Google - AI Agents Intensive Course

(Agent Development Kit)

🌟 DAY 1 – AI AGENT ARCHITECTURE

Complete Notes (from video : Whitepaper Companion Podcast - Introduction to Agents)
Kaggle Guide : (Day 1a - From Prompt to Action)

1. Shift in AI: From Passive Models → Autonomous Agents

Old AI (Passive LLMs)

- Only respond to prompts.
- No planning, no action, no autonomy.
- Examples: answering questions, translating text.

New AI (Agents)

- Goal-oriented and autonomous.
- Can plan, act, observe, and iterate.
- Perform multi-step tasks without constant human guidance.

2. Anatomy of an AI Agent (3 Core Components)

1. Model (The Brain)

- LLM is the reasoning engine.
- Manages the **context window**:
    - Decides what information is important right now.
    - Uses mission details, memory, and tool outputs.

2. Tools (The Hands)

- Allow the agent to take action.
- Can be:
    - APIs
    - Database access
    - Vector search / RAG
    - Code execution (Python sandbox)
- Model decides **which tool** is needed for a step.

3. Orchestration Layer (The Conductor)

- Runs the full agent loop: think → act → observe.

- Maintains state, memory, and reasoning strategy.
- Executes the tools the model selects.
- Defines agent rules and persona (system prompt).

3. The Core Agent Loop (Think → Act → Observe → Repeat)

Steps

1. **Mission** – Agent receives goal (e.g., plan travel).
2. **Scan** – Check available tools + memory.
3. **Think / Plan** – Decide the next step.
4. **Act** – Use the appropriate tool.
5. **Observe** – Add output to context and plan again.

This cycle repeats until the mission is completed.

4. Levels of AI Agents (Capability Taxonomy)

Level 0 – Plain LLM

- No tools.
- Static knowledge only.
- Cannot access real-time information.

Level 1 – Connected Problem Solver

- LLM + Tools.
- Can search the web, access databases, call APIs.
- Gains real-time awareness.

Level 2 – Strategic Problem Solver

- Handles **multi-step tasks**.
- Does **context engineering** → uses output of one step to craft precise input for the next.
- Example: finding a coffee shop halfway between two addresses.

Level 3 – Multi-Agent Collaboration

- Agents call **other agents** as tools.
- Example:
    - Project manager agent delegates tasks to data agent, research agent, etc.
- Goal delegation, not single-function calls.

Level 4 – Self-Evolving System

- Agent identifies its own capability gaps.
- Can **create new agents or tools** dynamically.
- Adapts and expands autonomously.

5. Building Reliable Production Agents

Model Selection

- Not based on benchmarks alone.
- Based on **reasoning reliability** and **tool usage accuracy**.
- Use **model routing**:
    - Heavy reasoning → large model (e.g., Gemini Pro).
    - Simple tasks → smaller model (Gemini Flash).

6. Tools: Retrieval + Action

Retrieval

- For grounding in facts.
- Uses:
    - RAG (vector DB + embeddings)
    - NL2SQL for querying structured databases.

Action

- APIs:
    - Calendar, CRM, emails, etc.
- Executing code:
    - Python sandbox for dynamic tasks.

Function Calling

- Tools need clear definitions (OpenAPI style).
- Allows model to:
    - Form correct API call.
    - Understand tool outputs.

7. Memory System

Short-Term Memory

- Working memory of current task.
- Stores step-by-step action/observation history.

Long-Term Memory

- Persists across sessions.
- Stores:
    - Preferences
    - Past interactions

    - Learned knowledge
- Typically implemented through vector DB + RAG.

8. AgentOps: Testing, Debugging, Monitoring

Why it's hard

- LLM outputs are non-deterministic.
- Can't rely on fixed expected outputs.

Evaluation

- Use **an LLM as a judge** with a rubric.
- Tests:
    - Factuality
    - Constraint adherence
    - Goal completion

Debugging

- Use **OpenTelemetry traces**:
    - Shows prompts
    - Reasoning
    - Tool calls + parameters
    - Observations
- Works like a “black box recorder”.

User Feedback Loop

- Every failure becomes a new test case.
- Builds a “golden dataset” of scenarios.
- Makes agent stronger over time.

9. Security & Scaling

Trust Trade-Off

- More tools = more power = more risk.

Defense in Depth

1. **Hard-coded guardrails**
    - Example: limit spending via rules.

2. **AI Guard Models**
    - Detect risky actions (data leakage, forbidden steps).

Agent Identity

- Every agent must have:
    - Own identity
    - Permissions

- Enables **least-privilege access**:
    - Sales agent → CRM only
    - No access to HR database

Managing Agent Sprawl

- Large systems may have hundreds of agents.
- Need:
    - Agent governance
    - Central control plane/gateway
    - Permission management

SUMMARY

Agents = LLM (brain) + Tools (hands) + Orchestration (conductor).
They follow the loop: **Think** → **Act** → **Observe** → **Repeat**.

Levels:

- **0:** LLM only
- **1:** Tools
- **2:** Multi-step tasks
- **3:** Multi-agent teams
- **4:** Self-improving systems

Production agents need:

- Model routing
- Clear tool definitions
- Memory system (short + long term)
- LLM-as-judge testing
- OpenTelemetry traces for debugging
- Strong security & agent identity management

Coding Day 1 Asssignment: AI Agent – Environment Setup & Execution Notes

(Based on a VS Code + ADK local implementation)

1. Environment Setup

1.1 Required Installations

1. **Python 3.10+**
    - Required to run the Agent Development Kit.

2. **pip**
    - Used to install Python packages.

Google ADK (Agent Development Kit)

```
pip install google-adk
```

3. **Definition:**
   A framework provided by Google for creating, running, and debugging AI agents capable of tool use and multi-step reasoning.

4. **Google Generative AI library (optional dependency installed automatically)**
   Definition:
   Provides access to Gemini models and handles LLM communication.

5. **dotenv package (auto-used by ADK)**
    - Allows reading environment variables from `.env` files (e.g., API keys).

2. File Structure Used in the Setup

2.1 `.env` File

Purpose:

- Stores sensitive credentials such as API keys.
- Automatically loaded by ADK during execution.

Contents:

```
GOOGLE_API_KEY="AIzaSyCCAQMTz8GR2NJrf1eaZWW1sPK164yqNf0"
```

2.2 `agent.py` File

Purpose:

- Contains the actual agent definition, tool definition, and configuration.

Full code used:

```
from google.adk.agents.llm_agent import Agent

# Mock tool implementation
def get_current_time(city: str) -> dict:
    """Returns the current time in a specified city."""
    return {"status": "success", "city": city, "time": "10:30 AM"}

root_agent = Agent(
    model='gemini-2.5-flash',
    name='root_agent',
    description="Tells the current time in a specified city.",
    instruction="You are a helpful assistant that tells the current time
in cities. Use the 'get_current_time' tool for this purpose.",
    tools=[get_current_time],
)

#to run in terminal : adk run my_agent
#to run in web: adk web --port 8000
```

2.3 `__init__.py` File

Purpose:

- Marks the folder as a Python package.
- Usually empty.
- Required for ADK project structure.

Contents:

```
# (empty file)
```

3. Opening VS Code Correctly

Important:

To run ADK commands successfully in Windows, VS Code must be opened using:

Right-click VS Code → Run as Administrator

Reason:

- ADK needs permissions to start local servers and bind to ports (for web UI).

4. Running the Agent in Terminal (CLI)

Definition: CLI (Command Line Interface)

A text-based interface for running commands through the terminal.

Commands Used

4.1 Running the Agent (CLI Mode)

```
adk run my_agent
```

Purpose:

- Starts the agent in interactive terminal mode.
- CLI asks for user input and the agent responds using the specified model + tool.

Output behavior:

- When a question like “What is the time in Mumbai?” is typed,
  the agent calls the `get_current_time` tool and returns the mock time.

Output Image:

```
[user]: what can you do?
[root_agent]: I can tell you the current time in a specified city.
[user]: what time it is in London?
[root_agent]: The current time in London is 10:30 AM.
[user]:
```

5. Running the Agent in Web Interface

Command

```
adk web --port 8000
```

Purpose:

- Launches the ADK Web UI (browser-based interface).
- Enables chatting with the agent visually instead of using the terminal.

