (or eager catalog summary)
L2: Playbook body	Complete `SKILL.md` workflow instructions (< 500 lines)	`load_skill`
L3: Deep assets & scripts	`references/` , `assets/` , and executable `scripts/`	`load_skill_resource` , `run_skill_script` (requires sandboxed `code_executor`)

- **Wiring local skills & `GCPSkillRegistry` :** Load local folders via `load_skill_from_dir(Path("./skills/incident-triage"))` into `SkillToolset(skills=[...], code_executor=AgentEngineSandboxCodeExecutor())` , or attach `GCPSkillRegistry` (**Agent Platform Skill Registry**) so agents dynamically discover versioned, organization-approved skills via `search_skills` without redeployment.
- **Configuring Skills & Plugins with `Agents CLI` :**
- **Skills & plugins management:** Install reusable skills (`npx skills add google/agents-cli` or `agents-cli` skill/plugin commands) and scaffold telemetry, Terraform, CI/CD, and security plugins across ADK workspaces.
- **Human mode vs. Agent mode:** In **human mode** (interactive TTY), `agents-cli` renders rich interactive wizards, colorized diffs, and confirmation prompts for developers. When invoked by autonomous coding harnesses (**Google Antigravity**, **Claude Code**, or CI runners), run `agents-cli` in **agent / non-interactive mode** (explicit CLI flags, no interactive stdin prompts, deterministic exit codes, and machine-readable JSON output) so automated workflows never hang waiting on terminal input. Same rule applies to L3 `SKILL.md` scripts: **non-interactive flags/env vars only**, `--help` support, and structured JSON on `stdout` .

COMMON TRAP

Treating an unreviewed third-party `SKILL.md` as harmless documentation. Skill instructions enter the model's privileged context and `run_skill_script` executes code—sync skills only from governed **Skill Registry** or code-reviewed repos, and never attach `UnsafeLocalCodeExecutor` to `SkillToolset` .

10. Callbacks, plugins and guardrails

Deterministic harness principle: Never rely on system-prompt wording (`"Please redact SSNs and never refund over $500"`) for security, compliance, or budget caps. Enforce hard invariants in deterministic ADK callbacks and `BasePlugin` s surrounding the model.

Lifecycle hook	Return `None`	Return a typed override object
`before_agent_callback(callback_context)`	Agent executes normally	`types.Content` → **skips agent run**; returns `Content` as the reply
`after_agent_callback(callback_context)`	Final output kept as-is	`types.Content` → **appends** an additional event (does **not** overwrite reply)
`before_model_callback(callback_context, llm_request)`	Calls LLM (`llm_request` mutable in place)	`LlmResponse` → **skips LLM call** (cache hit, fast policy block)
`after_model_callback(callback_context, llm_response)`	Keeps LLM output	`LlmResponse` → **replaces** model response (final output redaction/disclaimer)
`on_model_error_callback(callback_context, llm_request, error)`	Re-raises exception	`LlmResponse` → **suppresses error** and substitutes fallback (`after_model_callback` still runs)
`before_tool_callback(tool, args, tool_context)`	Executes tool (`args` mutable in place)	`dict` → **skips tool execution**; returns `dict` to the model as the tool result
`after_tool_callback(tool, args, tool_context, tool_response)`	Keeps raw tool result	`dict` → **replaces tool result** before it enters `session.events` or LLM context
`on_tool_error_callback(tool, args, tool_context, error)`	Re-raises tool exception	`dict` → **suppresses exception** and returns recovery `dict` (`after_tool_callback` still runs)

- **App-wide `BasePlugin` precedence:** Register cross-cutting guardrails (**Model Armor** screening, **Sensitive Data Protection** PII redaction, token budgets, audit logging) once on `App(name="...", root_agent=..., plugins=[...])` . **Plugin callbacks always execute before agent-level callbacks**, and the first hook returning a non- `None` override short-circuits remaining hooks in that phase.
- **Defense-in-depth ordering:** Screen user prompts in `before_model_callback` (fast regex → lightweight classifier / **Model Armor**) → validate tool parameters & spending caps in `before_tool_callback` → strip indirect prompt injections and **redact PII in `after_tool_callback`** → keep `after_model_callback` as the final outbound safety net.

COMMON TRAP

Scrubbing PII exclusively in `after_model_callback` . By that stage, unredacted customer records returned by a tool have already been committed to `session.events` , exported to traces (if content capture is on), and transmitted to the model context. Always scrub at **`after_tool_callback`** and **`before_tool_callback`** first.

