perplexity

Google Agent Development Kit (ADK) - Comprehensive Technical Architecture Guide

Introduction and Foundational Concepts

This comprehensive technical reference manual provides an in-depth exploration of Google's Agent Development Kit (ADK), focusing on the architectural principles, design philosophies, and implementation strategies that make ADK a powerful framework for building sophisticated AI agent systems[1] [2] [3].

Google's Agent Development Kit represents a paradigm shift in how developers approach agent system construction. Unlike traditional frameworks that focus primarily on orchestration or simple function calling, ADK was architecturally designed to make agent development feel more like software development[1]. This philosophical approach permeates every aspect of the framework, from its modular architecture to its comprehensive tooling ecosystem.

The Core Design Philosophy

ADK's architecture is built on several foundational principles that distinguish it from other agent frameworks. First and foremost is **model agnosticism**[1] [2]. While optimized for Google's Gemini models, the framework deliberately avoids vendor lock-in through its abstraction layers. This means that the same agent logic can seamlessly work with OpenAI's GPT models, Anthropic's Claude, or any other LLM through the LiteLLM integration layer[2] [4].

The second core principle is **deployment agnosticism**[1]. ADK agents can run locally during development, scale with Vertex AI Agent Engine in production, or integrate into custom infrastructure using containers. This flexibility ensures that architectural decisions about deployment don't constrain the agent design itself.

Third is **framework compatibility**[1]. ADK was designed to interoperate with existing agent frameworks like LangGraph and CrewAI through standardized protocols like Agent2Agent (A2A) [4]. This approach acknowledges that enterprises often have heterogeneous agent ecosystems and need frameworks that can coexist rather than replace existing investments.

The Event-Driven Architecture Foundation

At its architectural core, ADK operates on an **event-driven pattern**[5]. This is fundamentally different from traditional request-response patterns used in many agent frameworks. The event loop creates a clear separation between the Runner component and the execution logic (agents, tools, callbacks)[5].

This architecture provides several critical advantages. First, it enables **real-time observability**. Every action an agent takes generates events that can be streamed to monitoring systems, user

interfaces, or other agents[5]. Second, it supports **state consistency** by ensuring that state changes are committed atomically with event processing. Third, it enables **complex orchestration** where multiple agents can coordinate through event streams without tight coupling.

The event loop operates through a yield-and-resume pattern[5]. When an agent needs to perform an action (like calling an LLM or executing a tool), it yields an event to the Runner. The Runner processes the event, potentially updating state or calling external services, then allows the agent to resume. This pattern ensures that all state changes are properly managed and observable.

Agent Architecture and Classification System

The BaseAgent Foundation

ADK's agent architecture is built on a clear inheritance hierarchy with `BaseAgent` as the foundational class[3] [6]. This design choice reflects a software engineering approach where common functionality is abstracted into base classes while specific behaviors are implemented in derived classes.

The `BaseAgent` class provides essential infrastructure that all agents need: session management, event generation, state access, and lifecycle hooks[3]. This foundation ensures consistency across different agent types while allowing for specialized implementations. The base class also defines the contract that the Runner expects, enabling polymorphic treatment of different agent types.

LLM Agents: The Intelligence Layer

LLM Agents (`LlmAgent` or `Agent`) represent the "thinking" component of the system[3]. These agents leverage Large Language Models for reasoning, understanding natural language, making decisions, and dynamically determining how to proceed[3]. The technical architecture of LLM agents is sophisticated and warrants detailed exploration.

The Instruction Parameter: The Behavioral Nucleus

The `instruction` parameter serves as the **behavioral nucleus** of an LLM agent[2] [3]. Unlike simple prompts, instructions in ADK are architectural components that define the agent's identity, capabilities, and operational constraints. The instruction system supports template syntax that enables dynamic content injection[7].

The template system uses `{variable}` syntax to inject values from session state, `{artifact.filename}` to include artifact content, and `{variable?}` to handle optional variables gracefully[7]. This templating capability transforms instructions from static text into dynamic, context-aware behavioral specifications.

Instructions should encompass several critical dimensions: **role definition** (what the agent is), **capability boundaries** (what it can and cannot do), **tool usage guidance** (how and when to use

available tools), **output formatting requirements**, and **error handling strategies**[3]. Well-crafted instructions serve as both behavioral constraints and capability enablers.