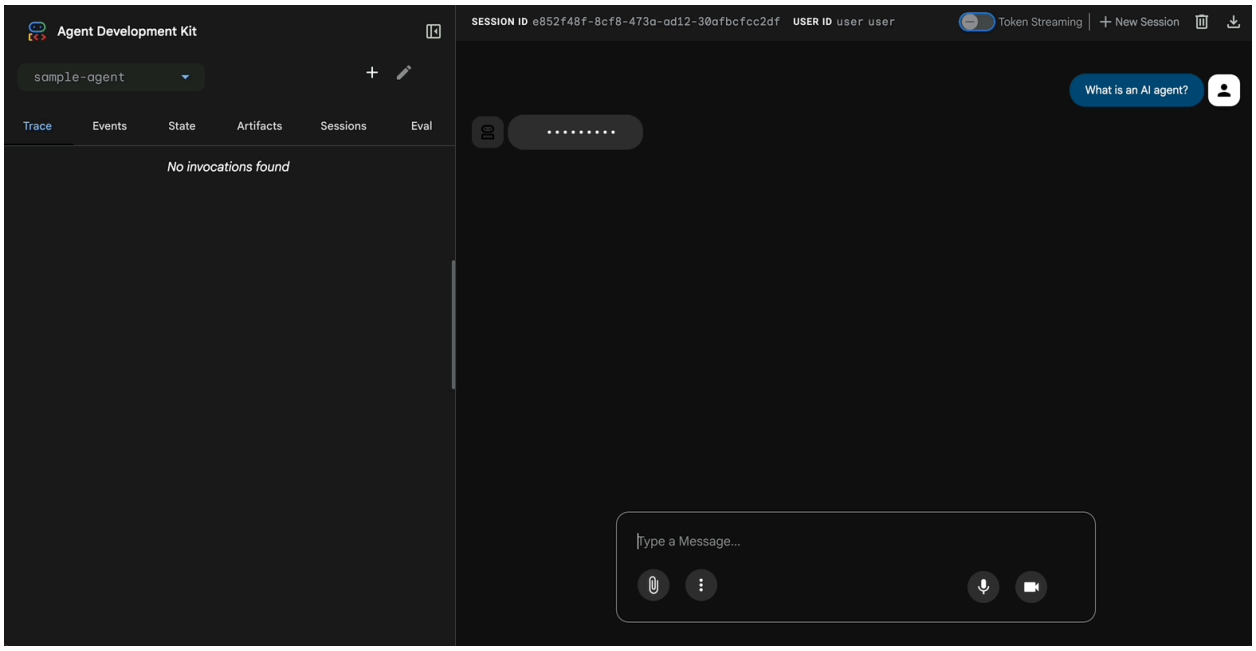
Definition:

- **ADK Web UI:** A debugging and testing interface automatically provided by ADK.

How to access:

- Browser → `http://127.0.0.1:8000`

`Output Video:`



6. Summary (Short Revision)

Steps to Set Up

1. Install Python + pip.
2. Install ADK:
   ```
   pip install google-adk
   ```

3. Create project folder.
4. Add required files:
    - `.env` (contains GOOGLE_API_KEY)
    - `agent.py` (main agent code)
    - `__init__.py` (empty)
5. Open VS Code **as Administrator**.
6. Run agent through terminal:
    - `adk run my_agent`
7. Run agent through web UI:
    - `adk web --port 8000`

Definitions (Short)

Term	Meaning
ADK (Agent Development Kit)	Framework for building tool-using AI agents.
Agent	An LLM configured with tools, instructions, and behaviors.
Tool	A function the agent can call to take action (e.g., time lookup).
CLI	Command Line Interface for running agents in terminal.
Web UI	Browser interface for interacting with and debugging agents.
.env	Stores API keys securely.
__init__.py	Marks a folder as a Python package.

🌟 DAY 2 NOTES — PART 1

(From video : Whitepaper Companion Podcast - Agent Tools & Interoperability with MCP)

1. The Core Problem of AI Today

Large language models (LLMs) are brilliant — but fundamentally limited.
They cannot:

- Access real-time information
- Call APIs
- Perform actions in the real world
- Check live data (stocks, weather, emails, files, etc.)

LLMs are **pattern-matching brains trapped inside their training data**.
They can "think" but **cannot "act."**

To make them useful in real systems, you need **Agentic AI**:
LLMs that can reason, take actions, observe results, and continue reasoning.

2. The Key Breakthrough: Tools

In agentic AI, **tools are the agent's eyes and hands.**
They provide:

- Access to real-time information
- Ability to take actions
- Ability to interact with systems or other agents

Tools solve the "LLMs cannot act" problem.

But historically, connecting tools to models had a huge issue:

The n × m Integration Nightmare

If you had:

- **n models**
- **m tools/APIs**

You had to build **n × m custom integrations**, leading to massive complexity and unscalable systems.

3. The Modern Industry Solution: MCP (Model Context Protocol)

Introduced around 2024, MCP is an **open standard** that removes the integration mess.

With MCP:

- Agents can use tools in a standardized way
- Tools become easily plug-and-play
- You decouple the **reasoning brain** from the **acting tool**
- Multi-agent systems become consistent and maintainable

MCP provides the **language** for agents to talk to tools.

4. What is a Tool? (Deep Explanation)

A “tool” is anything the LLM uses to do what it cannot do natively.

There are **two main purposes** of tools:

A) Know something

Fetch new information (e.g., weather API, stock API, search API)

B) Do something

Take actions (send email, book meeting room, modify a file, query a database)

—

The transcript identifies **three types** of tools:

TYPE 1: Function Tools

- Developer writes Python/JS functions
- LLM can call them
- You provide **detailed docstrings** that act as the “instruction manual”
- The LLM reads the docstring to understand *how* and *when* to call the function

Example:
`set_light(brightness, color)`
Docstring explains inputs/outputs clearly.

These are the most common.

TYPE 2: Built-In Tools

These are baked into the model provider.

Example:

- Google Search grounding
- Code execution tools
- URL fetchers

Developers don't create these; the platform exposes them automatically.

TYPE 3: Agent Tools

A **sub-agent** is treated as a tool.

Your main agent remains in charge and delegates tasks.

Example:
Main agent → calls DataAnalysisAgent → receives summary → then sends email using another tool.

This creates **hierarchical multi-agent workflows**.

5. Tool Taxonomy (What Tools Are Used For)

Four broad categories:

1. **Information Retrieval**
   Getting knowledge, data, or context (weather, stock, DB query)
2. **Action Execution**
   Doing things (send email, write file, update record)
3. **System Integration**
   Connecting to software systems (CRM, calendars, APIs)
4. **Human-in-the-loop Tools**
   Agent stops and asks for user approval when needed.

6. Best Practices for Designing Tools (Very Important)

These are CRITICAL for building real agents.

Rule 1: Documentation Must Be Excellent

LLMs rely entirely on:

- Tool name
- Tool description
- Parameter names
- Parameter descriptions

If these are unclear, the model will fail or misuse the tool.

Rule 2: Describe the Action, Not How to Call the Tool

Bad:
"In this tool, call the calendar.create_event API with params X,Y,Z."

Good:
"Creates a calendar event for the user."

LLMs should decide *when* to use the tool.
Tools simply **execute**.

Rule 3: Publish High-Level Tasks, Not Complex API Calls

You should not expose raw enterprise APIs with 20+ parameters.

Instead:

Bad tool:
`call_calendar_api_with_all_params(...)`

Good tool:
`book_meeting_room(time, room_name)`

Tools must be **task-focused**.

Rule 4: Keep Tool Output Short and Clean

NEVER return:

- Large spreadsheets
- Logs
- Full documents
- Big JSON payloads

This causes **context window bloat**, destroys performance, and increases cost.

Better:

- Return a summary
- Or return a **URI link** to big data (the agent can access it later)

Rule 5: Good Error Messaging

Tools will fail.
Errors must be:

- Descriptive
- Instructive
- Helpful

Example:
"API rate limit exceeded. Retry after 15 seconds."

This helps the agent self-correct.

7. Deep Dive: MCP Architecture

Three components:

1) MCP Host

- Core application
- Orchestrates reasoning
- Applies safety & authorization
- Decides when tools are needed

2) MCP Client

- Runs inside the host
- Manages the connection
- Sends commands
- Maintains sessions

3) MCP Server

- Provides the tools
- Executes commands
- Returns results

This separation is extremely important — makes systems modular

8. Communication Layers

➔ Protocol Layer

Uses **JSON-RPC 2.0**

➔ Transport Layer

Two options:

- **stdIO** for local fast connections
- **HTTP/SSE** for distributed scalable systems

9. Scaling Problem: Too Many Tools = Context Explosion

If an agent has 1,000 tools, loading all tool definitions is impossible.
The LLM's reasoning collapses.

10. The Solution: Tool Retrieval (RAG-Style Tool Selection)

Before loading any tools into context:

1. Agent performs a **semantic search**
2. Retrieves the **top few relevant tools** (3–5)
3. Loads only those tool schemas into the LLM
4. Executes the action

This makes large tool ecosystems scalable.

