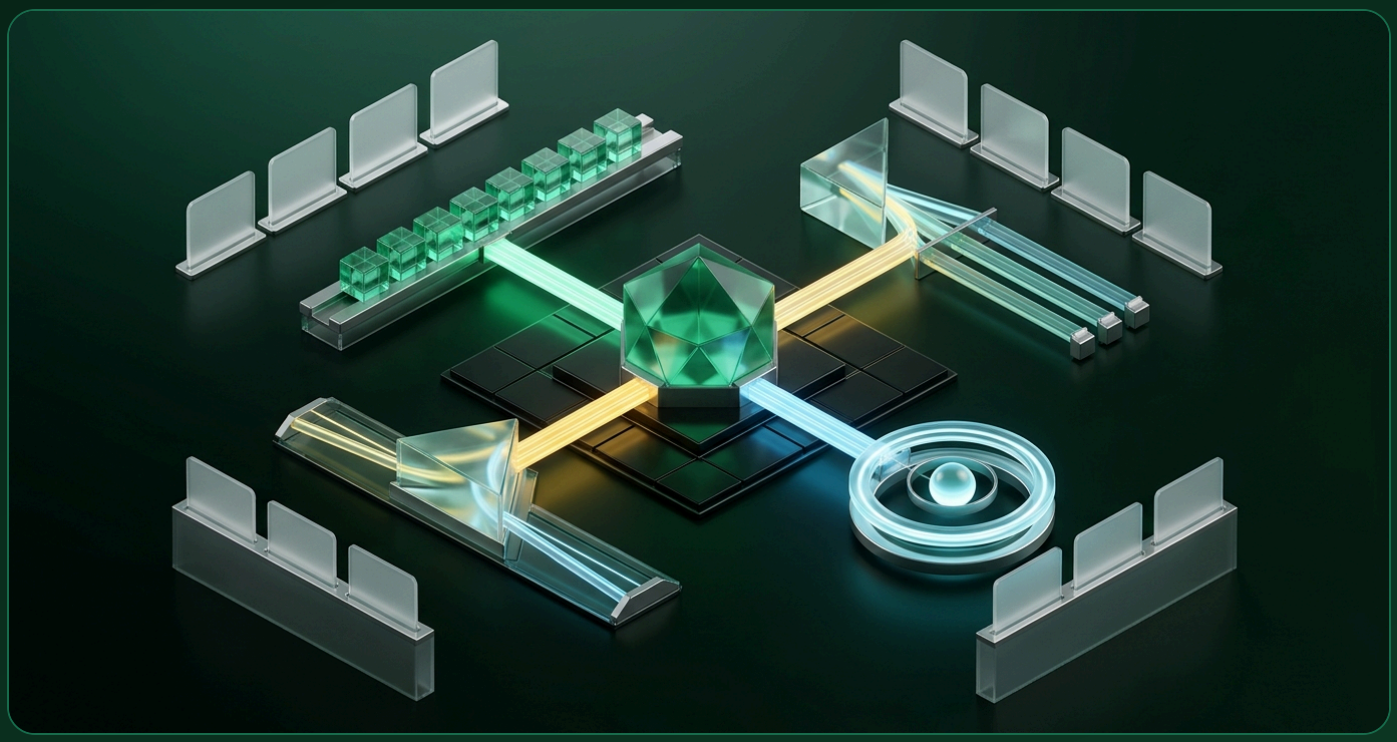
GOOGLE AI & CREATIVE STUDIOS • FIELD GUIDE

Google ADK & Agent2Agent (A2A) Protocol

Build, evaluate, and ship code-first multi-agent systems with Google's Agent Development Kit in Python and TypeScript—combining deterministic Workflow Agents, scoped state prefixes, `McpToolset`, Vertex AI Agent Engine, and cross-vendor A2A Agent Cards.

Python & TypeScript ADK | Sequential, Parallel & Loop | Scoped State & Callbacks | `adk web` & `adk eval` | A2A `agent-card.json` | Vertex AI Agent Engine



Deterministic Orchestration

Eliminate fragile routing loops with code-driven `SequentialAgent`, `ParallelAgent`, and `LoopAgent` controllers.

Open A2A Interoperability

Bridge Python, TypeScript, Go, and third-party agents across networks via `to_a2a()` and `RemoteA2aAgent`.

Eval-to-Cloud Lifecycle

Trace every turn in `adk web`, gate CI with `adk eval` rubrics, and deploy to Cloud Run or Vertex AI Agent Engine.