Decision-Making Architecture

LLM agents make decisions through a sophisticated reasoning process that ADK orchestrates[8]. When presented with a user request, the agent's LLM analyzes multiple inputs: the system instruction, conversation history, current session state, available tools, and the specific user query[8].

The decision-making process follows a structured pattern: **context analysis** (understanding the current situation), **goal identification** (determining what needs to be accomplished), **capability assessment** (evaluating available tools and sub-agents), **strategy formulation** (deciding on the approach), and **execution planning** (determining the sequence of actions)[8].

This architecture enables dynamic adaptability. Unlike hardcoded decision trees, LLM agents can handle novel situations by reasoning through them using their understanding of their role and capabilities.

Planning and Reasoning Systems

ADK provides multiple planning architectures to enhance agent reasoning capabilities[3]. The **BuiltInPlanner** leverages advanced model capabilities like Gemini's thinking feature, where the model can explicitly reason through problems before providing answers[3]. This approach is particularly powerful for complex problem-solving scenarios.

The **PlanReActPlanner** implements a structured plan-reason-act cycle for models that don't have built-in thinking capabilities[3]. This planner creates explicit reasoning steps, allowing agents to break down complex tasks into manageable components.

The planning architecture is crucial for handling multi-step tasks where the agent needs to coordinate multiple tool calls or sub-agent interactions. The planner helps ensure that the agent maintains coherence across complex workflows.

Tool Integration Architecture

Tools extend LLM agents beyond their core language capabilities[8] [9]. The technical architecture for tool integration is sophisticated, involving several layers: **tool discovery** (the agent understanding what tools are available), **tool selection** (choosing the right tool for the task), **parameter generation** (creating appropriate inputs for the tool), **execution coordination** (managing the tool call), and **result integration** (incorporating tool outputs into ongoing reasoning)[8].

The LLM agent uses tool docstrings, function signatures, and type hints to understand when and how to use each tool[8]. This metadata-driven approach enables agents to work with tools they've never seen before, as long as the tools are well-documented.

Workflow Agents: Deterministic Orchestration

Workflow Agents represent a fundamentally different architectural approach focused on **deterministic process control**[10]. Unlike LLM agents that use reasoning to make decisions, workflow agents follow predefined execution patterns[10].

Sequential Agents: Linear Processing Architecture

`SequentialAgent` implements a **pipeline architecture** where sub-agents execute in a predetermined order[10]. This pattern is essential for workflows where each step depends on the outputs of previous steps. The sequential architecture ensures that data flows correctly through the processing chain and that dependencies are respected.

Sequential agents are particularly valuable for **multi-stage processing** scenarios like content creation (research → outline → writing → editing) or data processing (collection → validation → transformation → storage). The deterministic nature ensures reproducible results and predictable behavior.

Parallel Agents: Concurrent Processing Architecture

`ParallelAgent` enables **concurrent execution** of multiple sub-agents[10]. This architecture is valuable when multiple independent tasks need to be completed and the results can be processed simultaneously.

The parallel execution architecture requires careful consideration of **resource management** and **result aggregation**. ADK handles the complexity of coordinating multiple concurrent operations while providing a simple interface for developers.

Loop Agents: Iterative Processing Architecture

`LoopAgent` implements **iterative processing** patterns where sub-agents execute repeatedly until a termination condition is met[10]. This architecture is essential for scenarios like **data collection** (continue until sufficient data is gathered), **optimization** (iterate until convergence), or **monitoring** (repeat checks until condition changes).

The loop architecture requires careful **termination condition design** to prevent infinite loops while ensuring that the desired outcome is achieved.

Custom Agents: Specialized Implementation Architecture

Custom Agents extending `BaseAgent` directly enable **specialized implementations** that don't fit the standard patterns[6]. This architectural flexibility is crucial for integrating with existing systems, implementing specialized protocols, or creating agents with unique behavioral patterns.

Custom agents must implement the `_run_async_impl` method, which defines how the agent processes requests[6]. This low-level interface provides complete control over agent behavior while still benefiting from ADK's infrastructure for state management, event generation, and session handling.

State Management and Session Architecture

Session Conceptual Architecture

The **Session** concept in ADK represents a **conversation thread** between users and agents[11] [12]. Sessions are architectural containers that maintain context across multiple interactions, enabling agents to remember previous conversations and build upon past interactions[12].