11. Human-in-the-loop, resilience and durability

Operational requirement	ADK implementation pattern
Conditional approval on sensitive tools	`FunctionTool(fn, require_confirmation=lambda amount, **_: amount > 500)` (note: built-in `require_confirmation` is **`InMemorySessionService`-only**)
Production durable HITL across DB / Agent Runtime sessions	**`LongRunningFunctionTool`** (returns pending ticket/status immediately; external approval webhook resumes with matching `function_call.id`) or ADK 2.0 `Workflow` generator yielding `RequestInput(message=..., response_schema=...)`
Crash-resilient invocation replay	`App(..., resumability_config=ResumabilityConfig(is_resumable=True))` resumed with `invocation_id` (**at-least-once** execution—tools must be idempotent!)
Runaway reasoning loop protection	`RunConfig(max_llm_calls=...)` (default **500**), `LoopAgent(max_iterations=...)`, plus per-session failure counters in `state`

3-Layer Fault Recovery Ladder: 1. **Transport layer (HTTP `429` / `503`):** Configure `HttpRetryOptions(attempts=3, initial_delay=1.0, exp_base=2.0)` on the `Gemini` model wrapper (and route stateless calls to the `global` endpoint or **Priority PayGo / Provisioned Throughput**). 2. **Tool argument self-correction:** Attach `ReflectAndRetryToolPlugin(max_retries=3, throw_exception_if_retry_exceeded=False)`. **Critical nuance:** out of the box it hooks raised exceptions (`on_tool_error_callback`); when your tools follow best practice and return `{"status": "error", ...}` dicts instead of raising, **override `extract_error_from_result`** on the plugin so structured error dicts trigger reflection. 3. **Model failover & graceful escalation:** Catch regional/model outages in `on_model_error_callback` (Python) or `RoutedLlm` (TypeScript) to fail over (e.g., `gemini-4-argon` → `gemini-3.8-flash`), and escalate deterministically when consecutive tool retries exhaust budget.

12. Context engineering, Gemini 4 Argon model routing and caching

Stuffing raw multi-turn transcripts and 50 KB JSON payloads into every specialist degrades accuracy ("lost in the middle"), inflates p95 latency, and multiplies token spend ~15× across multi-agent trees.

3-Tier Agentic Model Routing:

Routing tier	Recommended model & `thinking_level`	Workload & architectural role
Tier 1: High-QPS Router & Guardrail	**`gemini-3.5-flash-lite`** (`thinking_level="minimal"`) or fine-tuned **Gemma 4** SLM	Sub-200ms intent classification, slot extraction, schema normalization, and pre-LLM guardrail triage
Tier 2: Primary Agentic Workhorse	**`gemini-3.8-flash`** (`thinking_level="low"` or `"medium"`)	Default execution loop for MCP tool calling, code generation, RAG synthesis, and `AgentTool` specialists
Tier 3: Frontier Planner & Deep Synthesis	**`gemini-4-argon`** (`thinking_level="high"` or `"medium"`)	Ambiguous multi-step task decomposition, cross-document legal/financial conflict resolution, repo-scale refactoring, and unified 1M-token decodes
Real-Time Voice / Video Front-Door	**`gemini-3.8-live`** (`runner.run_live`) + **`gemini-4-argon`** (`AgentTool`)	`gemini-4-argon` does not support Live API WebSockets; handle sub-second voice + barge-in on `3.8-live` and delegate heavy backend reasoning via `AgentTool`

Context Compaction & Prefix Caching Mechanics (Gemini 4 Argon vs. Gemini 3.x):

Mechanism	Configured on	How it works & version-specific rules
Implicit Context Caching (Gemini 4 Argon & 3.x)	Automatic (model API)	**`gemini-4-argon` uses implicit caching exclusively** (triggers automatically on shared prefixes `≥ 4,096` tokens with zero hourly storage fee; **explicit `cachedContents` creation fails on `gemini-4-argon`**). Prefix stability via `static_instruction` is essential!
`ContextCacheConfig(cache_intervals, ttl_seconds, min_tokens)`	`App`	Manages explicit TTL `cachedContents` handles on **Gemini 3.x / 2.5** base models when deterministic cache retention across sparse traffic justifies storage fees
`EventsCompactionConfig`	`App`	Summarizes older `session.events` slices (`token_threshold=12000, event_retention_size=6` or `compaction_interval, overlap_size`) into compact summary events so long-running sessions avoid context degradation
`include_contents='none'` + `{state_key}`	`LlmAgent`	Strips prior chat history completely for stateless extractors, formatters, and independent auditors
Observation masking (`after_tool_callback`)	`LlmAgent` / `BasePlugin`	Trims verbose 40-column API responses down to the 4 needed fields before the event enters `session.events`

```
app = App(
    name="support_hub",
    root_agent=root_agent,
    events_compaction_config=EventsCompactionConfig(token_threshold=12000,
event_retention_size=6),
)
```