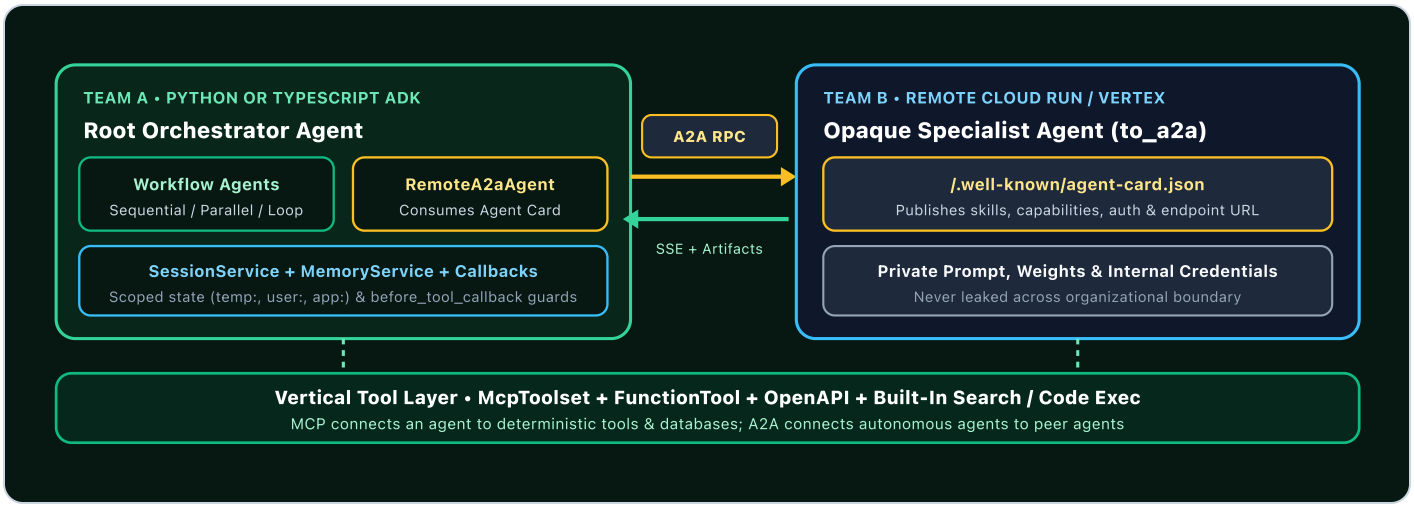
Code-First Agents Meet Open Network Interoperability

Prompt-only multi-agent setups break down when teams scale across repos, languages, and security boundaries. **Google Agent Development Kit (ADK)** gives you software-engineering rigor inside each agent, while the Linux Foundation **Agent2Agent (A2A) Protocol** connects independent agents across networks without exposing private prompts or credentials.

Quick Navigation

1. ADK + A2A Mental Model p. 2
2. Setup, Quotas & Runtime Matrix p. 3
3. Annotated `adk web` UI Tour p. 4
4. Workflow Agents in Motion p. 5
5. State Prefixes, Callbacks & MCP p. 6
6. Troubleshooting & Pre-Flight p. 12
7. A2A Protocol & Agent Cards p. 7
8. Evals (`adk eval`) & Deployment p. 8
9. 5 Copy-Paste Production Recipes p. 9
10. Security, IAM & Memory Isolation p. 10
11. Cross-Tool Synergy Pipeline p. 11

Architecture: Vertical Tooling (MCP) vs. Horizontal Delegation (A2A)



The Golden Rule: MCP for Tools, A2A for Peers

Use **MCP** (`McpToolset`) when your agent needs structured functions like querying SQL or reading Git files. Use **A2A** (`RemoteA2aAgent`) when delegating a multi-turn goal to another team's specialist agent that maintains its own memory, reasoning loop, and private secrets.

Installation, Runtime Selection & Vertex Agent Engine Meters

ADK ships first-class SDKs for Python, TypeScript, Go, and Java. Start locally against a Gemini API key in seconds, then promote the exact same agent graph to Cloud Run or managed Vertex AI Agent Engine with zero rewrites.

1 Python ADK Bootstrap (`google-adk`)

Install `google-adk` (plus the optional `[a2a]` extra for cross-network serving) and scaffold a ready-to-run package:

```
pip install "google-adk[a2a]"
adk create support_router --model gemini-2.5-flash

# Launch interactive trace UI on localhost:8000
adk web
# Or run directly in terminal / expose REST API
adk run support_router
adk api_server --port 8080
```

2 TypeScript ADK Bootstrap (`@google/adk`)

Requires Node.js 24.13+ and Zod schemas for typed `FunctionTool` definitions, paired with `@google/adk-devtools`:

```
mkdir ts-agent && cd ts-agent && npm init -y
npm pkg set type="module"
npm i @google/adk zod dotenv
npm i -D @google/adk-devtools typescript tsx

# Launch local Dev UI or terminal runner
npx adk web
npx adk run src/agent.ts
```