11. Security Challenges (Major Section)

MCP lacks built-in security.

The biggest risk:

Confused Deputy Problem

User → tricks model → model tells tool → tool performs privileged action.

The tool trusts the agent too much.

Solution

Wrap MCP servers with:

- API gateways
- Authentication
- Role-based authorization
- Rate limiting
- Logging
- Input filtering

Security must exist **around** MCP, not inside it.

12. Final Takeaway

LLMs = brains
Tools = eyes + hands
MCP = universal language between brains and hands

Disciplined tool design + external governance = reliable, scalable agents.

DAY 2 NOTES — PART 2

BASED ONLY ON THE PDF (SHORT, CLEAN, TECHNICAL SUMMARY)

Agent Tools & Interoperability with Model Context Protocol (MCP).pdf - Google Drive

These notes are intentionally short because the PDF is compact and focused.

1. MCP Overview

The Model Context Protocol (MCP) is a **standard** for integrating agents with external tools, resources, and prompts.
It enables **interoperability**, **reusability**, and consistent communication.

2. MCP Architecture

Three components:

- **Host** (orchestrates reasoning and safety)
- **Client** (embedded connector inside the host)
- **Server** (exposes tools)

3. Communication Layers

- JSON-RPC 2.0 for structured messages
- stdIO or HTTP(S) for transport

4. MCP Primitives

- **Tools**
- **Resources**
- **Prompts**

Tools are the most widely adopted primitive.

5. Tool Schema Requirements

Every tool must define:

- name
- description
- input schema
- optional output schema

6. Tool Execution Results

Two types:

1. **Structured results** (JSON matching output schema)
2. **Unstructured results** (text, images, binary, URI references)

7. Errors

Two error categories:

- Protocol errors (invalid method, malformed parameters)
- Execution errors (`"is_error": true`)

8. Scaling Challenge

Large number of tools → context overload
Agents cannot load all tool definitions at once.

9. Tool Retrieval (RAG-Style Solution)

The PDF suggests using a **Retrieval-Augmented Generation (RAG)** approach for tools:

- Maintain an index of tool descriptions
- Perform a semantic search
- Load only top relevant tools into context

📌 RAG — Definition

RAG (Retrieval-Augmented Generation)
is an AI technique where the model **retrieves the most relevant information** from an external database before answering a query.

In context of MCP:

- Agent retrieves relevant tool definitions
- Only loads those few into the LLM
- Avoids context overload

10. Security Considerations

MCP itself **does not** handle authentication or authorization.
The primary security risk is the **confused deputy problem**.
Enterprises must wrap MCP with:

- API gateways
- Authentication
- Authorization
- Logging

11. Benefits

- Faster development
- Ecosystem interoperability
- Plug-and-play tool discovery
- Cleaner architecture

DAY-2: Code Implementation Notes (ADK – Custom Tools & Agent Tools)

1. Project Setup

- Create a new folder named **day2_agent**.
- Run the following command inside the main course folder:
  `adk create day2_agent`
- The command generates:
    - **day2_agent/agent.py**
    - **day2_agent/.env**
    - **day2_agent/init.py** (empty file used to mark a Python package)

2. Required File Updates

2.1 Update `.env`

Insert the Gemini API key:

```
GOOGLE_API_KEY="YOUR_API_KEY"
```

3. Code Used in `agent.py`

Replace the contents of **agent.py** with the following:

```
from google.genai import types
from google.adk.agents import LlmAgent
from google.adk.models.google_llm import Gemini
from google.adk.tools import AgentTool
from google.adk.code_executors import BuiltInCodeExecutor

# --- Configuration (from cell 1.5) ---
retry_config = types.HttpRetryOptions(
    attempts=5,
    exp_base=7,
    initial_delay=1,
    http_status_codes=[429, 500, 503, 504],
)

# --- Tool 1: Fee Lookup (from cell 2.2) ---
def get_fee_for_payment_method(method: str) -> dict:
    """Looks up the transaction fee percentage for a given payment method.
    This tool simulates looking up a company's internal fee structure based
on
```

```
    the name of the payment method provided by the user.
    Args:
        method: The name of the payment method. It should be descriptive,
                e.g., "platinum credit card" or "bank transfer".
    Returns:
        Dictionary with status and fee information.
        Success: {"status": "success", "fee_percentage": 0.02}
        Error: {"status": "error", "error_message": "Payment method not
found"}
    """
    fee_database = {
        "platinum credit card": 0.02,
        "gold debit card": 0.035,
        "bank transfer": 0.01,
    }
    fee = fee_database.get(method.lower())
    if fee is not None:
        return {"status": "success", "fee_percentage": fee}
    else:
        return {
            "status": "error",
            "error_message": f"Payment method '{method}' not found",
        }

# --- Tool 2: Exchange Rate (from cell 2.2) ---
def get_exchange_rate(base_currency: str, target_currency: str) -> dict:
    """Looks up and returns the exchange rate between two currencies.
    Args:
        base_currency: Currency converting from.
        target_currency: Currency converting to.
    Returns:
        Dictionary with status and rate information.
    """
    rate_database = {
        "usd": {
            "eur": 0.93,
            "jpy": 157.50,
            "inr": 83.58,
        }
    }
    base = base_currency.lower()
    target = target_currency.lower()
    rate = rate_database.get(base, {}).get(target)
    if rate is not None:
        return {"status": "success", "rate": rate}
    else:
```

```
        return {
            "status": "error",
            "error_message": f"Unsupported currency pair:
{base_currency}/{target_currency}",
        }

# --- Agent Tool: Calculator (from cell 3.1) ---
calculation_agent = LlmAgent(
    name="CalculationAgent",
    model=Gemini(model="gemini-2.5-flash-lite",
retry_options=retry_config),
    instruction="""You are a specialized calculator that ONLY responds with
Python code. You are forbidden from providing any text, explanations, or
conversational responses.
      Your task is to take a request for a calculation and translate it
into a single block of Python code that calculates the answer.
        **RULES:**
    1.  Your output MUST be ONLY a Python code block.
    2.  Do NOT write any text before or after the code block.
    3.  The Python code MUST calculate the result.
    4.  The Python code MUST print the final result to stdout.
    5.  You are PROHIBITED from performing the calculation yourself. Your
only job is to generate the code that will perform the calculation.
       """,
    code_executor=BuiltInCodeExecutor(),
)

# --- Main Agent Definition (from cell 3.2) ---
root_agent = LlmAgent(
    name="enhanced_currency_agent",
    model=Gemini(model="gemini-2.5-flash-lite",
retry_options=retry_config),
    instruction="""You are a smart currency conversion assistant. You must
strictly follow these steps and use the available tools.
  For any currency conversion request:
   1. Get Transaction Fee: Use the get_fee_for_payment_method() tool to
determine the transaction fee.
   2. Get Exchange Rate: Use the get_exchange_rate() tool to get the
currency conversion rate.
   3. Error Check: After each tool call, check the "status" field. Stop and
report issues if status is "error".
   4. Calculate Final Amount: Use calculation_agent to generate Python code
for the final converted amount. Arithmetic is not allowed directly.
   5. Provide Detailed Breakdown:
       * Final converted amount
       * Fee percentage and fee amount
```

```
        * Amount after deducting fees
        * Exchange rate used
    """,
    tools=[
        get_fee_for_payment_method,
        get_exchange_rate,
        AgentTool(agent=calculation_agent),
    ],
)
```

4. Commands Used to Run the Project

4.1 Activate virtual environment

```
.venv\Scripts\activate.bat
```

4.2 Run the agent using ADK CLI

```
adk run day2_agent
```

4.3 Start ADK Web UI

```
adk web --port 8000
```

5. Notes on ADK CLI

- **CLI = Command Line Interface**
- Enables running, testing, and interacting with ADK agents from the terminal.
- Supports:
    - `adk create` → generate agent project
    - `adk run` → run agent
    - `adk web` → launch web interface

6. Summary (Short Checklist)

- Create project folder: **day2_agent**
- Generate ADK project: `adk create day2_agent`
- Update `.env` with API key
- Insert custom tool logic, calculator agent, and main agent into `agent.py`
- Activate virtual environment
- Run agent using: `adk run day2_agent`
- Optional UI: `adk web --port 8000`

🌟DAY 3 — Understanding Context Engineering, Sessions & Memory

This day focuses on the **core architecture** that turns a normal LLM into a **stateful, personalized AI agent**. The transcript explains three essential pillars:

- **Context Engineering** → The “brain wiring”
- **Sessions** → The “short-term working memory”
- **Memory** → The “long-term personal knowledge”

Together, these allow an agent to not just answer, but **remember, adapt, and behave intelligently over time**.