COMMON TRAP (PREFIX-CACHE INVALIDATION)
Trying to provision explicit `cachedContents` (`ContextCacheConfig`) for **`gemini-4-argon`**, or injecting `{current_timestamp}` or `{user_id}` at line 1 of `instruction`. `gemini-4-argon` rejects explicit `cachedContents` and relies on **automatic implicit caching (`≥ 4,096` tokens)**, while even on Gemini 3.x `ContextCacheConfig`, mutating the leading prompt prefix invalidates every cached token block (0% cache hit rate)!

13. Security and sandboxing

Threat vector	Attack scenario	Deterministic ADK & platform control
Direct prompt injection	User attempts to override system instructions	`before_model_callback` + **Vertex AI Model Armor** prompt screening
Indirect prompt injection & skill poisoning	Malicious instructions hidden inside fetched web pages, PDFs, or dynamic `SKILL.md` loads	Sanitize untrusted payloads in `after_tool_callback`, isolate untrusted reader subagents (`include_contents="none"`, zero write tools), and enforce **Skill Governance (`govern-agent-skills`, Preview)** on `load_skill` at **Agent Gateway (`CONTENT_AUTHZ`)**
Confused deputy / excessive agency	Read-only support bot tricked into invoking `delete_account`	Least-privilege `tool_filter` allowlist, `WriteMode.BLOCKED`, and per-agent **Agent Identity**
Cross-agent taint propagation	Compromised sub-agent poisons downstream publisher	Enforce strict Pydantic `output_schema` + boundary validation callbacks between handoffs
Host container escape via generated code	LLM-generated Python reads `/proc/self/environ` or metadata credentials	Never use `UnsafeLocalCodeExecutor` in prod; isolate in **Agent Runtime Code Execution** or **GKE Agent Sandbox (gVisor)**

`code_executor` implementation	Isolation boundary & statefulness	Recommended environment
`BuiltInCodeExecutor`	Gemini server-side sandbox (stateless, 30 s max, zero network egress; cannot mix directly with custom tools on one agent)	Zero-infra math, data wrangling, charts

`code_executor` implementation	Isolation boundary & statefulness	Recommended environment
`AgentEngineSandboxCodeExecutor`	Managed **Agent Runtime** isolated container (stateful files across turns, up to 14-day TTL, 100 MB I/O)	Default production choice on Google Cloud
`GkeCodeExecutor(executor_type="sandbox")`	Pre-warmed **GKE Agent Sandbox** backed by **gVisor** kernel isolation (sub-second startup; replaces slow per-run K8s `"job"` mode)	GKE clusters needing custom VPC/hardware control
`ContainerCodeExecutor`	Dedicated Docker container	Local staging or custom system binary testing
`UnsafeLocalCodeExecutor`	Runs `exec()` inside the ADK host process with full IAM/env access	Throwaway local debugging only, **never** staging or prod

Three identity authorities: (1) Google Cloud Agent Identity (`principal://...` / `spiffe://...` per-agent cryptographic identity bound to **Principal Access Boundary [PAB]** policies and **Dual-Attribution Cloud Audit Logs**), **(2) Workload Service Account** (coarse infrastructure identity), and **(3) User OAuth 2.0 Delegation** (`ToolContext.request_credential` / **Auth Manager** `GcpAuthProviderScheme` when acting strictly under the end user's own entitlements).

14. Evaluation: ADK `evalset` vs. Gen AI Evaluation Service & observability

Core principle: Grade **both** the final response and the **tool trajectory**. An agent that guesses the right refund figure while skipping mandatory `verify_return_window` compliance checks is a production defect.

Architectural Decision Matrix: Which Evaluation Engine Should You Use?