Vertex AI Agent Engine Meters & Default Quotas

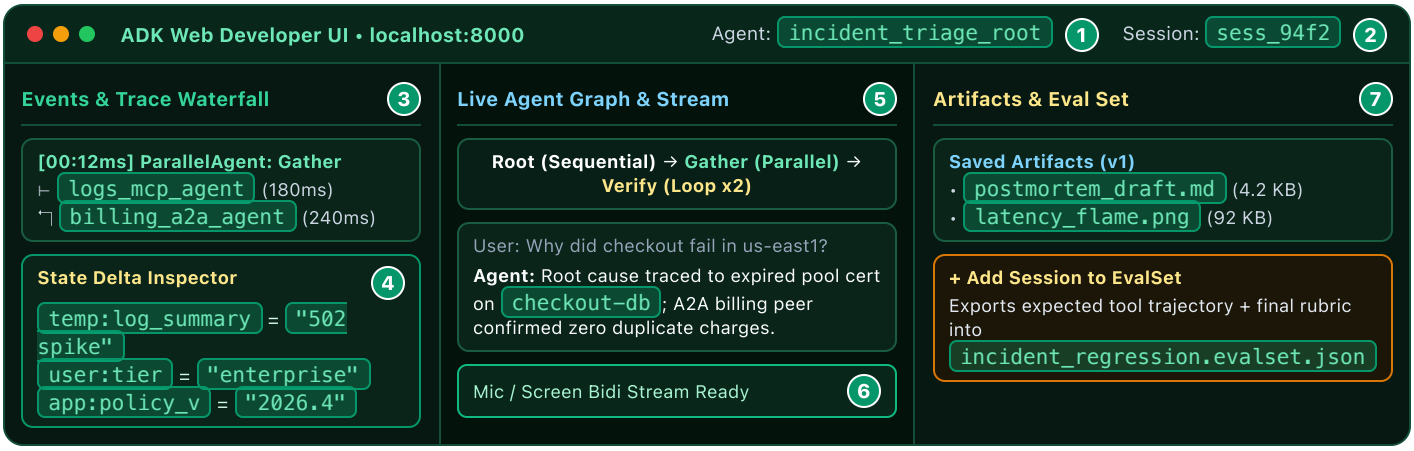
Resource / Meter	On-Demand Rate	Monthly Free Tier	Default Regional Quota
Agent Compute & Sandbox	`$0.085` / vCPU-hour (active only; $0 idle)	First **50 vCPU-hours** free / project	`90 RPM` queries; `1,000 RPM` sandbox execs
Agent Memory (RAM)	`$0.009` / GiB-hour while active	First **100 GiB-hours** free / project	`1,000` active sandbox entities / region
Sessions & Memory Bank	`~$0.30` / GiB-month stored events	First **1 GiB-month** free	`10,000 RPM` session reads; `100 RPM` writes
A2A & Live Bidi Streams	Standard compute + model tokens	Included in runtime tier	`60 RPM` A2A POSTs; `10/min` BidiStream connects

Deployment Target Decision Matrix

Target	Command	Best When You Need	Trade-Off
Vertex Agent Engine	`adk deploy agent_engine`	Turnkey managed Sessions, Memory Bank & Code Execution Sandbox	Opinionated Python-first managed runtime
Google Cloud Run	`adk deploy cloud_run --with_ui`	Custom Dockerfiles, Python or TypeScript, public A2A endpoints, scale-to-zero	Bring your own Postgres/Redis session store for multi-instance state
GKE / Self-Hosted	`adk deploy gke` / Uvicorn	VPC-isolated GPUs, strict data residency, sidecar service meshes	Highest Kubernetes operational overhead

Annotated Tour: Debugging Traces & State in `adk web`

Running `adk web` (or `npx adk web` for TypeScript) opens a local browser workbench at `http://localhost:8000`. Every prompt, sub-agent transfer, tool payload, state mutation, and saved golden eval lives one click away.



Control Legend: What Each Numbered Zone Controls

1 **Agent Package Picker:** Hot-switches between every folder under your workspace containing a root `agent.py` or `agent.ts` export without restarting the server.

2 **Session Switcher:** Creates fresh `sessionId` runs or replays SQLite-persisted conversation histories to compare prompt edits side by side.

3 **Event & Latency Waterfall:** Click any span to inspect exact Gemini request tokens, function arguments, MCP tool returns, and remote A2A JSON-RPC payloads.