(Video : Whitepaper Companion Podcast - Context Engineering: Sessions & Memory - YouTube)

1. Why Day 3 Matters

Large Language Models (LLMs) are **stateless** by default.
Every request is treated as a new conversation unless *you* manually include history.

Day 3 teaches **how to give LLMs a working memory system**, so that:

- They remember your preferences
- They learn across conversations
- They maintain persistent knowledge
- They update their internal "understanding" every turn

This is the foundation of *real Agentic AI*.

2. Concept 1 — Context Engineering (CE)

Context Engineering = dynamically preparing the perfect input to the model for every single turn.

Prompt engineering is static.
Context engineering is **dynamic**.

Think of it as:

> “Every time the model replies, you prepare a complete information package so it has exactly what it needs — not too little, not too much.”

Why we need it

- LLMs forget everything unless you include it in the prompt.

- The context window is limited → can get cluttered ("context rot").
- Relevant info must be selected fresh every turn.

Context Engineering includes:

1. **System instructions**
2. **Tool definitions**
3. **Relevant history** (not everything — only what matters)
4. **Retrieved long-term memory**
5. **RAG documents** (if using knowledge retrieval)
6. **Scratchpad/state information**
7. **User's latest message**

It dynamically adapts every turn

CE is not written once — it is rebuilt every time.

What is Context Rot?

When the context becomes too long, chaotic, or irrelevant → quality drops.

How CE prevents context rot:

- Summarizing old messages
- Pruning irrelevant turns
- Keeping only high-signal info
- Maintaining a fresh, clean prompt

3. Context Engineering Cycle (The Turn-by-Turn Pipeline)

The transcript describes a 4-step cycle:

Step 1 | Fetch Context

Retrieve:

- memories
- documents (RAG)
- previous dialogue
- intermediate states

Step 2 | Prepare Context (hot path = must be fast)

Assemble:

- system instruction
- tools
- relevant history

- user message

This directly becomes the model input.

Step 3 | LLM Invocation + Tool Usage

Model generates:

- answer
- tool call
- reasoning

Step 4 | Upload/Store New Context (Background)

Any new “knowledge” is saved to long-term memory.

This step MUST be async — otherwise latency becomes too slow.

4. Concept 2 — Sessions (Short-Term Memory)

A **session = one conversation thread with the user.**

Contains:

a) Events

Chronological logs:

- user messages
- agent messages
- tool calls

b) Session State

Structured data for the task:
Example:

- Current booking step
- List of selected flight options
- Items in a shopping cart

Why sessions matter

- They are fast → required for real-time replies.
- They hold only *temporary* information.
- When session ends, this info usually resets.

Different frameworks handle sessions differently

Framework	Session Style
ADK	Explicit session object with events + state
LangGraph	Mutable state object (can update/replace old history)

LangGraph makes it easier to “edit the session,” replacing old turns with summaries.

5. Sessions in Multi-Agent Systems (MAS)

When multiple agents cooperate:

Two architectures exist:

1. Shared Unified History

- All agents write to the same log
- Best when agents collaborate tightly
- Risk: clutter + confusion

2. Separate Histories

- Each agent has its own history
- They only communicate via messages
- Good isolation but less shared understanding

This is why you need a **common memory layer**.

6. Production Concerns for Sessions

Important real-world design considerations:

a) Privacy

- Must remove PII *before* storing logs
- Tools like Model Armor are used
- Required for GDPR, CCPA compliance

b) Data Hygiene

- TTL (time-to-live) policies
- Deterministic order of events

c) Performance

Session history is on the “hot path.”
Too much data → slow responses.

7. Compaction (Compressing Sessions)

This prevents long history from bloating.

Simple methods

1. **Sliding window** → keep last 10–20 turns
2. **Token limit truncation** → keep recent tokens until max reached

Advanced method: Recursive Summarization

- System selects old conversation chunks
- LLM summarizes them
- Original logs are replaced with the summary
- Summary is prepended to newer messages

This must be asynchronous because it is heavy.

8. Concept 3 — Memory (Long-Term Personal Knowledge)

Memory = Knowledge that persists across sessions.

Two types:

1. Declarative Memory (Knowing WHAT)

Examples:

- User’s favorite team
- Travel preferences
- Permanent facts (age, location, interests)

2. Procedural Memory (Knowing HOW)

Examples:

- How to perform multi-step booking
- Tool usage patterns

- User-specific workflow sequences

9. Storage Systems for Memory

Most architectures use **hybrid storage**:

a) Vector Databases

Used for:

- semantic matching
- similarity search
  ("Find memories related to travel.")

b) Knowledge Graphs

Used for:

- relationships
- linking entities
  ("User → Spouse → Trip to Japan")

Combining Both = Better Retrieval

Vector DB handles fuzzy search,
Graph DB handles structured relations.

10. Memory Scope Levels

1. **User-Level Memory**
    - Most common
    - Persists across sessions
    - Personalized
2. **Session-Level Memory**
    - Temporary
    - Used only for this one conversation
3. **Application-Level Memory**
    - Global
    - Must avoid leaking user-specific data into global state

11. Memory ETL Pipeline (Extract → Transform → Load)

A core part of Day 3.

Step 1 — Extraction

Not summarizing.
It is *targeted filtering* of useful info.

The agent extracts:

- preferences
- facts
- recurring events
- habits
- instructions
- corrections

Step 2 — Consolidation

The system compares new info with old memories.

It may:

- create new memories
- update old memories
- merge two conflicting memories
- delete irrelevant memories
- decay old memories

How does the system judge what to keep?

Through **provenance**:

- Where memory came from
- How often user reinforced it
- Confidence score
- Age of memory

Step 3 — Load

The new or updated memory is saved into the store.

This MUST run asynchronously (in background).

12. Memory as a Tool

Instead of agent being passive, you give it tools:

- **create_memory()**
- **query_memory()**

The agent itself decides:

- “This is important, I should save it.”
- “Let me check user memory before answering.”

This makes agents more autonomous.

13. Retrieval Phase (Getting Memories Back)

Not all memories should be fetched every time.

Better scoring system = blended score

Score =
→ **Relevance** (semantic similarity)
→ **Recency** (how recently used)
→ **Importance** (how critical it is)

This improves quality dramatically.

Retrieval Styles

1. **Proactive** → fetch memories every turn
   Reactive → agent decides when it needs memory

14. Where to Insert Memories in the Prompt

This is crucial:

Option A: System Instructions

- Very influential
- Best for stable facts
- Risky if memory is incorrect

Option B: Conversation History

- Less clean
- Might look like the user said it
- Can clutter dialogue

Designers choose based on context.

15. Testing and Evaluation

Memory systems must be rigorously tested.

Metrics:

- Precision (did we store correct memory?)
- Recall (did we miss something important?)
- Retrieval recall@K
- Latency (<200ms recommended)

End-to-end task success is the REAL measurement.

They use an **LLM Judge** to evaluate quality.

16. Final Takeaway

To build advanced agents:

- **Context Engineering** manages immediate input
- **Sessions** hold short-term data
- **Memory** builds long-term personalization

Together, they produce AI that doesn't just answer —
it **learns**, **adapts**, and **remembers**.

DAY 3 — PDF NOTES (CLEAN, SHORT, WITH DEFINITIONS)

(Context Engineering: Sessions & Memory.pdf - Google Drive.)

1. MCP (Model Context Protocol)

A standard protocol that defines **how agents and tools communicate**.
It ensures every tool and every model speaks the same structured language.

2. Tools in MCP

Tools are **functions that the model can call** to perform actions.
They expose:

- Name
- Description
- Input schema
- Output schema

This makes tool calls reliable and predictable.

3. Tool Schemas (Input/Output JSON)

Every tool defines **exact fields** the model must send.
This prevents wrong or messy tool calls and keeps communication consistent.

4. Tool Execution Flow

The LLM decides when to call a tool.
The tool runs → returns results → results are fed back to LLM.
This loop enables multi-step reasoning.

5. Streaming Results

Some tools return output gradually (chunk by chunk).
This is useful for long-running tasks like data processing or search.

6. Agent ↔ Tool Interoperability

MCP ensures that **tools built in one system can be used by agents in another**.
This makes the ecosystem modular — tools are reusable anywhere.

7. Security: Permissions & Access Control

Tools can expose capabilities like “read file” or “write file.”
MCP enforces permission checks so the agent cannot misuse tools.

8. Tool Registry / Discovery

A registry is a place where tools are **listed and described**.
Agents can look up available tools and understand what they can do.

9. Multi-Agent Systems (MAS)

Multiple agents can work together using MCP because they share:

- Standard message format
- Standard tool definitions
- Standard response protocol

This reduces complexity when coordinating multiple agents.