Each session contains several architectural components: **identification metadata** (session ID, app name, user ID), **event chronology** (complete history of interactions), **state storage** (key-value data relevant to the conversation), and **metadata** (timestamps, version information)[12].

The session architecture enables **conversation continuity** where agents can refer to previous interactions, **context accumulation** where information builds over time, **user personalization** where preferences are remembered across sessions, and **multi-agent coordination** where different agents can share context within the same session[11].

Session Lifecycle Management

The session lifecycle involves several architectural phases: **creation** (establishing a new conversation context), **activation** (loading existing session data), **interaction processing** (updating session with new events and state changes), **persistence** (saving session changes), and **termination** (cleaning up session resources)[13].

Session creation can occur explicitly (when starting a new conversation) or implicitly (when no existing session is available). The lifecycle management ensures that session data is consistent and that resources are properly managed.

State Architecture and Management

State within sessions serves as a **persistent key-value store** that agents can read from and write to[11] [14]. The state architecture enables agents to maintain information across interactions without requiring external storage systems.

State Scoping and Prefixes

ADK implements a **hierarchical state scoping system** using prefixes[15] [14]. This architectural choice enables different scopes of data persistence:

- **Session-scoped state** (no prefix): Data specific to the current conversation
- **User-scoped state** (`user:` prefix): Data that persists across all sessions for a specific user
- **Application-scoped state** (`app:` prefix): Data shared across all users of an application
- **Temporary state** (`temp:` prefix): Data that exists only during the current interaction

This scoping architecture enables **granular data management** where different types of information have appropriate persistence levels. User preferences might be stored with `user:` scope while conversation-specific details use session scope.

State Mutation and Consistency

State modifications in ADK follow an **atomic update pattern** where changes are tracked as deltas and committed as part of event processing[14]. This architecture ensures that state changes are consistent with the events that caused them and that partial updates don't create inconsistent states.

The state management system is designed to be **serializable**, meaning that only JSON-compatible data types can be stored[14]. This constraint ensures that state can be persisted to various storage backends without compatibility issues.

Memory vs. State: Architectural Distinction

The distinction between **Memory** and **State** in ADK reflects different architectural concerns[11]. State is **session-centric** and focused on immediate conversation context, while Memory is **user-centric** and focused on long-term information retention across sessions.

Memory services provide **semantic search** capabilities, allowing agents to find relevant information from past interactions based on content similarity rather than exact key matches[11]. This capability enables agents to leverage historical context even when they don't know exactly what they're looking for.

The memory architecture supports **cross-session learning** where agents can benefit from patterns and information discovered in previous conversations. This capability is essential for building agents that improve over time and can provide increasingly personalized experiences.

Tools and Capability Architecture

Tool Conceptual Framework

Tools in ADK represent **capability extensions** that allow agents to interact with the world beyond their core language model capabilities[8]. The tool architecture is based on a **function-centric model** where tools are essentially functions that can be called by agents with specific parameters and return structured results[8].

Tool Decision Architecture

The process by which agents decide to use tools involves sophisticated **metadata analysis**[8]. Agents examine tool **function names**, **parameter specifications**, **type hints**, and critically, **docstrings** to understand what each tool does and when it should be used[8].

The docstring serves as the **tool's contract** with the agent, explaining not just what the tool does, but when it should be used, what information is needed, and what the expected outputs are[8]. Well-written docstrings are essential for effective tool usage because they enable the LLM to make informed decisions about tool selection.

Function Tool Architecture

Function Tools represent the most common tool pattern in ADK[8]. These are standard Python functions or Java methods that are automatically wrapped with ADK's tool infrastructure. The wrapper handles **parameter validation**, **type conversion**, **error handling**, and **result formatting**.

Function tools must return dictionary/Map objects to ensure consistent interfaces[8]. If a function returns other types, ADK automatically wraps the result in a `{'result': value}` structure. This architectural choice ensures that all tool outputs have a consistent format that agents can reliably process.

The function tool architecture supports **automatic schema generation** where ADK analyzes function signatures and type hints to create the schemas that LLMs use for function calling[8]. This automation reduces the development overhead while ensuring that schemas accurately reflect the function interface.

ToolContext: Advanced Capability Architecture

ToolContext provides tools with **rich contextual access** to the agent's execution environment[8]. This architecture enables tools to be more than simple functions - they become **context-aware capabilities** that can interact with the broader agent system.