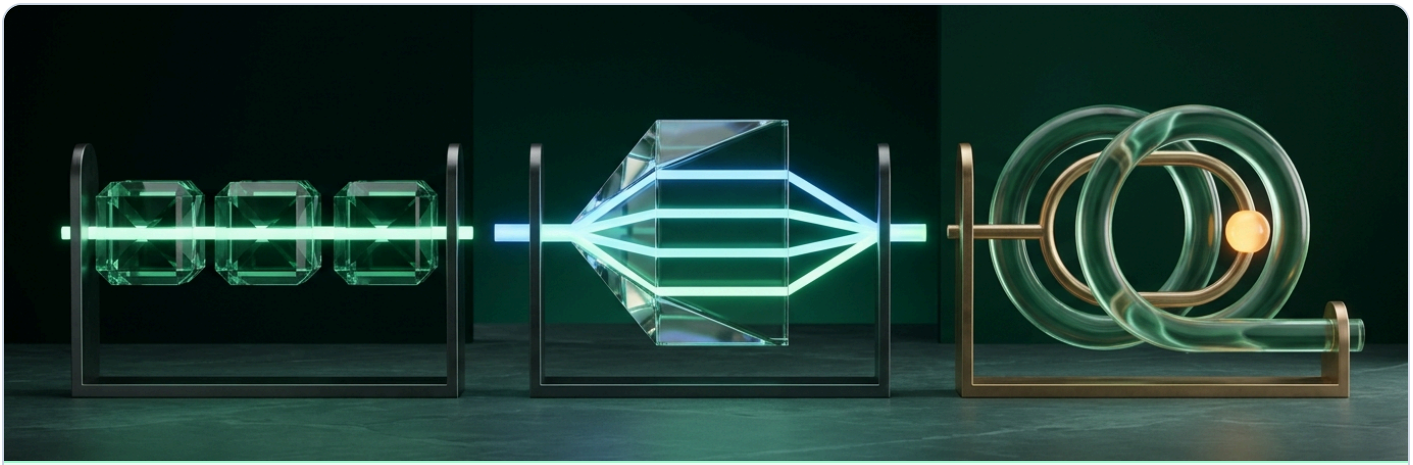
4 **State Delta Diff:** Highlights every key written via `output_key` or `tool_context.state` during that exact turn across `temp:`, `user:`, and `app:` scopes.

5 **Interactive Topology Graph:** Highlights the currently executing node in real time so you can spot infinite loops or misrouted sub-agent transfers immediately.

6 **Live Audio/Video &** 7 **Eval Export:** Stream live mic/camera input for bidirectional voice testing, then freeze passing runs into `.evalset.json` files for CI.

Deterministic Workflow Agents: Zero-Hallucination Control Flow

Asking an LLM router to decide every next step wastes latency and invites skipped checks. ADK separates **reasoning nodes** (`LlmAgent`) from **deterministic workflow controllers** (`SequentialAgent`, `ParallelAgent`, and `LoopAgent`) that execute in predictable code order.



Three Composable Control Topologies: Chain strict stages with `SequentialAgent` (left), fan out concurrent research with `ParallelAgent` (center), and iterate draft-critic cycles until quality passes with `LoopAgent` (right).

1 `SequentialAgent`

Runs `sub_agents` strictly one after another. Pair each stage's `output_key="parsed_spec"` with `{parsed_spec}` placeholders in the next agent's instruction.

2 `ParallelAgent`

Executes independent sub-agents concurrently in the same session, cutting wall-clock latency when querying docs, logs, and remote A2A peers simultaneously.

3 `LoopAgent`

Repeats its sub-agents until a reviewer tool sets `tool_context.actions.escalate = True` or the safety ceiling `max_iterations` is reached.

Composing All Three Primitives in 18 Lines of Python

```
from google.adk.agents import LlmAgent, SequentialAgent, ParallelAgent, LoopAgent
from google.adk.tools import ToolContext

def approve_draft(tool_context: ToolContext) -> dict:
    """Call only when the security report meets every citation requirement."""
    tool_context.actions.escalate = True  # Cleanly exits the enclosing LoopAgent
    return {"status": "approved"}

gather = ParallelAgent(name="GatherIntel", sub_agents=[cve_agent, repo_scan_agent])
writer = LlmAgent(name="Writer", model="gemini-2.5-pro",
                  instruction="Synthesize {cve_notes} and {scan_notes}.", output_key="draft")
critic = LlmAgent(name="Critic", model="gemini-2.5-flash", tools=[approve_draft],
                  instruction="Audit {draft}. Request fixes or call approve_draft.")

refine = LoopAgent(name="RefineLoop", sub_agents=[writer, critic], max_iterations=3)
root_agent = SequentialAgent(name="AuditPipeline", sub_agents=[gather, refine])
```

Distinct Output Keys Inside `ParallelAgent`

Because all branches of a `ParallelAgent` share the same active session state, assign each branch a unique `output_key` (such as `cve_notes` vs. `scan_notes`) so concurrent completions never overwrite each other.

Scoped State Prefixes, Callbacks & `McpToolset`

Production agents need strict boundaries around what persists across turns and which tools are allowed to fire. ADK bakes persistence scopes directly into state key prefixes and wraps every model and tool boundary with synchronous guard callbacks.

1 Four Built-In State Scopes

- **No prefix (`state["draft"]`):** Bound to the current `sessionId` conversation thread.
- **`temp:` (`state["temp:raw_sql"]`):** Ephemeral scratchpad wiped automatically when the current invocation turn finishes.
- **`user:` (`state["user:timezone"]`):** Persists across all sessions belonging to the same authenticated `userId`.
- **`app:` (`state["app:feature_flags"]`):** Global read-mostly configuration shared across every user of the application.