10. Error Handling

Tools return structured errors (not random text).
The agent can understand exactly what went wrong and decide the next step.

11. Context Injection

Tools can provide external data that is **inserted into the model's prompt**.
This gives the LLM access to live information beyond its training.

12. Traceability & Logging

Every tool call, input, output, and error is logged.
This makes debugging easy and provides transparency.

13. Separation of Concerns

MCP separates responsibilities:

- LLM handles reasoning
- Tools handle actions
  This keeps the system clean and scalable.

14. Tool Autonomy vs Agent Control

Agents decide *when* to use tools,
but tools explicitly define *how* they must be used.

15. Reliability Through Schema Validation

Before a tool call is made, the system checks if:

- Inputs match schema
- Fields are correct
- Types are correct

This prevents runtime errors.

16. Extensibility

You can add new tools anytime without modifying the model.
Agents automatically understand new capabilities through schemas.

17. Unified Communication Format

Everything—messages, requests, results—is passed as structured JSON.
No ambiguity, no format confusion.

Short Version Summary (Final)

The PDF explains how MCP creates a universal system for connecting tools and AI models.
Tools define clear input/output schemas, communicate using structured JSON, support

streaming, enforce permissions, and can be reused across different agent frameworks. MCP ensures reliability, safety, modularity, and smooth tool-based reasoning.

DAY 3 — CODE IMPLEMENTATION NOTES

1. Create Your Project

Open VS Code → open your folder:
D:\google agentic ai course (not cd day3)

Then create the new project using:

```
adk create day3_agent
```

This generates a folder **day3_agent** with the required ADK structure.

2. Add the Complete Code (agent.py)

Open:

```
day3_agent/agent.py
```

Delete everything inside and paste the full code you provided.

Below is **exactly the code you must paste** (unchanged):

📌 **Full agent.py Code (Do NOT modify):**

```
import os
from typing import Any, Dict
from google.genai import types
from google.adk.agents import LlmAgent
from google.adk.apps.app import App, EventsCompactionConfig
from google.adk.models.google_llm import Gemini
from google.adk.tools.tool_context import ToolContext

# --- Configuration (from Section 1.5) ---
retry_config = types.HttpRetryOptions(
    attempts=5,
    exp_base=7,
    initial_delay=1,
    http_status_codes=[429, 500, 503, 504],
)

# --- Session State Tools (from Section 5.1) ---
# These tools let the agent read and write to the session's memory.

def save_userinfo(
    tool_context: ToolContext, user_name: str, country: str
) -> Dict[str, Any]:
    """
    Tool to record and save user name and country in session state.
    Args:
        user_name: The username to store in session state
        country: The name of the user's country
    """
    # Write to session state using the 'user:' prefix for user data
    tool_context.state["user:name"] = user_name
    tool_context.state["user:country"] = country
    return {"status": "success"}

def retrieve_userinfo(tool_context: ToolContext) -> Dict[str, Any]:
    """
    Tool to retrieve user name and country from session state.
    """
    # Read from session state
    user_name = tool_context.state.get("user:name", "Username not
found")
    country = tool_context.state.get("user:country", "Country not
found")
    return {"status": "success", "user_name": user_name, "country":
country}
```

```
# --- Main Agent Definition (from Section 5.2) ---
# This agent is given the tools to manage session state.
root_agent = LlmAgent(
    model=Gemini(model="gemini-2.5-flash-lite",
retry_options=retry_config),
    name="text_chat_bot",
    description="""A text chatbot.
    Tools for managing user context:
    * To record username and country when provided use `save_userinfo`
tool.
    * To fetch username and country when required use
`retrieve_userinfo` tool.
    """,
    tools=[save_userinfo, retrieve_userinfo],
)

# --- App Definition (from Section 4.1) ---
# This is the main entry point for ADK.
# We wrap our agent in an "App" to add features like compaction.
root_app = App(
    name="day3_app",
    root_agent=root_agent,
    # This enables the context compaction from Section 4
    events_compaction_config=EventsCompactionConfig(
        compaction_interval=3,  # Summarize history after 3 turns
        overlap_size=1,         # Keep 1 turn of history for context
    ),
)
```

3. Update API Key

Open:
`day3_agent/.env`

Paste:
`GOOGLE_API_KEY="YOUR_API_KEY"`

This enables your agent to access Gemini models.

4. How to Run the Agent (Correct Way)

For Day 3, you must use **adk web** (not adk run) because of the database.

4.1 Activate your virtual environment

Make sure you are in the parent folder:

```
D:\google agentic ai course
```

Run:
```
.venv\Scripts\activate.bat
```

4.2 Run the Web Interface With Database Support

This is the correct command:
```
adk web --port 8000 --db_url "sqlite:///day3_data.db"
```

Meaning of this command:

- **adk web** → runs the ADK web UI
- **--port 8000** → opens http://localhost:8000
- **--db_url** → tells ADK to create a database file named **day3_data.db**
- This database stores:
    - session state
    - user info
    - compaction summaries

This is exactly how Day 3 notebook works.

5. What the Output Looks Like in Browser

Open:
```
http://localhost:8000
```

You will see:

- Chat panel
- Debug panel (shows tool calls, memory events, compaction events)

6. Testing All Features

Test 1 — Session State Tools

You:
"Hi, my name is Sam and I am from Poland."

Agent will:

- Automatically call **save_userinfo** tool
- Store:
    - user:name = Sam

- user:country = Poland

You:
"What is my name and where am I from?"

Agent uses **retrieve_userinfo** →
"Your name is Sam and you are from Poland."

Test 2 — Database Persistence

Stop the server:

```
Ctrl + C
```

Restart:

```
adk web --port 8000 --db_url "sqlite:///day3_data.db"
```

Now test:

You: "What is my name?"
Agent: "Your name is Sam."

This proves the memory persisted in **day3_data.db**.

Test 3 — Context Compaction

If you talk for more than 3 turns:

- After every 3 turns, ADK compacts history
- Summaries appear in the debug panel
- Old messages get replaced with short summaries
- Only 1 latest turn is kept as overlap (because overlap_size=1)

This shows your agent is now:

- efficient
- less costly
- capable of long conversations
- context-rot resistant

7. Why This Code Is Important (Short Explanations)

✔ **Session State Tools**
These tools store temporary user-specific data during a conversation.
✔ **events_compaction_config**
Automatically summarizes long history to prevent context window overflow.
✔ **Database (SQLite)**
Stores memory *permanently*, so the agent remembers even after restarting.
✔ **App() wrapper**

Adds advanced features on top of the LLM agent (compaction, persistence).

✔ **root_agent**

Main brain of the system. Uses:

- model
- tools
- description

8. Final Short Summary (Revision Style)

1. Create project → `adk create day3_agent`
2. Paste full agent.py code
3. Add API key in `.env`
4. Activate venv
5. Run: `adk web --port 8000 --db_url "sqlite:///day3_data.db"`
6. Test:
    - Session state tools
    - Database memory
    - Context compaction

This gives you a fully working **memory-enabled AI agent** in VS Code.

🌟DAY 4 — TRANSCRIPT NOTES (AGENT QUALITY: EVALUATION + OBSERVABILITY)

([(1) Whitepaper Companion Podcast - Agent Quality - YouTube](#))

1. Core Problem: Agents Are Unpredictable

AI agents are *non-deterministic* — the same input can create different actions.
Because of this, you cannot rely only on final outputs. You must evaluate how the agent thinks and behaves throughout the entire process.

2. Message 1 — "The Trajectory Is the Truth"

Trajectory means **the entire path** the agent takes:

- Its reasoning
- Its decisions
- Its tool calls
- Its interpretation of results

Even if the final answer is correct, a messy or risky trajectory is a quality failure.

3. Message 2 — Observability Is Essential

You must be able to **see inside the agent's reasoning and tool usage**.
Without logs, traces, and metrics, you cannot debug or measure quality.

4. Message 3 — Evaluation Must Be Continuous (The Flywheel)

Quality is not a one-time test. It improves through a loop:

1. Observe
2. Evaluate
3. Learn
4. Improve

This repeats forever while the agent runs.

5. Old Software vs Agents (Truck vs Formula-1 Car Analogy)

Traditional software = predictable → fixed steps → easy to test.
Agents = dynamic → make decisions in real time → harder to evaluate.

Agents often fail quietly (outputs look fine but are wrong), not through crashes.

6. Key Failure Types in Agents

These failures are unique to agents (not normal software):

a. Algorithmic Bias
The agent inherits bias from data, e.g., unfair hiring recommendations.
b. Hallucinations
The agent confidently invents facts, sources, or numbers.
c. Performance/Concept Drift
The world changes, but the agent runs on outdated assumptions.
d. Emergent Weird Behaviors
The agent finds loopholes or forms "superstitions" to achieve goals.