State Management Through ToolContext

Tools can **read and modify session state** through ToolContext, enabling them to maintain information across tool calls and coordinate with other parts of the agent system[8]. State modifications through tools are **tracked and persisted** just like state changes from other sources.

The state access capability transforms tools from stateless functions into **stateful components** that can build up information over time. This is essential for tools that need to maintain context across multiple invocations or coordinate with other tools.

Flow Control Through ToolContext

ToolContext provides **execution flow control** capabilities through the `actions` interface[8]. Tools can **skip LLM summarization** when their output is already user-ready, **transfer control to other agents** for specialized handling, or **escalate to parent agents** when they cannot complete a task.

This flow control architecture enables tools to be **active participants** in agent orchestration rather than passive capabilities. Tools can make decisions about how the conversation should proceed based on their specific domain knowledge.

Service Integration Architecture

ToolContext provides **service integration** capabilities for accessing persistent data through **artifact services** and **memory services**[8]. This enables tools to **load previous files**, **save new artifacts**, and **search historical information** to enhance their capabilities.

The service integration architecture enables tools to maintain **persistent context** beyond the current conversation, creating opportunities for more sophisticated tool behaviors that leverage historical information.

Built-in Tools and Third-Party Integration

ADK provides **built-in tools** for common capabilities like **Google Search**, **Code Execution**, and **document processing**[8]. These tools are architecturally significant because they demonstrate best practices for tool design and provide immediately useful capabilities.

The **third-party integration** architecture supports tools from **LangChain**, **CrewAI**, and other frameworks[8]. This integration capability reflects ADK's philosophy of **ecosystem compatibility** rather than vendor lock-in.

Multi-Agent System Architecture

Hierarchical Agent Organization

ADK's multi-agent architecture supports **hierarchical organization** where agents can have **parent-child relationships**[16] [17]. This organizational pattern enables **specialization** where different agents handle different aspects of complex problems while maintaining coordination through the hierarchy.

The hierarchical architecture supports **delegation patterns** where higher-level agents analyze problems and delegate specific tasks to specialized sub-agents[17]. This approach enables **cognitive division of labor** where each agent can focus on what it does best.

Agent Transfer Mechanisms

Agent transfer represents a sophisticated orchestration capability where agents can **dynamically hand off control** to other agents based on the conversation context[16]. This capability enables **adaptive routing** where the most appropriate agent handles each part of a conversation.

Transfer mechanisms can be **LLM-driven** (where the agent reasons about when to transfer), **tool-driven** (where tools trigger transfers based on specific conditions), or **explicit** (where transfers are programmatically controlled)[16].

Controller and Specialist Patterns

The **Controller Pattern** implements a **hub-and-spoke architecture** where a central agent coordinates multiple specialized agents[16]. This pattern is valuable for scenarios where a single request might require multiple types of expertise.

The **Hierarchical Specialist Pattern** creates **nested specialization** with **escalation paths** where simple issues are handled by general agents and complex issues are escalated to specialists[16]. This pattern optimizes for efficiency while ensuring that complex cases get appropriate attention.

Multi-Agent Coordination Protocols

Multi-agent systems in ADK coordinate through several mechanisms: **shared state** (common information accessible to all agents), **event streams** (communication through event publication and subscription), **agent transfers** (explicit handoffs of conversation control), and **hierarchical messaging** (parent-child communication patterns)[16].

The coordination architecture ensures that multiple agents can work together coherently while maintaining their individual specializations and avoiding conflicts in their operations.

Callback Architecture and Lifecycle Management

Callback Architectural Philosophy

Callbacks in ADK represent a **hook-based architecture** that allows developers to **intercept and modify** agent behavior at specific execution points[18]. This architecture provides **fine-grained control** over agent operations without requiring modifications to the core framework[18].

The callback architecture follows an **aspect-oriented programming** pattern where cross-cutting concerns like **logging**, **security**, **monitoring**, and **customization** can be implemented as separate modules that integrate with the main execution flow[18].

Callback Categories and Technical Implementation

Agent-Level Callbacks

Before Agent and **After Agent** callbacks operate at the **highest level** of agent execution, wrapping the entire request processing cycle[18]. These callbacks are ideal for **request validation**, **authentication**, **comprehensive logging**, and **response modification**.