2 Six Lifecycle Callback Intercepts

- **`before_agent` / `after_agent`:** Check tenant quotas, inject trace tags, or redact PII on final responses.
- **`before_model` / `after_model`:** Short-circuit cached prompts or inspect LLM outputs before tool dispatch.
- **`before_tool_callback`:** Inspect `(tool, args, tool_context)`; return a dict to **block execution** cleanly without crashing the agent loop.
- **`after_tool_callback`:** Truncate oversized payloads or save binary blobs via `tool_context.save_artifact()`.

Connecting External MCP Servers & Blocking Unsafe Calls

```
from google.adk.agents import LlmAgent
from google.adk.tools.mcp_tool import McpToolset
from google.adk.tools.mcp_tool.mcp_session_manager import StdioConnectionParams
from mcp import StdioServerParameters

def block_destructive_sql(tool, args: dict, tool_context):
    sql = str(args.get("query", "")).upper()
    if any(op in sql for op in ("DROP ", "TRUNCATE ", "DELETE ")):
        return {"error": "Blocked by policy: read-only SELECT queries required."}
    return None  # Returning None allows normal tool execution

db_tools = McpToolset(
    connection_params=StdioConnectionParams(
        server_params=StdioServerParameters(command="npx", args=["-y", "@bytebase/dbhub"]),
    ),
    tool_filter=["execute_sql", "list_tables"],  # Expose only approved MCP tools
)
analyst = LlmAgent(name="SqlAnalyst", model="gemini-2.5-flash",
                   tools=[db_tools], before_tool_callback=block_destructive_sql)
```

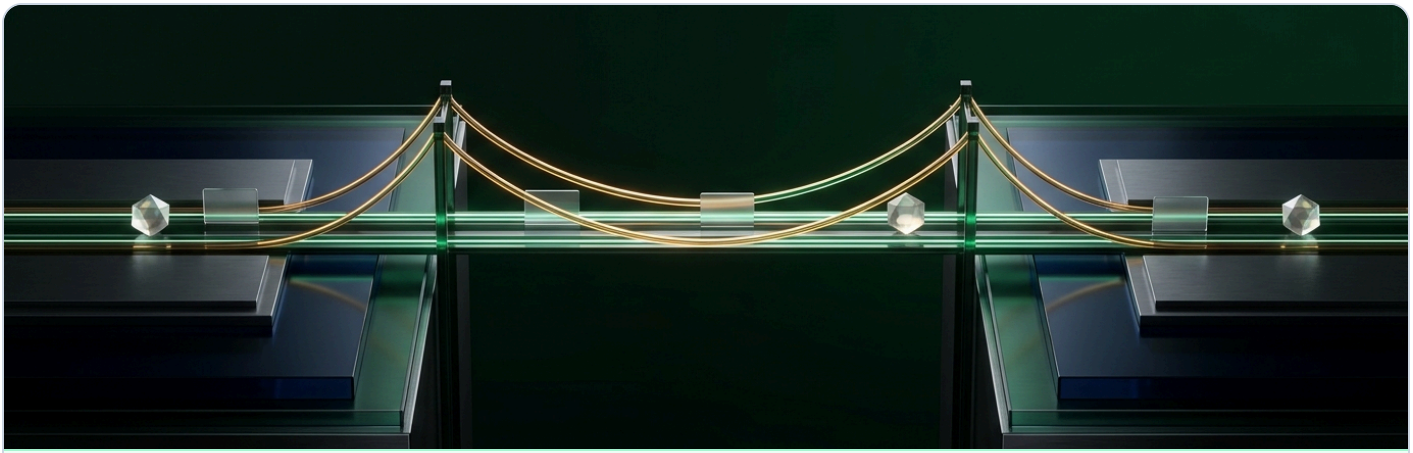
Color-Coded Blueprint: Anatomy of a Production ADK Architecture Spec

Wrap a ParallelAgent intake stage and a 2-pass LoopAgent verifier inside a SequentialAgent root, assign gemini-2.5-flash to fast triage leaf agents and gemini-2.5-pro to the synthesis writer, store scratch payloads in temp:raw_chunks and user preferences in user:locale, filter McpToolset with tool_filter plus a before_tool_callback blocking mutating SQL, delegate billing disputes via RemoteA2aAgent pointing to /.well-known/agent-card.json, and gate deployment on adk eval passing 0.90 tool_trajectory_avg_score.

Workflow Topology | Model Tiering | State Scoping | MCP + Guardrail | A2A Delegation | CI Eval Gate

Speaking A2A: Agent Cards, Tasks & Cross-Network Handoffs

When your Python ADK orchestrator needs help from a finance team's Java service or a partner's LangGraph agent, the **Agent2Agent (A2A)** standard lets them collaborate over JSON-RPC 2.0 and Server-Sent Events (SSE) using self-describing **Agent Cards**.