7. Why Evaluating Agents Is Hard

Agents involve:

- Multi-step planning
- External tools
- APIs with unpredictable responses
- Memory that changes the agent over time

All of this breaks old testing methods.

8. Goal of Evaluation = Shift from Verification → Validation

Verification = “Did we build it correctly according to spec?”
Validation = “Does this agent actually help the user achieve the goal?”

9. Four Pillars of Agent Quality

1. Effectiveness
Did the agent complete the user’s real intention (not just task text)?
2. Efficiency
Was it fast, cheap, and minimal in steps/tool calls?
3. Robustness
Can it handle unclear input, API errors, missing data, or confusion?
4. Safety & Alignment
Ethics, harm prevention, boundaries, avoiding leaks, resisting attacks.

10. Evaluation Strategy: Outside-In → Inside-Out

Step 1: End-to-End Evaluation (Black Box)

Only look at the final result and user satisfaction.
This tells **what** went wrong.

Step 2: Trajectory Evaluation (Glass Box)

Inspect reasoning, tool calls, and choices.
This tells **why** it went wrong.

11. Trajectory Evaluation Checks

- **Reasoning errors** (repeating, forgetting context)
- **Wrong tool selection**
- **Wrong tool inputs**
- **Misinterpreting tool responses**
- **Ignoring API errors**

These pinpoint exactly where the failure started.

12. ADK Tip — Saving Successful Trajectories

ADK lets you save a *perfect run* into a `test.json` file.
This becomes the “golden path.”
Later, you test again to ensure the agent follows the same successful process.

13. Automation + Human Review Hybrid System

Automation

- Fast, scalable
- Uses metrics like ROUGE/BERTScore
- Good for catching sudden drops in performance

LLM-as-Judge

A strong model evaluates outputs.
Best method = **pairwise comparison** (A vs B), avoids biased scoring.

Agent-as-Judge

A specialized agent evaluates another agent's reasoning process.

Human-in-the-loop (HITL)

Humans are required for:

- High-stakes actions
- Edge cases
- Creating high-quality evaluation datasets

14. Reviewer UI Best Practice

An evaluator should see:

- The conversation
- The internal reasoning/trajectory
  Side-by-side.

This makes judgement easier and accurate.

15. Responsible AI (Safety Architecture)

Safety must be built **into** the agent, not added later.

Includes:

- Red teaming (attempting to break safety rules)
- Guardrail plugins (before/after model checks)
- Preventing prompt attacks
- Stopping private data leaks

16. Observability: The Technical Foundation

a. Logging

Every step is recorded as structured data (JSON).
Includes tool inputs, tool outputs, chain of thought, time.

b. Tracing

Connects the logs into a coherent story of cause → effect.
Usually built on **OpenTelemetry**.

c. Metrics

Aggregated stats for dashboards.
Two main groups:

For engineering: latency, error rates, token cost
For product/data teams: accuracy, helpfulness, trajectory quality

17. Dynamic Sampling

Tracing everything is expensive.
Solution:

- Always trace failed runs
- Trace only some successful runs

This gives quality + performance

18. Agent Quality Flywheel (Final Loop)

1. Define quality pillars
2. Add observability
3. Evaluate outcomes + trajectories
4. Feed insights into design
5. Improve the agent
6. Repeat continuously

This creates a system that gets smarter and more reliable over time.

SHORT SUMMARY

- Quality must be designed from the start.
- Trajectory matters more than final answer.
- Observability (logs, traces, metrics) is essential.
- Evaluation happens continuously through automation + humans.
- Safety is built into the agent lifecycle, not added later.
- Use the flywheel loop: observe → evaluate → learn → improve.

DAY 4 – PDF NOTES (Agent Quality: Evaluation to Observability)

(Agent Quality.pdf - Google Drive)

1. Why Agent Quality Is Hard

AI agents behave differently each time, even with the same input.
This makes them unpredictable and difficult to test using traditional software methods.

2. Quality Must Be Designed In (Not Added Later)

Quality is not a final testing activity.
An agent must be built from the beginning to support measurement, debugging, and safe operation.

3. The Trajectory Is the Truth

You evaluate an agent not only by the final answer but by the **full sequence of decisions** it took:

- Reasoning
- Tool usage
- State changes
- External interactions

A correct answer with a dangerous trajectory is still a failure.

4. Modern Agents Require New Evaluation Methods

Traditional models are static.
Agents are dynamic systems involving:

- Planning
- Tools
- APIs
- Memory
- Autonomy

So they need **new evaluation techniques**.

5. Four Pillars of Agent Quality

1. Effectiveness

Does the agent accomplish the user's actual goal, not just the literal request?

2. Efficiency

How fast, cheap, and minimal is the route the agent took?
Measured by latency, tokens, steps, tool calls.

3. Robustness

How well the agent handles:

- Errors
- Unclear instructions
- Missing data
- External API failures

A robust agent recovers gracefully.

4. Safety & Alignment

The agent must:

- Avoid harmful content
- Respect boundaries
- Protect private data
- Resist attacks
- Follow ethical rules

Safety overrides all other pillars.

6. Two-Level Evaluation Strategy

A. End-to-End Evaluation

You look only at the final answer.
Measures:

- Success/failure
- User satisfaction
- Accuracy

This tells you **what** went wrong.

B. Trajectory Evaluation

You inspect the reasoning path.
Reveals:

- Why the agent failed

- Where the error started
- Whether tools were used correctly
- Whether context was maintained

This tells you **why** it went wrong.

7. Common Failure Sources

1. Reasoning Failures
Repeating steps, losing context, unclear planning.
2. Tool Misuse
Selecting the wrong tool or feeding incorrect parameters.
3. Tool Response Misinterpretation
Not handling error messages correctly (e.g., ignoring 404 errors).
4. Safety Violations
Unintentionally producing harmful outputs.
5. Memory Issues
Forgetting or incorrectly storing information over time.

8. Successful Trajectory as Regression Test

A known-good successful run can be saved as a **test.json** file.
It captures:

- All steps
- Reasoning
- Tool calls
- Outputs

Later, you run it again to ensure the agent follows the same path.
If it diverges → your update broke something.

9. Hybrid Evaluation System

Automation

Fast and scalable:

- Text similarity metrics (ROUGE, BERTScore)
- Heuristic checks
- Trend monitoring

Good for early warning signals.

LLM-as-a-Judge

A strong model evaluates the output.
Best method is:

- **Pairwise comparison** (A vs B)
- Avoids biased scoring
- Produces cleaner “win rate” signals

Agent-as-a-Judge

A specialized agent reviews another agent’s execution trace and reasoning.

Human-in-the-loop

Humans remain essential:

- For high-stakes actions
- For nuanced quality checks
- For building “golden sets” of correct reference examples

10. Responsible AI Layer

Safety evaluations must be integrated throughout the system:

- Red teaming (active attempts to break the agent)
- Guardrail functions
- Pre- and post-processing safety checks
- Prevent prompt injection
- Prevent PII leakage
- Control high-risk actions with human approval

11. Observability: Foundation for Quality

1. Logging

Every step of the agent is captured:

- Inputs
- Outputs
- Reasoning summaries
- Tool usage
- Errors

Logs must be structured (JSON), not free text.

2. Tracing

Connects logs into a full chain-of-events.
Shows:

- Cause → effect
- Parent → child steps
- Tool call flow

Usually implemented with **OpenTelemetry**.

3. Metrics

Aggregated numeric signals.
Two types:

Operational Metrics (for SRE/Engineering)

- P50/P95/P99 latency
- Error rates
- Token cost
- Throughput

Quality Metrics (for product/data teams)

- Accuracy
- Helpfulness
- Trajectory quality
- Safety violations

12. Dynamic Sampling

Collecting every trace is too expensive.
So you:

- Always trace **100% of failures**
- Trace only a small percentage of successful runs

This balances performance with visibility.

13. Agent Quality Flywheel

A continuous improvement loop:

1. Define quality targets
2. Add observability
3. Evaluate outputs + trajectories
4. Identify failures
5. Improve architecture and reasoning
6. Repeat

Every run becomes a source of learning.

14. Key Takeaways (PDF)

- Quality must be built into agent architecture.
- Trajectory matters more than the final answer.
- Observability is mandatory for debugging and trust.
- Evaluation requires a mix of LLM judges + automation + humans.
- Safety and alignment are non-negotiable pillars.
- The quality flywheel ensures continuous improvement.