Evaluation tier	Primary tool / API	Executes where & against what?	Key strengths & when to choose
1. Deterministic Harness Replay	`adk conformance record` + `adk conformance test --mode=replay`	Local / PR CI in `<5s` using recorded LLM responses (zero live model calls)	Verifies Python/Go/TS callback wiring, `state_delta` routing, and guardrail refactors with **zero token cost and zero sampling variance** on every commit
2. Local & CI Golden Regression	`adk eval <dir> <evalset.json>` / `pytest` `AgentEvaluator.evaluate(...)` / `adk web` Eval tab	In-repo CI runner executing your ADK agent against `*.test.json` (unit) and `*.evalset.json` (multi-turn)	Fast developer inner-loop and PR merge gate grading `tool_trajectory_avg_score` (`EXACT` , `IN_ORDER` , `ANY_ORDER`), `final_response_match_v2` , `hallucinations_v1` , `UserSimulatorProvider` , and Python `custom_metrics` (`CodeExecutionMetric`)
3. Managed Cloud Fleet Evaluation & Simulation	**Agent Platform Gen AI Evaluation Service** (`client.evals` / `EvalTask` / `agents-cli eval compare --fail-on-regression`)	Managed Google Cloud service grading **any** agent (ADK on Agent Runtime, Go/TS/Java Cloud Run endpoints, LangGraph, or custom callables)	**Choose when you need: Adaptive Rubrics,** `GROUNDING` vs. `HALLUCINATION` vs. `PairwiseMetric` (A/B order swap), **2-Step Agent Simulation** (from immutable **Agent Registry** snapshots), **Environment Simulation** (injecting HTTP 503/429/timeout tool faults safely), **Online Evaluation Monitors** (`evaluate-online`), **Auto-Loss Analysis** (`client.evals.generate_loss_clusters()`), or **`adk optimize`** (`optimize-agent`)

ADK `test_config.json` criterion	What it grades	Rater type & exam nuances
`tool_trajectory_avg_score`	Predicted vs. reference tool sequence (`match_type` : `EXACT` default, `IN_ORDER` allows extra helper calls between required steps, `ANY_ORDER` for parallel lookups)	Deterministic (mirrors cloud `trajectory_exact_match` / `trajectory_in_order_match` / `trajectory_any_order_match` / `trajectory_precision` / `trajectory_recall`)
`response_match_score` vs. `final_response_match_v2`	Lexical ROUGE-1 overlap vs. semantic equivalence	Deterministic vs. LLM judge (prefer `final_response_match_v2` whenever valid phrasing varies)
`rubric_based_final_response_quality_v1` / `rubric_based_tool_use_quality_v1`	Domain-specific quality rubrics	LLM judge
`hallucinations_v1` & `safety_v1`	Context faithfulness (catches invented numbers despite calling the right tool; requires tool-output spans in context to avoid false positives) & harmlessness	LLM judge (`safety_v1` / `safety_v3` delegates to Gen AI Eval Service)
`multi_turn_task_success_v1` / `multi_turn_trajectory_quality_v1` / `per_turn_user_simulator_quality_v1`	End-to-end goal completion and red-teaming with `UserSimulator` (`UserPersona` , `starting_prompt` , `conversation_scenario` , and adversarial **`violation_rubrics`**) plus simulator persona-fidelity meta-scoring (`per_turn_user_simulator_quality_v1`)	LLM judge
`tool_call_count_v1` , `token_usage_v1` , `invocation_duration_v1`	Resource efficiency telemetry	**`INFORMATIONAL`** **only**: setting thresholds on these in `test_config.json` throws an error; enforce hard call/cost gates via Python `custom_metrics` (`code_config.name` / `CodeExecutionMetric`)

Observability, Loss Clustering & `adk optimize` : - **OpenTelemetry GenAI spans:** ADK emits hierarchical spans (`invoke_agent` → `call_llm` → `execute_tool`). Export via `--otel_to_cloud` locally or `GOOGLE_CLOUD_AGENT_ENGINE_ENABLE_TELEMETRY=true` on Agent Runtime (powers the 8 **Agent Registry Observability** tabs: Overview, Evaluation, Models, Tools, Usage, Logs, Traces, Topology). - **Privacy-by-default:** Keep `ADK_CAPTURE_MESSAGE_CONTENT_IN_SPANS=false` (and `OTEL_INSTRUMENTATION_GENAI_CAPTURE_MESSAGE_CONTENT` unset/false) in production unless paired with **Sensitive Data Protection** de-identification and restricted log buckets. - **Closed-loop flywheel & `adk optimize` (`optimize-agent`):** Export low-scoring **Online Evaluation Monitor** traces and human escalations → cluster root causes via **`client.evals.generate_loss_clusters()`** → de-identify PII with SDP (`ReplaceWithInfoTypeConfig` / `CryptoDeterministicConfig`) → promote into **Example Store** and permanent `*.evalset.json` suites → run **`adk optimize`** (GEPA data-driven prompt optimizer) with a strict **`train`** **vs. held-out** **`validation`** **split** so mutated instructions generalize without overfitting.