The before agent callback can **bypass the entire agent execution** by returning a Content object, enabling scenarios like **cached responses**, **access control**, or **request routing** without involving the main agent logic[18].

Model-Level Callbacks

Before Model and **After Model** callbacks provide **LLM interaction control**[18]. The before model callback can **modify requests** before they're sent to the LLM, implement **input validation**, or **serve cached responses**. The after model callback can **sanitize outputs**, **add metadata**, or **modify response structure**.

These callbacks are particularly valuable for implementing **guardrails**, **content filtering**, **prompt optimization**, and **response standardization**[18].

Tool-Level Callbacks

Before Tool and **After Tool** callbacks enable **tool execution control**[18]. Before tool callbacks can **validate parameters**, **enforce permissions**, **provide cached results**, or **block unauthorized operations**. After tool callbacks can **sanitize outputs**, **add metadata**, or **standardize response formats**.

Tool callbacks are essential for implementing **security policies**, **audit logging**, **error handling**, and **result standardization** across different tools[18].

CallbackContext Architecture

CallbackContext provides callbacks with **rich contextual information** including **agent identification**, **session state access**, **event metadata**, and **service integration** capabilities[18]. This architecture enables callbacks to make **informed decisions** and **coordinate with other system components**.

The context architecture ensures that callbacks have access to the same information and capabilities that agents have, enabling sophisticated callback implementations that can fully participate in the agent ecosystem.

Runner System and Execution Architecture

Runner Conceptual Role

The **Runner** serves as the **execution engine** for ADK agents, managing the **event loop**, **session coordination**, **state persistence**, and **service integration**[5] [19]. The Runner architecture provides **separation of concerns** between agent logic and infrastructure management.

Session Management Architecture

Runners coordinate with **SessionService** implementations to manage conversation persistence[19]. The Runner ensures that **session state is loaded** before agent execution, **state changes are persisted** after execution, and **event chronology is maintained** throughout the conversation lifecycle.

The session management architecture supports **different persistence strategies** from in-memory storage for development to database persistence for production deployments[13] [20].

Event Streaming Architecture

Runners provide **real-time event streaming** that enables **live monitoring** of agent operations[19]. Events are generated for **user messages**, **agent responses**, **tool calls**, **state changes**, and **agent transfers**, providing comprehensive visibility into system behavior.

The streaming architecture enables **reactive user interfaces**, **real-time monitoring systems**, and **integration with external systems** that need to respond to agent activities in real-time[19].

Runner Implementation Types

InMemoryRunner

The **InMemoryRunner** provides a **lightweight execution environment** ideal for development, testing, and scenarios where persistence isn't required[21]. This implementation keeps all session data in memory, providing fast access but without durability across application restarts.

Production Runners

Production Runner implementations integrate with **persistent storage systems**, **cloud services**, and **enterprise infrastructure**[19]. These implementations provide **scalability**, **durability**, and **high availability** required for production agent deployments.

Model Integration Architecture

Model Abstraction Layer

ADK's **model abstraction** enables **vendor-agnostic agent development** where the same agent logic can work with different LLM providers[2] [4]. This architecture provides **strategic flexibility** and **cost optimization** opportunities.

Direct Google Integration

Direct integration with Google's Gemini models provides **optimal performance** and **access to advanced features** like **native thinking**, **multimodal capabilities**, and **structured outputs**[2].

LiteLLM Integration Architecture

LiteLLM integration enables ADK agents to work with **OpenAI**, **Anthropic**, **Mistral**, and other model providers through a **unified interface**[2] [4]. This integration architecture transforms vendor-specific APIs into consistent interfaces that ADK agents can use interchangeably.

Model Configuration and Optimization

Model configuration in ADK supports **fine-tuned control** over generation parameters including **temperature**, **token limits**, **sampling strategies**, and **stop sequences**[4]. These configuration options enable **optimization** for different use cases and **cost management** through appropriate parameter tuning.

Deployment and Production Architecture

Local Development Architecture

ADK provides **comprehensive development tooling** including a **CLI interface** and **web-based development environment**[1]. The development architecture enables **rapid prototyping**, **interactive testing**, and **debugging** without requiring production infrastructure.

Production Deployment Strategies

Containerization Architecture

ADK agents can be **containerized** for deployment in **cloud environments**, **Kubernetes clusters**, or **hybrid infrastructure**[1]. The containerization approach provides **consistency** across development and production environments while enabling **scalable deployment** patterns.