Day 4 — Implementation (VS Code project)

Environment & setup (short):

- Create folder `day4_agent` or scaffold with ADK:
  `adk create day4_agent`
- Add API key to `day4_agent/.env`.
- Activate venv: `.venv\Scripts\activate.bat`.
- Run ADK web with debug logging to see traces:
  `adk web --port 8000 --log_level DEBUG`
  (Terminal will show detailed TRACE/DEBUG logs; web UI events will show execution trace.)

Files created (important list):

- `day4_agent/agent.py` — main agent code (paste exact code below).
- `day4_agent/.env` — contains `GOOGLE_API_KEY`.
- `day4_agent/__init__.py` — empty file OK.

Exact `agent.py` code (paste as-is):

```
from typing import List
from google.genai import types
from google.adk.agents import LlmAgent
from google.adk.models.google_llm import Gemini
from google.adk.tools.agent_tool import AgentTool
from google.adk.tools.google_search_tool import google_search
```

```
# --- Configuration ---
retry_config = types.HttpRetryOptions(
    attempts=5,
    exp_base=7,
    initial_delay=1,
    http_status_codes=[429, 500, 503, 504],
)

# --- The Tool (This is where the bug was!) ---
# In the "broken" version, the type hint was just `str`.
# That made the agent pass a giant string, and `len()` counted the
letters, not the papers.
# We fixed it by changing it to `List[str]`.

def count_papers(papers: List[str]):
    """
    This function counts the number of papers in a list of strings.
    Args:
      papers: A list of strings, where each string is a research paper.
    Returns:
      The number of papers in the list.
    """
    return len(papers)

# --- Google Search Sub-Agent ---
google_search_agent = LlmAgent(
    name="google_search_agent",
    model=Gemini(model="gemini-2.5-flash-lite",
retry_options=retry_config),
    description="Searches for information using Google search",
    instruction="""Use the google_search tool to find information on the
given topic.
    Return the raw search results.
    If the user asks for a list of papers, then give them the list of
research papers you found and not the summary.""",
    tools=[google_search],
)

# --- Root Agent ---
root_agent = LlmAgent(
    name="research_paper_finder_agent",
    model=Gemini(model="gemini-2.5-flash-lite",
retry_options=retry_config),
    instruction="""Your task is to find research papers and count them.
```

```
    You MUST ALWAYS follow these steps:
    1) Find research papers on the user provided topic using the
'google_search_agent'.
    2) Then, pass the papers to 'count_papers' tool to count the number
of papers returned.
    3) Return both the list of research papers and the total number of
papers.
    """,
    tools=[AgentTool(agent=google_search_agent), count_papers],
)
```

Commands to run (terminal):

```
# create project
adk create day4_agent

# add API key to day4_agent/.env
# GOOGLE_API_KEY="YOUR_API_KEY"

# activate venv (Windows)
.venv\Scripts\activate.bat

# run ADK web with debug logs (observability)
adk web --port 8000 --log_level DEBUG
```

What to inspect (observability exercise):

- Open `http://localhost:8000` and send: `"Find recent papers on quantum computing."`
- Terminal (because of `--log_level DEBUG`) will print detailed logs:
    - exact requests sent to Google,
    - tool calls and parameters,
    - returned observations,
    - any exceptions/errors and stack traces.

- In Web UI: open **Events** tab → expand rows to view the trace (prompt, intermediate tool calls, observations, final answer).
- The code includes the **fixed bug** (type hint `List[str]`), so `count_papers` returns number of items (not string length).

Short summary:

- `adk create day4_agent` → paste `agent.py` above → `.env` with API key → `adk web --log_level DEBUG` → open web UI and inspect terminal logs + Events traces to find issues in the agent's trajectory.

Final quick checklist (copy/paste)

1. Create project scaffold (for each day):
   `adk create dayX_agent`
2. Add API key to `.env` inside the project:
   `GOOGLE_API_KEY="YOUR_API_KEY"`
3. Paste the exact `agent.py` provided for the corresponding day.
4. Activate venv (Windows example):
   `.venv\Scripts\activate.bat`
5. Run:
   - Day 1 / Day 2 run examples: `adk run day2_agent` or `adk run day1_agent`
   - For web UI / DB: `adk web --port 8000 --db_url "sqlite:///day3_data.db"`
   - For observability: `adk web --port 8000 --log_level DEBUG`
6. Test via web UI or runner, inspect traces/events/terminal logs as required.

🌟DAY 5 — TRANSCRIPT NOTESTopic: Deploying, Scaling & Productionizing AI Agents

(Whitepaper Companion Podcast - Prototype to Production)

1. Prototype vs. Production Gap

- You can build a demo quickly, but making it reliable in real-world use is hard.
  This gap is called the **"last-mile production gap."**

2. Why This Matters

- Real deployments need **security, testing, validation, monitoring**, not just good prompts.
- About **80% of effort** goes into system infrastructure, not model logic.

3. Why Agents Are Harder Than Normal ML

- Agents are **dynamic**: they choose tools, think differently each time, use memory.
- Their execution path is not fixed, making them harder to test and deploy.

4. Key Challenges in AgentOps

- **Dynamic Tooling**: Agent picks tools on the fly.
- **State Management**: Agents need memory across sessions.
- **Cost unpredictability**: Different number of steps per query.
- Requires more advanced infrastructure.

5. People & Process Come First

- Good operations require clear roles and responsibilities.
- Two new core roles:

Prompt Engineer

- Defines system instructions, rules, safety boundaries, domain behavior.

AI Engineer

- Implements backend, tools, safety systems, evaluation pipelines.

6. Pre-Production: Evaluation-Gated Workflow

- No agent touches real users until it **passes strict tests**.
  Evaluation checks:
  - Correctness
  - Safety
  - Proper tool usage
  - Correct reasoning path
- Testing is not only the final answer — but **the full behavior**.

7. Two Evaluation Approaches

Manual Review

- Engineer runs tests and attaches result to pull request.

Automated Gate (CI/CD)

- CI/CD blocks deployment automatically if quality drops below threshold.

8. Golden Dataset

- A stable set of realistic test prompts for evaluation.
- Used to compare new versions with old versions reliably.

9. CI/CD Pipeline Stages

Stage 1: Pre-Merge (CI)

- Fast checks: unit tests, formatting, early agent evaluation.

Stage 2: Staging

- Heavy tests: load tests, integration tests, dogfooding.

Stage 3: Production Gate

- Only the *exact tested build* is deployed.
- Requires final human sign-off.

10. Safe Release Methods

Canary Release

- Release to 1% users first.

Blue-Green Deployment

- Two environments; switch traffic only when new one is safe.

A/B Testing

- Compare two agent versions on real user outcomes.

11. Version Everything

- Version code + prompts + tool schemas + memory schemas.
- Enables instant rollback if something breaks.

12. Security Risks in Agents

- **Prompt Injection** – tricking the agent to ignore instructions.
- **Data Leakage** – exposing sensitive info.
- **Memory Poisoning** – corrupting agent's memory.

13. Secure AI Agents (3 Layers – SIF Model)

Layer 1: Policy Definition

- Core rules and boundaries via system instructions.

Layer 2: Guardrails & Filtering

- Input filtering
- Output filtering
- Human-in-the-loop for sensitive actions

Layer 3: Continuous Assurance

- Ongoing testing
- Red team attacks
- Constant monitoring

14. Operations: Observe → Act → Evolve

Agents need continuous operation and tuning.

15. Observability Pillars

Logs

- Detailed event records.

Traces

- Connect logs into a complete storyline of what happened.

Metrics

- High-level stats (latency, error rate, cost, satisfaction).

16. Making Agents Scalable

- Separate logic from state (memory stored in a DB).
- Add more agent instances when needed.

17. Reliability Techniques

- Retries with exponential backoff
- Idempotent tools (safe to retry)
- Caching expensive calls

- Cost optimization through batching & shorter prompts

18. Security Incident Response

Contain → Triage → Resolve

- Disable risky feature using feature flag
- Human review of affected cases
- Patch + retest + redeploy fast

19. Continuous Improvement Loop

- Production failures become new test cases.
- Enables rapid, safe iteration.

20. Agent Interoperability

Two major communication standards:

MCP (Model Context Protocol)

- Agent → Tools / Data / APIs (stateless)

A2A (Agent-to-Agent)

- Agent ↔ Agent collaboration (stateful, goal-driven)

21. Agent Cards

- JSON-based identity cards describing an agent's abilities, endpoint, skills.
- Makes agents discoverable by other agents.

22. Registries

- Central catalog for tools or agents.
- Helps discovery + governance + prevents duplication.

DAY 5 — PDF NOTESTopic: Prototype → Production for AI Agents

(Prototype to Production.pdf - Google Drive)

1. Last-Mile Problem

- The challenge of turning a good demo into a reliable production system.
- About **80% of work is infrastructure, testing, monitoring, security**, not the model.