Opaque Peer Collaboration: Each domain keeps its internal prompts, databases, and keys sealed behind its own boundary while exchanging typed A2A Tasks, streaming status events, and structured Artifacts across the bridge.

1. Expose Any ADK Agent via `to_a2a()`

```
# server.py — serves /.well-known/agent-card.json
from google.adk.agents import LlmAgent
from google.adk.a2a.utils.agent_to_a2a import to_a2a

billing_agent = LlmAgent(
    name="billing_specialist",
    model="gemini-2.5-flash",
    description="Audits invoices and issues SLA
credits.",
    instruction="Verify invoice IDs before credit
quotes.",
    tools=[lookup_invoice, calculate_sla_credit],
)
a2a_app = to_a2a(billing_agent, port=8001)
# Run: uvicorn server:a2a_app --host 0.0.0.0 --port 8001
```

2. Consume a Remote Peer via `RemoteA2aAgent`

```
# coordinator.py — zero custom HTTP boilerplate
from google.adk.agents import LlmAgent
from google.adk.agents.remote_a2a_agent import (
    RemoteA2aAgent, AGENT_CARD_WELL_KNOWN_PATH
)

remote_billing = RemoteA2aAgent(
    name="billing_peer",
    description="Delegates invoice & SLA credit checks.",

agent_card=f"https://billing.internal{AGENT_CARD_WELL_KNO
WN_PATH}",
)
root_agent = LlmAgent(
    name="support_lead", model="gemini-2.5-pro",
    sub_agents=[remote_billing],
    instruction="Delegate billing questions to
billing_peer."
)
```

Core A2A Protocol Methods & Task Lifecycle

JSON-RPC Method	Transport Behavior	Task Lifecycle States & Payload Semantics
`message/send`	Sync/Async HTTP POST	Sends text, file, or JSON `parts` with `contextId`; returns a reply or stateful `Task`.
`message/stream`	SSE Streaming POST	Streams incremental progress chunks, status updates, and partial artifacts in real time.
`tasks/get` & `tasks/cancel`	Task Polling & Abort	Tracks long-running states: `submitted` → `working` → `input-required` → `completed` / `failed`.

Regression Testing with `adk eval` & One-Command Deploys

In multi-agent systems, a final answer that sounds polite is still a bug if the agent skipped an authorization tool or called the wrong A2A peer. `adk eval` grades both the **exact tool-call trajectory** and the **semantic response quality** on every pull request.

1 Configure Rubrics in `test_config.json`

Set strict pass thresholds for both tool sequence accuracy and response similarity:

```
{
  "criteria": {
    "tool_trajectory_avg_score": 1.0,
    "response_match_score": 0.85
  }
}
```

Run in CI: `adk eval ./support_router tests/refunds.evalset.json --config_file_path=tests/test_config.json --print_detailed_results`

2 TypeScript Runner & Zod Tool Pattern

In `@google/adk`, combine Zod-typed tools with `InMemoryRunner` inside Vitest suites:

```
import { LlmAgent, FunctionTool, InMemoryRunner }
from "@google/adk";
import { z } from "zod";

const lookupOrder = new FunctionTool({
  name: "lookup_order",
  description: "Fetch order status by ID.",
  parameters: z.object({ orderId: z.string() }),
  execute: async ({ orderId }) => ({ orderId, state:
"shipped" }),
});
export const rootAgent = new LlmAgent({
  name: "ops_ts", model: "gemini-2.5-flash", tools:
[lookupOrder]
});
```

3. Shipping to Cloud Run or Vertex AI Agent Engine

```
# Option A: Serverless container on Google Cloud Run (supports Python & TypeScript + A2A)
adk deploy cloud_run \
  --project=$GOOGLE_CLOUD_PROJECT \
  --region=us-central1 \
  --service_name=support-router-a2a \
  ./support_router

# Option B: Managed Vertex AI Agent Engine with persistent Sessions & Memory Bank
adk deploy agent_engine \
  --project=$GOOGLE_CLOUD_PROJECT \
  --region=us-central1 \
  --staging_bucket=gs://$GOOGLE_CLOUD_PROJECT-adk-staging \
  ./support_router
```

Swap InMemory Stores Before Multi-Instance Scaling

`InMemorySessionService` resets whenever a container scales down. On Cloud Run, pass `--session_service_uri="postgresql+asyncpg://..."` (DatabaseSessionService) or point to Vertex AI Agent Engine Sessions so users never lose mid-conversation state across load-balanced replicas.

5 Real-World Copy-Paste Recipes

Drop these five battle-tested snippets straight into your Python or TypeScript repository to wire sub-agent tools, A2A manifests, artifact storage, CI gates, and Cloud Run deployments cleanly on the first try.