> COMMON TRAP
> Testing agent resilience against rate limits or ERP outages by firing tests against live staging/production APIs, or running `adk optimize` on a single unsplit test file (overfitting prompt edits to test questions). Always use **Environment Simulation** + **2-Step Agent Simulation** against an immutable **Agent Registry** snapshot and hold out a separate `validation` split during prompt optimization.

15. Deployment, live streaming and developer tooling

Deployment target	CLI workflow	Infrastructure you manage	Choose when
Agent Runtime (managed `reasoningEngines`)	`adk deploy agent_engine` or `agents-cli deploy`	Agent code & dependencies only (**Python** payload omits API server; **Go** payload bundles its dedicated ADK API server)	**Default on Google Cloud:** turnkey `VertexAiSessionService` , **Memory Bank**, **Code Execution** sandbox, **Agent Identity**, and **Gemini Enterprise** registration

Deployment target	CLI workflow	Infrastructure you manage	Choose when
Cloud Run	`adk deploy cloud_run [--trigger_sources="pubsub,eventarc"]`	Container image, session DB (`DatabaseSessionService` / `agentengine://`), scaling	**TypeScript / Java / Kotlin** ADK containers, custom FastAPI/HTTP gateways, scale-to-zero, or **ambient event-driven agents** triggered by Pub/Sub & Eventarc
GKE	`adk deploy gke` / custom Helm	Cluster, node pools, networking	Strict VPC/data-residency isolation, co-located self-hosted **Gemma 4** GPUs/TPUs, or **gVisor Agent Sandbox** (`GkeCodeExecutor`)

Event-driven or async trigger	Delivery & timeout limits	Reliability & rate-control pattern	Best workload fit
Pub/Sub Push / Eventarc → Cloud Run or Agent Runtime	At-least-once delivery; **`600 s` (10 min) max ack deadline**	Enforce an atomic **`messageId` idempotency check** in **Memorystore** or **Firestore** before mutative tools, attach a **Dead-Letter Topic** (**`maxDeliveryAttempts ≥ 5`**), and tune `container_concurrency` for async I/O	1-to-N ambient event fan-out (GCS uploads, Cloud Audit events, CRM webhooks)
Cloud Tasks → Cloud Run / HTTP endpoint	Point-to-point dispatch; **`30 min`** HTTP timeout	Throttle bursty workloads via **`maxDispatchesPerSecond`** and `scheduleTime` backoff (no 1-to-N topic fan-out)	Protecting fragile downstream ERPs and Gemini RPM/TPM quotas from spiky agent bursts
Cloud Run Jobs	Headless batch execution; up to **`24 h`** max duration & `10,000` parallel tasks	Shard batch input in Cloud Storage / BigQuery with task-index checkpoints; zero interactive HTTP/SSE surface	Nightly corpus audits, bulk document triage, and offline batch agent evaluations

PLATFORM DEEP DIVE

See **Agent Platform: Build, Retrieve & Run** for capacity planning, regional compute exhaustion vs. project quotas, Provisioned Throughput GSUs, and Agent Registry governance.

```
runner = Runner(agent=agent, app_name="hub",
session_service=session_service)
msg = types.Content(role="user", parts=[types.Part(text=prompt)])

# Drain runner stream; filter out intermediate thought summaries (p.thought
== True)
async for event in runner.run_async(user_id="u_42", session_id=session.id,
new_message=msg):
    if event.is_final_response() and event.content and event.content.parts:
        final_answer = "".join(
            p.text or "" for p in event.content.parts if not getattr(p,
"thought", False)
        )
```

- **Real-time bidirectional voice & video (`runner.run_live`):** Open low-latency **Gemini 3.8 Live (`gemini-3.8-live`)** sessions via `runner.run_live(user_id=..., session_id=..., live_request_queue=LiveRequestQueue(), run_config=RunConfig(response_modalities=["AUDIO"]))` (`send_realtime(Blob)` for PCM audio/video frames, halt playback immediately when `event.interrupted == True` on user barge-in). Remember: setting `StreamingMode.BIDI` on `runner.run_async()` is a **silent no-op** in Python ADK (`run_async` only supports `StreamingMode.SSE`), and **`gemini-4-argon` does not support Live API WebSockets**; pair `gemini-3.8-live` at the voice front-door with a `gemini-4-argon` `AgentTool` for deep backend reasoning.