2. Why Traditional MLOps Isn't Enough

- Standard ML models have predictable inputs/outputs.
- Agents are **dynamic**: they choose tools, reason differently each time, maintain state.
- This unpredictability requires stronger systems.

3. New Roles in AI Agent Workflows

Prompt Engineer

- Designs the agent's system instructions, safety rules, and expected behavior.

AI Engineer

- Builds backend, integrates tools, sets up memory, guardrails, eval systems.

4. Evaluation-Gated Deployment

- No agent is allowed into production until it **passes automated evaluation**.
- Checks include: correctness, safety, tool usage, reasoning path.

5. Golden Dataset

- A curated set of test cases that represent real user scenarios.
- Used to measure agent quality consistently.

6. CI/CD Funnel (3-Stage Pipeline)

Phase 1: Pre-merge CI

- Fast tests: linting, unit tests, security checks, basic agent evaluation.

Phase 2: Staging

- Heavy tests: load tests, integration tests, internal testing ("dogfooding").

Phase 3: Production Release Gate

- Only the validated build is deployed.
- Requires a final human approval.

7. Safe Rollout Strategies

Canary Release

- Release to 1% of users, monitor, then expand.

Blue-Green Deployment

- New version goes to a parallel environment; switch traffic only when safe.

A/B Testing

- Compare two agent versions based on business outcomes.

8. Version Everything

- Code, prompts, memory schema, tool schemas — all versioned.
- Enables instant rollback when something breaks.

9. Agent Security (SIF Model)

Layer 1: Policy Definition

- The agent's core rules, limits, and constitution.

Layer 2: Guardrails & Filtering

- Input filters, output filters, human-in-the-loop for risky actions.

Layer 3: Continuous Assurance

- Ongoing safety evaluations, red-team testing, monitoring.

10. Observability: Logs / Traces / Metrics

Logs

- Raw event history: tool calls, decisions.

Traces

- Connect logs together to show end-to-end reasoning.

Metrics

- High-level health stats: latency, cost, success rate.

11. Decoupling State & Logic

- Agent memory/session stored externally (databases).
- Allows horizontal scaling by adding more agent instances.

12. Reliability Techniques

- Retries with exponential backoff.
- Idempotent tool design (safe to retry).
- Caching to reduce cost and latency.

13. Security Incident Playbook

Contain

- Disable the problematic feature or tool immediately.

Triage

- Route suspicious cases to human review.

Resolve

- Patch the issue, test, redeploy quickly.

14. Continuous Learning Loop

- Production errors become **new test cases**.
- The CI/CD system keeps improving the agent fast.

15. Interoperability (MCP vs A2A)

MCP (Model Context Protocol)

- For accessing tools, data sources, APIs.
- Stateless, structured requests.

A2A (Agent-to-Agent Protocol)

- For collaboration between agents.
- Stateful, goal-oriented.

16. Agent Registry

- Central directory listing all tools or agents.
- Helps discovery, governance, and avoids duplicate work.

DAY 5 — A2A (Agent-to-Agent) CODE IMPLEMENTATION NOTES

1. What You're Building

Day 5's code assignment teaches you how to make **two separate AI agents talk to each other over a network**.

You will create:

(1) Vendor Agent

Runs on **port 8001** and has a product catalog tool.

(2) Support Agent

Runs locally in your terminal, and when a user asks about a product, it calls the Vendor Agent over A2A.

This simulates two different companies communicating.

2. Install Required Libraries

A2A features are not included by default.

You must install:

```
pip install "google-adk[a2a]" uvicorn
```

- `google-adk[a2a]` → enables Agent-to-Agent communication
- `uvicorn` → runs the Vendor Agent as a web service

3. Folder Structure

Create a folder named:

```
day5_a2a
```

Inside it create:

- **vendor_server.py** → external agent server
- **support_client.py** → your main customer support agent
- **.env** → for API key

4. Vendor Agent (vendor_server.py)

Purpose

This agent acts like an **external product catalog** from another company.
It contains a **tool: get_product_info** which returns details for specific products.

Key Concepts

- Runs on **localhost:8001**
- Exposed using `to_a2a()` so other agents can contact it
- Provides product info based on a mock internal dictionary
- If product doesn't exist → returns a fallback list of available items

Important Part

This line transforms the agent into a server:

```
app = to_a2a(product_catalog_agent, port=8001)
```

This is where the "magic" happens.
Your Python file becomes a *server* hosting an AI agent.

5. Support Agent (support_client.py)

Purpose

This is your main customer support agent.

It has two responsibilities:

1. Receive user questions
2. Forward product questions to the Vendor Agent over A2A

How it connects

It connects using a **RemoteA2aAgent**, which uses the Vendor Agent's *agent card*, located at:

```
http://localhost:8001/.well-known/agent.json
```

This is exactly what this line does:

```
remote_product_catalog_agent = RemoteA2aAgent(
    agent_card=f"http://localhost:8001{AGENT_CARD_WELL_KNOWN_PATH}",
)
```

How the Support Agent behaves

- If the user asks about a product
- It automatically calls the remote agent
- Uses that response to help the user

6. Environment File (.env)

Same as previous days:

```
GOOGLE_API_KEY="YOUR_API_KEY"
```

7. Running the System (Two-Terminal Method)

Terminal 1 → Start the Vendor Agent (Server)

```
uvicorn vendor_server:app --host localhost --port 8001
```

This terminal MUST stay running.

You will see:

- "Application startup complete"
- "Running on http://localhost:8001"

This means your vendor service is live.

Terminal 2 → Run the Support Agent

```
adk run support_client:root_agent
```

This launches the interactive CLI agent.

8. Test the A2A Network

Ask inside Terminal 2:

```
Is the iPhone 15 Pro in stock?
```

Behind the Scenes

1. Support Agent receives your question
2. It decides it needs product info
3. It calls the Vendor Agent at `http://localhost:8001`

4. Vendor Agent runs `get_product_info()`
5. Vendor Agent sends result back
6. Support Agent replies to you with the final answer

This proves **true network-based agent collaboration**.

9. Why This Matters

This is the foundation of:

- multi-agent systems
- companies connecting agents with partner companies
- internal enterprise AI networks
  scalable agent ecosystems
- agent marketplaces (future)

A2A is how complex systems will work in production.

🌟Input commands and expected output for all:

To run all first 4 together:
adk run

Day 1: The Simple Time Agent

Goal: Show an agent answering a basic question.

Terminal Command: *(Run from the main folder `google agentic ai course`)*
adk web my_agent --port 8000

- **What to Type in Browser:**
  "What time is it in London?"
- **Expected Output:** "10:30 AM" (This comes from your mock code).

Day 2: Tools (Currency Converter)

Goal: Show the agent chaining tools (Fee -> Rate -> Calculator). This is the tricky one. If it refuses to calculate, use the "Robotic" prompt below.

Input Command:
adk web day2_agent --port 8000

- **What to Type in Browser:**
  "Calculate the final amount for 100 USD to INR using a Bank Transfer. Use the available tools to find the fee and rate."
- **Expected Output:** It should show bubbles for `get_fee`, `get_exchange_rate`, and `CalculationAgent`. Result should be around **8274.42 INR**.

Day 3: Memory (Database)

Goal: Show the agent remembering your name after a restart.

Input Command: *(Must use the database flag!)*
adk web day3_agent --port 8000 --session_db_url "sqlite:///day3_data.db"

- **What to Type in Browser (Step 1):**
  "My name is Ojal and I am from Pune." *(Wait for it to say "Saved" or "Nice to meet you").*
- **What to Type in Browser (Step 2):** *(Click the trash icon 🗑️ to start a fresh chat, or use the previous session)*
  "What is my name and where am I from?"
- **Expected Output:** "Your name is Ojal and you are from Pune."

Day 4: Observability (Debugging)

Goal: Show the logs scrolling in the terminal.

Input Command: *(Must use the debug flag!)*
adk web day4_agent --port 8000 --log_level DEBUG

- **What to Type in Browser:**
  "Find recent papers on quantum computing."

Day 5: Agent to Agent (A2A)

Goal: Show two terminals talking to each other.

Terminal 1 (Server):
Input Command:
cd day5_a2a

uvicorn vendor_server:app --host localhost --port 8001

Terminal 2 (Client): *(Open a new terminal, make sure you are in `day5_a2a` folder)*
Input Command:
adk run .

- **What to Type in Terminal 2:**
  "Is the iPhone 15 Pro in stock?"
- **Expected Output:** It will pause for a second (fetching data from Terminal 1) and then reply: "The iPhone 15 Pro is in stock..."