Recipe & Goal	Copy-Paste Snippet / Command	Expected Outcome
1. Wrap Specialist as `AgentTool` (Keep Control)	`from google.adk.tools.agent_tool import AgentTool` `coordinator = LlmAgent(name="Lead", model="gemini-2.5-pro", tools=[AgentTool(agent=math_specialist)])`	Calls `math_specialist` like a function and returns control straight back to `Lead` instead of handing off the chat.
2. Inspect a Live A2A Agent Card Manifest	`curl -s https://billing.internal/.well-known/agent-card.json | jq '{name, version, capabilities, skills: [.skills[].id]}'`	Verifies published A2A capabilities (`streaming`), skill IDs, and RPC URL before wiring `RemoteA2aAgent`.

3. Save Versioned Binary Artifacts in Tools	`from google.genai import types`
	`part = types.Part.from_bytes(data=pdf_bytes, mime_type="application/pdf")`
	`ver = await tool_context.save_artifact("quote.pdf", part)`
		Stores heavy PDFs/CSVs out of the LLM token window while tracking monotonic revision numbers (`v0`, `v1`).

4. Deterministic CI Trajectory Gate	`adk eval ./agents/triage tests/golden.evalset.json --config_file_path=tests/strict.json --print_detailed_results`	Fails the GitHub Action or Cloud Build step immediately if tool order or arguments drift from golden runs.
5. Postgres-Backed Cloud Run Deploy	`adk deploy cloud_run --project=$PROJECT_ID --region=us-central1 --service_name=ops-agent --session_service_uri="$DB_URI" ./agents/ops`	Builds and ships a scale-to-zero container whose sessions survive restarts in Cloud SQL / Postgres.

`sub_agents` Transfer vs. `AgentTool` Invocation: Know the Difference

Listing an agent inside `sub_agents=[...]` lets the parent transfer active conversation ownership to that child for follow-up turns. Wrapping it in `AgentTool(agent=...)` executes the child in an isolated call, captures its final summary, and keeps the parent orchestrator strictly in charge of user replies.

Security, A2A Auth & Multi-Tenant Memory Isolation

Connecting autonomous agents across team boundaries multiplies blast radius if identity, memory scopes, or remote Agent Cards are left unverified. Enforce these four enterprise guardrails before exposing any ADK service outside localhost.

1 Authenticated A2A Agent Cards

- **Declare `securitySchemes`:** Publish OAuth 2.0, OpenID Connect, or mTLS requirements directly inside `agent-card.json` rather than relying on network obscurity.
- **Cloud Run OIDC Service-to-Service:** Require IAM invoker tokens between internal Cloud Run A2A peers so rogue workloads cannot call `/v1/message:send`.
- **Treat Remote Replies as Untrusted:** Validate structured A2A payloads against Pydantic or Zod schemas before feeding them into privileged local tools.

2 Prevent Cross-Tenant Memory Poisoning

- **Bind `userId` from Verified JWTs:** Never let prompt text or client JSON body fields supply the `userId` passed to `SessionService` or Vertex Memory Bank.
- **Audit `app:` Writes:** Restrict `app:*` state keys to startup config or admin pipelines; letting user turns mutate `app:` leaks data globally across tenants.
- **Scrub Secrets into `temp:`:** Short-lived OAuth exchange tokens belong in `ToolContext` auth handlers or `temp:`, never persisted in session history.

3 Least-Privilege `McpToolset` Filtering

Third-party MCP servers often ship 25+ read, write, and delete tools in one bundle. Always pass an explicit allowlist via `tool_filter=["get_ticket", "search_docs"]` and pair mutating tools with human confirmation (`input-required` A2A state or ADK tool confirmation).

4 Bounded Loops & Isolated Code Sandboxes

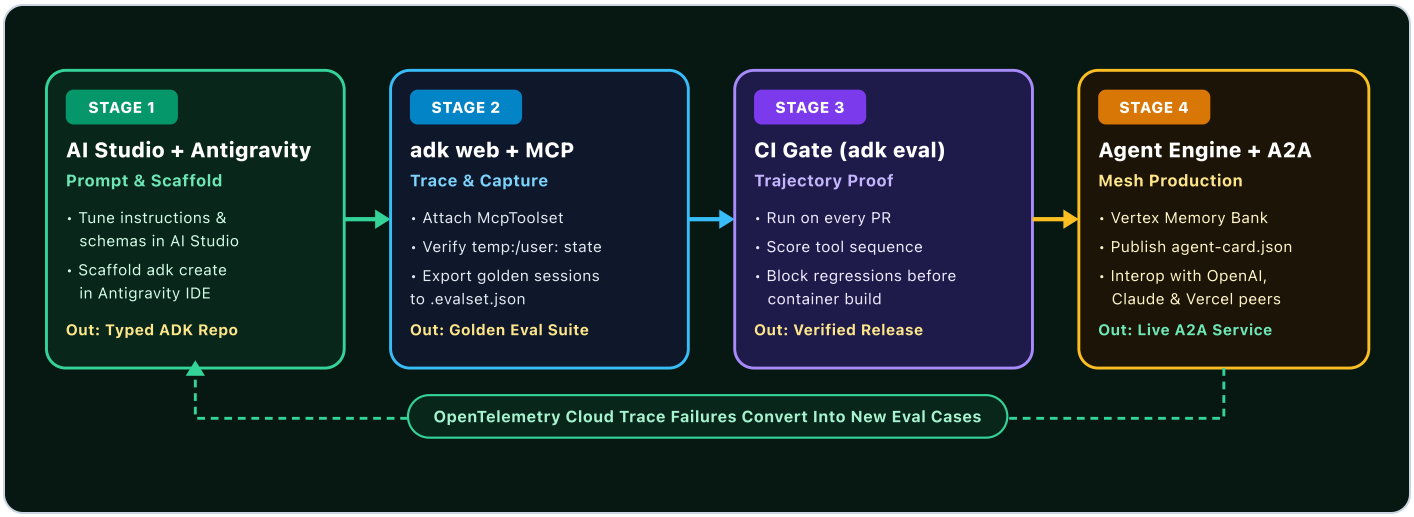
Always set a low `max_iterations` (typically `2`–`4`) on every `LoopAgent` to prevent runaway token bills, and route dynamic Python calculation tasks to Vertex AI Agent Engine's isolated **Code Execution Sandbox** instead of host `subprocess` calls.