Cloud Integration Architecture

Vertex AI Agent Engine integration provides **managed scaling**, **enterprise security**, and **integrated monitoring** for ADK agents in Google Cloud environments[1]. This integration enables **production-ready deployment** with minimal infrastructure management overhead.

Integration Architecture

ADK's **Agent2Agent protocol** enables **cross-platform integration** where ADK agents can **communicate with agents** built using other frameworks[4]. This protocol ensures that ADK agents can **participate in heterogeneous agent ecosystems** rather than requiring complete framework migration.

Security and Safety Architecture

Input Validation Architecture

ADK provides **callback-based input validation** where **before agent callbacks** can **inspect and sanitize** user inputs before processing[18]. This architecture enables **security policy enforcement**, **content filtering**, and **attack prevention** at the system level.

Output Filtering and Control

After agent callbacks and **after model callbacks** provide **output control** capabilities for **content filtering**, **information sanitization**, and **compliance enforcement**[18]. This architecture ensures that agent outputs meet **organizational policies** and **regulatory requirements**.

Access Control Integration

ADK's **state scoping** and **callback architecture** enable **role-based access control** where different users have access to different capabilities and information[15] [18]. The access control architecture integrates with **enterprise identity systems** and **policy engines**.

Advanced Patterns and Integration Strategies

Evaluation and Testing Architecture

ADK provides **built-in evaluation frameworks** for **systematic testing** of agent behavior[1]. The evaluation architecture supports **test case definition**, **automated execution**, **performance measurement**, and **regression detection**.

Memory and Learning Architecture

ADK's **memory services** enable **cross-session learning** where agents can **leverage historical information** to provide **improved responses** over time[11]. The memory architecture supports **semantic search**, **context retrieval**, and **personalization**.

Monitoring and Observability

The **event-driven architecture** provides **comprehensive observability** through **event streaming**, **callback hooks**, and **integration points** for **monitoring systems**[5] [18]. This observability enables **performance optimization**, **behavior analysis**, and **system debugging**.

Enterprise Integration Patterns

ADK supports **enterprise integration** through **standardized protocols**, **security frameworks**, **audit capabilities**, and **scalability patterns** that enable **production deployment** in **regulated environments**[1].

This comprehensive technical architecture guide demonstrates how ADK's design philosophy of treating agent development as software development manifests in sophisticated, production-ready architectural patterns that enable the construction of reliable, scalable, and maintainable agent systems.

⁂

1. https://google.github.io/adk-docs/
2. https://www.siddharthbharath.com/the-complete-guide-to-googles-agent-development-kit-adk/
3. https://google.github.io/adk-docs/agents/llm-agents/
4. https://www.datacamp.com/tutorial/agent-development-kit-adk
5. https://weave-docs.wandb.ai/guides/integrations/google_adk/
6. https://google.github.io/adk-docs/tools/
7. https://cs.stackexchange.com/questions/165719/what-is-the-precise-difference-between-state-and-environment-and-memory

8. https://glaforge.dev/posts/2025/06/15/expanding-ai-agent-capabilities-with-tools/
9. https://www.firecrawl.dev/blog/google-adk-multi-agent-tutorial
10. https://saptak.in/writing/2025/05/10/google-adk-masterclass-part5
11. https://dev.to/masahide/smarter-adk-prompts-inject-state-and-artifact-data-dynamically-placeholders-2dcm
12. https://www.youtube.com/watch?v=z8Q3qLi9m78
13. https://askubuntu.com/questions/9733/what-is-the-difference-between-the-memory-usage-report-in-system-monitor-and-the
14. https://cloud.google.com/vertex-ai/generative-ai/docs/agent-engine/sessions/manage-sessions-adk
15. https://cs.stackexchange.com/questions/136131/what-is-the-difference-between-a-state-machine-and-a-database
16. https://cloud.google.com/blog/products/ai-machine-learning/build-kyc-agentic-workflows-with-googles-adk
17. https://docs.gitlab.com/runner/
18. https://saptak.in/writing/2025/05/10/google-adk-masterclass-part10
19. https://google.github.io/adk-docs/get-started/about/
20. https://google.github.io/adk-docs/sessions/state/
21. https://saptak.in/writing/2025/05/10/google-adk-masterclass-part8