16. Failure modes and exam traps

Symptom or production failure	Root cause → architectural fix	Tempting distractor (wrong)
`HTTP 400 INVALID_ARGUMENT` when configuring thinking on `gemini-4-argon` or `gemini-3.8-flash`	Passed `thinking_level="minimal"` (unsupported on `4-argon` & `3.8-flash`) or legacy integer `thinking_budget` on `4-argon` → use `thinking_level` in `{"low", "medium", "high"}` (reserve `"minimal"` for `gemini-3.5-flash-lite`)	Setting `temperature=0.0` to disable thinking
Explicit `cachedContents` creation fails when migrating an ADK app to `gemini-4-argon`	`gemini-4-argon` supports **implicit caching only** (`≥ 4,096` tokens, zero storage fee) → remove explicit cache creation and place static prompt content in `static_instruction`	Increasing `ContextCacheConfig.ttl_seconds`
`ManagedAgent` raises `NotImplementedError` or regional connection error	Attached local Python `FunctionTool` / `McpToolset` or targeted `us-central1` → `ManagedAgent` supports **server-side tools only** and requires the **`global`** location on Agent Platform; keep local/MCP tools on a companion `LlmAgent`	Wrapping `McpToolset` in `StdioConnectionParams`
Chaining `previous_interaction_id` on `Interactions API` (`background=True`) returns `400 Bad Request`	Previous background interaction is still `in_progress` → poll interaction status until `completed` before sending the follow-up turn	Resending the full chat history without `previous_interaction_id`
Java/Kotlin ADK team tries to import ADK 2.0 `Workflow(edges=[...])` or mix `GoogleSearchTool` + custom tools on one agent	ADK 2.0 graph `Workflow` is currently **Python, TypeScript, and Go**; Java/Kotlin use ADK 1.x workflow agents (`SequentialAgent` / `ParallelAgent` / `LoopAgent`) and must isolate `GoogleSearchTool` inside an `AgentTool` specialist	Enabling `bypass_multi_tools_limit=True` in Java
Coding agent hangs indefinitely running `agents-cli` or a `SKILL.md` helper script in CI	Executed in interactive human TTY mode waiting on stdin confirmation → invoke `agents-cli` and skill scripts in **non-interactive agent mode** with explicit CLI flags and JSON stdout	Extending the Cloud Build timeout to 60 minutes
Cross-user data leak under load	State stored on `self` inside a singleton `Agent` → move all mutable state to `SessionService` (`app_name` / `user_id` / `session_id`)	Instantiating a new MCP server per HTTP request
`LoopAgent(max_iterations=3)` finishes without converging, yet downstream `publisher` still runs	`LoopAgent` exits **silently** at `max_iterations` → set `state["temp:approved"] = True` in the exit tool alongside `escalate = True` and gate `publisher` via `before_agent_callback`	Catching `TimeoutError` around `LoopAgent`
Specialist hijacks user conversation instead of returning a structured sub-result	Placed specialist in `sub_agents` (`mode="chat"` transfer) → wrap in `AgentTool(agent=...)` or use ADK 2.0 `mode="single_turn"`	Prompting `"Do not talk to the user"`
`KeyError` before LLM call when instruction contains `{"id": 1}`	ADK parses `{id}` as a `session.state` placeholder → pass an `InstructionProvider` callable `def prompt(ctx: ReadonlyContext) -> str:`	Escaping braces as `{{id}}`
Early `return` on `event.is_final_response()` returns step 1's draft or raw thoughts	Every sub-agent in a pipeline emits `is_final_response()` → drain the full `run_async` generator and filter `not getattr(p, "thought", False)`	Switching from `run_async` to `run_live`
`ReflectAndRetryToolPlugin` ignores failing tools	Tools return `{"status": "error"}` dicts instead of raising exceptions → override `extract_error_from_result` on the plugin	Raising `max_retries`