Never Forward Raw User Credentials Over A2A Messages

A2A preserves agent opacity in both directions. Exchange scoped, audience-restricted OAuth tokens via protocol headers rather than pasting user API keys or session cookies inside conversational `message/send` text parts where remote LLM logs would capture them.

Cross-Tool Synergy: From AI Studio Prototype to Multi-Cloud A2A Mesh

ADK shines brightest as the connective backbone between rapid prompt exploration, local IDE coding agents, CI trajectory gates, and cross-framework enterprise deployments.



A Bridging Frontend Apps (Next.js / Android)

Connect Next.js or Kotlin mobile clients to `adk api_server` or Vertex Agent Engine SSE endpoints for user-facing chat while letting the backend orchestrator fan out over private internal A2A microservices.

B Polyglot Team Autonomy

Data science teams can ship Python ADK agents on Vertex Agent Engine while web platform teams ship TypeScript `@google/adk` services on Cloud Run—both discovering each other via standard `/.well-known/agent-card.json` manifests.

Troubleshooting Clinic & 6-Point Pre-Flight Checklist

Diagnose the seven most common ADK and A2A pitfalls in seconds, then run the 6-point pre-flight check before promoting any multi-agent workflow to production.

Symptom / Error	10-Second Root-Cause Fix
1. `adk web` dropdown shows zero agents	Run `adk web` from the **parent directory** above your agent package folder, and ensure `__init__.py` imports `agent` which defines `root_agent`.
2. `{draft}` placeholder raises `KeyError` in instruction	The upstream agent failed or has not set `output_key="draft"` yet. Initialize default keys in session state or append `?` (`{draft?}`) for optional state interpolation.
3. `LoopAgent` burns tokens without ever stopping	No tool set `tool_context.actions.escalate = True` and `max_iterations` was omitted. Always pass `max_iterations=3` and expose an explicit exit tool.
4. LLM ignores your Python custom function tool	ADK generates tool schemas from type hints and docstrings. Add explicit parameter types, return type `dict`, and a clear docstring explaining *when* to call it.
5. `RemoteA2aAgent` fails on `localhost` inside Docker	`agent-card.json` advertised `http://localhost:8001` instead of the reachable service hostname/HTTPS URL. Pass public `host` / `port` or explicit RPC URL to `to_a2a()`.
6. Cloud Run users lose history mid-conversation	Default `InMemorySessionService` lives inside container RAM. Configure `--session_service_uri` with Cloud SQL Postgres or deploy on Vertex AI Agent Engine.
7. `McpToolset` hangs or leaks zombie subprocesses	Stdio MCP servers require clean async lifecycle management. Use current `google-adk` async runner patterns and restrict slow servers via `tool_filter`.

6-Point Pre-Flight Checklist Before Production Deploy

- [x] **Deterministic Control First:** Replaced open-ended LLM routers with `SequentialAgent`, `ParallelAgent`, and capped `LoopAgent` nodes.
- [x] **Scoped State Hygiene:** Verified `temp:`, `user:`, and `app:` prefixes with JWT-derived `userId` isolation.
- [x] **Filtered MCP & Callbacks:** Applied `tool_filter` on every `McpToolset` plus `before_tool_callback` policy guards.
- [x] **Verified A2A Agent Card:** Checked `/.well-known/agent-card.json` public URLs, `securitySchemes`, and OIDC auth.
- [x] **Durable Session Backend:** Replaced `InMemorySessionService` with Postgres or Vertex AI Agent Engine Sessions.
- [x] **Green `adk eval` Gate:** Golden `.evalset.json` trajectory and rubric scores pass automatically in CI.