The ADK Solutions Architect's Handbook (Extended Edition)

This document is the canonical guide for designing and building robust, scalable, and maintainable applications with the Google Agent Development Kit (ADK). It provides architectural patterns, best practices, component deep dives, and advanced techniques.

Part 1: Foundational Principles of ADK Development

These are non-negotiable principles that form the foundation of any high-quality ADK application.

1. **Strict Separation of Concerns**: Logic MUST be separated into distinct files. A standard agent package ALWAYS contains:
    - `agent.py`: Core `Agent` definition(s) and their wiring.
    - `prompts.py`: All LLM instruction strings. No prompts should be hardcoded in `agent.py`.
    - `tools.py`: All custom business logic, API calls, and external interactions.
2. **Package-First Mentality**: ADK projects are Python packages. The structure MUST follow the `project_name/agent_name/` pattern. All internal imports MUST be relative (e.g., `from . import tools`). This ensures portability and proper dependency management.
3. **Embrace Simplicity**: Always begin with the simplest viable architecture. A single, powerful agent with well-defined tools is often superior to an unnecessarily complex multi-agent system. Introduce complexity only when a clear separation of duties is required.
4. **Stateless Tools, Stateful Context**: Tools themselves should be stateless. All state (e.g., user preferences, conversation history, cart contents) MUST be managed explicitly through the `ToolContext` and `CallbackContext` objects provided by the ADK runtime. This is critical for scalability, testability, and predictable behavior.

5. **Prompts as Code**: Treat the instruction in `prompts.py` with the same rigor as Python code. It is an executable part of your application. It must be clear, specific, persona-driven, and explicitly define the agent's capabilities, constraints, and desired output format.

Part 2: Core Architectural Patterns

Select the correct pattern based on the complexity and nature of the user's request.

Pattern A: The Simple Tool-Using Agent

- **Use Case**: The default pattern for most tasks. Ideal for agents that need to perform actions, answer questions using external APIs, or follow a straightforward reasoning process.
- **Examples**: `customer-service` agent, a simple weather agent.
- **Pros**: Easy to understand, build, and maintain. Fast iteration cycle.
- **Cons**: Can become unwieldy if the number of tools or the complexity of the prompt grows too large.
- **Diagram**: `User -> Agent -> Tool(s) -> Response`

Pattern B: The RAG-Powered Q&A Agent

- **Use Case**: The agent's primary function is to answer questions based on a large, proprietary corpus of documents.
- **Example**: `rag` agent.
- **Pros**: Excellent for knowledge retrieval. Offloads the burden of "knowing everything" from the prompt to a scalable RAG backend.
- **Cons**: Requires setup and maintenance of a Vertex AI RAG Engine corpus. Less suitable for tasks requiring actions or state changes.
- **Key Component**: `google.adk.tools.retrieval.vertex_ai_rag_retrieval.VertexAiRagRetr`

Pattern C: The Multi-Agent System (Coordinator/Specialist)

- **Use Case**: For complex, multi-step workflows where tasks can be broken down into distinct, specialized roles.

- **Examples**: `financial-advisor` (Data Analyst, Risk Analyst), `data-science` (DB Agent, Analytics Agent).
- **Pros**: Highly modular and scalable. Each specialist agent can have its own tailored prompt, tools, and even a different LLM, optimizing it for its specific task. Promotes code reuse.
- **Cons**: Introduces more architectural complexity. Requires careful design of the Coordinator's prompt to ensure proper delegation.
- **Key Component**: `google.adk.tools.agent_tool.AgentTool`, which wraps a specialist agent so it can be used as a tool by the coordinator.
- **Diagram**: `User -> Coordinator -> AgentTool(Specialist) -> Response`

Pattern D: The Code-Executing Agent

- **Use Case**: When the agent needs to perform dynamic data manipulation, analysis, plotting, or complex calculations that cannot be pre-defined in a tool.
- **Example**: `data-science` agent's `analytics` sub-agent.
- **Pros**: Extremely powerful and flexible for analytical tasks. Can generate visualizations and handle unforeseen data structures.
- **Cons**: Incurs the overhead of a secure code execution environment. Requires careful prompt engineering to generate safe and correct code.
- **Key Component**: The `code_executor=VertexAiCodeExecutor()` parameter in the `Agent` constructor.

Pattern E: Hybrid Patterns

These patterns are not mutually exclusive. A sophisticated agent often combines them. * **Example**: A Multi-Agent System where a Specialist Agent is a RAG-Powered Agent. * **Example**: A Code-Executing Agent that uses a simple Function Tool to first fetch data from a database before analyzing it.

Part 3: Component Deep Dive

The `tools.py` File: Anatomy and Best Practices

1. Anatomy of a Good Tool

A tool should be a well-defined, documented Python function.

```
def get_product_recommendations(plant_type: str, customer_id: str) ->
    """
    Provides product recommendations based on the type of plant.
    A clear, concise docstring explaining what the tool does is CRITIC
    The LLM uses this docstring to decide when and how to call the too

    Args:
        plant_type: The type of plant (e.g., 'Petunias'). Type hints a
        customer_id: Optional customer ID for personalized recommendat

    Returns:
        A dictionary of recommended products. The return type hint is
    """
    # ... function logic ...
    return {"recommendations": [...]}
```

2. Error Handling in Tools

Never let a tool crash silently. Return a descriptive error message that the LLM can use to self-correct or inform the user.

```
def approve_discount(value: float, ...) -> dict:
    """Approves a discount if it is within predefined limits."""
    if value > 10.0:
        # Return a structured error message!
        return {
            "status": "rejected",
            "message": "Discount value is too large. Manager approval
        }
    # ... approval logic ...
    return {"status": "approved"}
```

3. Using State in Tools (`ToolContext`)

To read from or write to the session state, add `tool_context: ToolContext` as the last argument to your tool function.

```
from google.adk.tools import ToolContext

def add_item_to_cart(item_id: str, tool_context: ToolContext) -> dict:
    """Adds an item to the user's shopping cart in the session state.
    # 1. Read current state
    cart = tool_context.state.get("user_cart", [])

    # 2. Modify state
    cart.append(item_id)

    # 3. Write state back
    tool_context.state["user_cart"] = cart

    return {"status": "success", "new_cart_size": len(cart)}
```

The `prompts.py` File: Prompt Engineering Strategies

1. **Persona-Driven Prompts**: Give the agent a clear identity. This dramatically improves the tone and quality of its responses.