Symptom or production failure	Root cause → architectural fix	Tempting distractor (wrong)
Built-in `require_confirmation=True` breaks after switching to `DatabaseSessionService` / `VertexAiSessionService`	Built-in tool confirmation is `InMemorySessionService`-only → use `LongRunningFunctionTool` or ADK 2.0 `RequestInput`	Switching SQLAlchemy drivers
Raw SSNs appear in `session.events` and traces despite `after_model_callback` redaction	Tool results are persisted and sent to the LLM *before* `after_model_callback` → scrub PII at the tool boundary (`after_tool_callback` & `before_tool_callback`) and keep `ADK_CAPTURE_MESSAGE_CONTENT_IN_SPANS=false`	Strengthening system prompt privacy rules
`adk eval` fails when setting a threshold on `tool_call_count_v1` or `token_usage_v1`	Built-in efficiency metrics are `INFORMATIONAL` only → enforce call/token budgets via Python `custom_metrics` (`code_config.name`)	Lowering threshold values to `0.5`
Team wants to test 503 tool outages and auto-generate multi-turn personas across a deployed Go Cloud Run agent	`adk eval` is designed for local/CI golden files → use **Agent Platform Gen AI Evaluation Service (2-Step Agent Simulation + Environment Simulation + Adaptive Rubrics)**	Running chaos faults against live production ERP APIs
Audit logs show one principal for every specialist, and a research sub-agent can reach the refund MCP server	Shared identity across sub-agents: every in-process `sub_agents` child runs under its deployment's single Agent Identity, so policy cannot separate them → deploy the privileged specialist as its own Agent Runtime agent with its own identity, call it over A2A, and scope UAP allow rules to that principal	Adding "only the billing agent may issue refunds" to the coordinator's instruction
Coordinator's A2A call to a partner agent is denied at Agent Gateway even though the URL responds	Unregistered remote agent: the destination is not in Agent Registry and no access rule targets its URL, so the gateway denies by default → register it manually from its Agent Card, grant `iap.googleapis.com/resources.egressViaIAP` through a UAP, and resolve it with `get_remote_a2a_agent()`	Exporting a service account key so the coordinator can call the partner directly
Two agents pass the same task back and forth until the LLM call budget runs out	Delegation loop: peers can hand work to each other with no termination rule → enforce hub-and-spoke (`disallow_transfer_to_peers=True`, specialists as `AgentTool` or `mode="single_turn"` returning to the coordinator), keep `RunConfig(max_llm_calls=...)` as a backstop, and add a UAP deny rule so the specialist's identity cannot call back into the coordinator	Raising `max_llm_calls`
Handoffs to a partner agent start failing with connection or schema errors after the partner redeploys, while the coordinator's config has not changed	Stale Agent Card or endpoint drift: the peer URL or card is pinned in code, or a manual registry entry was never updated, so the coordinator works from an old snapshot → resolve the peer with `get_remote_a2a_agent()` (Agent Runtime registrations sync on update), update manual registrations from the new card in the partner's release pipeline, re-resolve on connection or schema errors, and compare the card's `version` and `protocol_version`	Raising `HttpRetryOptions(attempts=...)`

Symptom or production failure	Root cause → architectural fix	Tempting distractor (wrong)
Latency and token cost climb with every handoff, and specialists answer about topics from earlier in the conversation	Context bloat: `sub_agents` transfer shares the full `session.events`, so each specialist reads the whole transcript → call specialists through `AgentTool` (isolated child session) or set `include_contents="none"` and pass explicit state keys, enable `EventsCompactionConfig(compaction_interval=..., overlap_size=...)` on the `App` to summarize older events, and send a short task message over A2A instead of the transcript	Switching to a model with a larger context window
One of several parallel researchers errors or hangs, and the merge step returns an incomplete answer or the whole turn fails	Partial branch failure in parallel fan-out: branches share no state, and the merger assumes every branch's key exists → have branch tools return structured error dicts with timeouts set in tool code, write one `output_key` per branch, make the merger handle missing or error keys and label the result as partial, and in a graph route on the failing node's output	Rewriting the fan-out as a `SequentialAgent`
Remote tasks pile up unanswered, users never see the peer's question, and coordinator connections stay open	Orphaned `input-required` task: the peer returned `TASK_STATE_INPUT_REQUIRED`, which is interrupted but not terminal, and nothing surfaced or closed it → if the card declares the `pushNotifications` capability, register a webhook with `CreateTaskPushNotificationConfig` (`url`, `token`) instead of holding a stream open, store the task ID with a deadline in session state, and call `CancelTask` when it expires (the task ends in `TASK_STATE_CANCELLED` when the peer supports cancellation)	Raising the HTTP client timeout
ADK 2.0 `Workflow` graph ignores `RequestInput` approval pauses or `RetryConfig` automatic retries never fire	Catching broad `BaseException` (or bare `except:`) traps `NodeInterruptedError` and breaks `RequestInput` pauses, while broad `except Exception:` inside tools/nodes masks errors from `RetryConfig(max_attempts=...)` → let `BaseException` and transient `Exception`s propagate, catching only specific recoverable domain errors	Replacing `RequestInput` with `require_confirmation=True` on `VertexAiSessionService`
Eventarc / Pub/Sub-triggered Cloud Run agent issues duplicate refunds or ERP tickets on slow turns	Pub/Sub at-least-once delivery (`600s` max ack deadline) redelivers unacknowledged events while the turn is still running → perform an atomic `messageId` idempotency check in Memorystore or Firestore before executing mutative tools, configure a Dead-Letter Topic (`maxDeliveryAttempts` ≥ 5), or throttle via Cloud Tasks (`maxDispatchesPerSecond`)	Setting Pub/Sub `ackDeadlineSeconds` to `3600`
Multi-MB PDFs sent over A2A fail at Agent Gateway `CONTENT_AUTHZ`, or attaching `AgentRegistry.get_remote_a2a_agent()` to two workflow branches raises `ValueError`	Inline base64 A2A `Part` blobs saturate gateway payload inspection and every ADK agent instance obeys the single-parent rule → persist binaries in Cloud Storage via `GcsArtifactService` (`Runner` raises `ValueError` if `artifact_service=None`), pass `gs://` URIs across A2A, and resolve `get_remote_a2a_agent()` once at startup calling `.clone()` per additional branch or parent	Disabling `CONTENT_AUTHZ` on Agent Gateway and resolving the registry inside every turn