    - **Bad**: "You are an assistant."
    - **Good**: "You are 'Project Pro,' the primary AI assistant for Cymbal Home & Garden, a big-box retailer... Maintain a friendly, empathetic, and helpful tone."

2. **Explicit Instructions & Constraints**: Tell the agent exactly what to do and what *not* to do. Use markdown for structure. ``` **Workflow:**

    1. Greet the user and access their cart using `access_cart_information`.

2. ...

Constraints: - You MUST use markdown to render any tables. - Never mention "tool_code" or "print statements" to the user. ```

3. **Few-Shot Examples (When Needed)**: For complex output formats, provide a few examples directly in the prompt.
    - **Use Case**: For an agent that must generate a specific JSON or XML structure.
    - **Example**: "When the user asks for a summary, format your output like this: `{'summary': '...', 'key_points': [...]}`."
4. **Dynamic Context with `GLOBAL_INSTRUCTION`**: Use this to inject information that changes from turn to turn, keeping the main `instruction` static and cacheable.
    - **Example from `customer-service`**: Loading the customer's profile into the `global_instruction` at the start of each turn.

State Management Best Practices (`state`)

The `state` object is a dictionary-like object that persists across turns in a session.

- **What to Store in State**:
    - Session identifiers (`session_id`, `user_id`).
    - User profile information (`customer_profile`).
    - Conversation context (`user_cart`, `last_topic`).
    - User preferences (`preferred_language`).
- **What NOT to Store in State**:
    - **Secrets or Credentials**: Never store API keys, tokens, or passwords in the state. Use secure secret management.
    - **Large Data Payloads**: Do not store large dataframes or file contents in the state. Instead, save them as an artifact and store the artifact ID in the state.

    - **Internal Tool Implementation Details**: The state is for session-level data, not for the internal workings of a single tool call.

Part 4: Advanced ADK Techniques

Callbacks for Cross-Cutting Concerns

Callbacks are functions that run at specific points in the agent's lifecycle. They are perfect for logic that applies to many tool calls. * **`before_tool_callback`**: Ideal for input validation, request logging, or implementing business rules *before* a tool executes. * **Example**: A callback that checks if a discount amount is within an auto-approval limit. If so, it can bypass the `ask_for_approval` tool entirely and return an "approved" message immediately. * **`after_tool_callback`**: Useful for post-processing tool results, logging outputs, or triggering follow-up actions. * **`before_agent_callback`**: Runs before the agent's main reasoning step. Perfect for loading initial session data, like fetching a customer profile from a CRM.

Multi-modality (Handling Image/Video)

The ADK agent itself doesn't directly process video streams. The correct pattern is to use a tool to facilitate the connection, while the multi-modal processing happens on the client-side or a separate service. 1. Agent's prompt instructs it to ask the user for a video if needed. 2. If the user agrees, the agent calls a tool like `send_call_companion_link(phone_number)`. 3. This tool sends a link to the user. The user clicks it, opening a separate application that handles the video stream and analysis. 4. The results of that analysis (e.g., "The plant was identified as Petunias") are then fed back into the conversation as user input for the agent to continue.

Part 5: The Development and Deployment Lifecycle

Testing and Evaluation

A robust agent requires two forms of testing:

- **Unit Tests (`tests/unit/`)**: Test individual tools in isolation. Use `pytest` to assert that `tools.py` functions behave as expected with mock inputs. This ensures your business logic is correct.
- **Evaluation Tests (`eval/`)**: Test the end-to-end behavior of the agent. Use `AgentEvaluator` with `conversation.test.json` files to define multi-turn dialogues. This asserts that the LLM correctly understands intent, calls the right tools in the right order, and produces the desired final response.

Deployment

To deploy an agent to Vertex AI Agent Engine, you must package it. 1. **Build the Wheel**: `poetry build --format=wheel --output=deployment` 2. **Deploy**: Use a script (like `deployment/deploy.py`) to call `vertexai.agent_engines.create()`, passing the path to the generated `.whl` file in the `requirements` and `extra_packages` arguments.

Part 6: Troubleshooting - Common Pitfalls & Anti-Patterns

- **Anti-Pattern**: Hardcoding IDs, URLs, or prompts directly in `agent.py`.
  - **Fix**: Move all prompts to `prompts.py`. Move all configuration to a `config.py` file or environment variables.
- **Anti-Pattern**: A single, monolithic tool that tries to do too many things.
  - **Fix**: Break it down into smaller, single-purpose tools. This makes it easier for the LLM to choose the right one.
- **Anti-Pattern**: Assuming a tool call succeeded without checking its return value.
  - **Fix**: The agent's prompt must instruct it to check the result of a tool call (e.g., `{"status": "rejected"}`) and react accordingly.
- **Anti-Pattern**: Using global variables to share data between tool calls.
  - **Fix**: Use `tool_context.state` to pass data between turns.

- **Anti-Pattern**: The LLM hallucinates tool names or parameters.
    - **Fix**: Ensure your tool functions have crystal-clear names, docstrings, and type hints. The clearer they are, the less the LLM will hallucinate.