17. Glossary

- **`Runner`**: ADK's stateless execution engine (`run_async`, `run_live`) implementing the commit-then-resume event loop across `SessionService`, `ArtifactService`, and `MemoryService`.
- **`LlmAgent` vs. `Workflow` (ADK 2.0):** probabilistic model-driven agent (`mode="chat"` or `mode="single_turn"`) vs. ADK 2.0's deterministic/conditional graph engine (`edges`, `BaseNode`, `RequestInput`, `RetryConfig`) across Python, TypeScript, and Go.
- **`ManagedAgent`**: ADK wrapper around the Agent Platform Managed Agents API (`Interactions API` in `global`) supporting server-side tools only (`NotImplementedError` on client `FunctionTool` or `McpToolset`).
- **`Session` / `Event` / `State`**: single-conversation append-only event stream and prefix-scoped scratchpad (`none` session, `user:`, `app:`, `temp:` current-invocation-only).
- **`VertexAiMemoryBankService`**: managed cross-session long-term semantic memory service that asynchronously extracts, consolidates, and revises durable user facts from completed sessions.
- **`GcsArtifactService`**: Cloud Storage-backed binary artifact store (`{app}/{user}/{session}/{filename}/{version}` and `user:` scope) used to keep multi-MB blobs out of `session.state` and inline A2A `Part` payloads.
- **`AgentTool` vs. `sub_agents`**: explicit tool-like specialist invocation in an isolated child session vs. control-flow handoff (`transfer_to_agent`) sharing `session.events` under the single-parent rule (`.clone()` required to reuse an instance).
- **`ApplicationIntegrationToolset` / `ToolboxToolset` / `APIHubToolset` / `McpToolset`**: ADK's four enterprise tool connectors for Integration Connectors (100+ SaaS/ERPs with Auth Manager 3LO), MCP Toolbox pooled SQL databases (`tools.yaml`), Apigee API Hub OpenAPI specs, and Streamable HTTP / stdio MCP servers.
- **`RequestInput` & `NodeInterruptedError`**: ADK 2.0 durable human-in-the-loop pause primitive that raises a `BaseException` subclass to checkpoint graph execution cleanly across restarts.
- **`before_tool_callback` / `after_tool_callback` & `BasePlugin`**: deterministic single-agent interception hooks and `Runner`-wide cross-cutting guardrails (such as `ReflectAndRetryToolPlugin` and tool-boundary SDP redaction).
- **`AgentRegistry.get_remote_a2a_agent()`**: startup-time discovery helper resolving governed A2A Agent Cards from Agent Registry for cross-runtime delegation through Agent Gateway.
- **`adk eval` vs. `agents-cli eval`**: local/CI trajectory grading on `*.evalset.json` files vs. non-interactive production dataset evaluation (`EvaluationDataset` + `eval_config.yaml`) wired into Agent Platform CI/CD pipelines.