ADK Multi-Agent System

Best Practices

A deep-dive into the architectural, efficiency, and structural patterns for production-ready agents.

The Core Architecture

Understanding the rationale behind multi-agent systems and the role of delegation.

Three Pillars of ADK Modularity

1. Agents as Modules

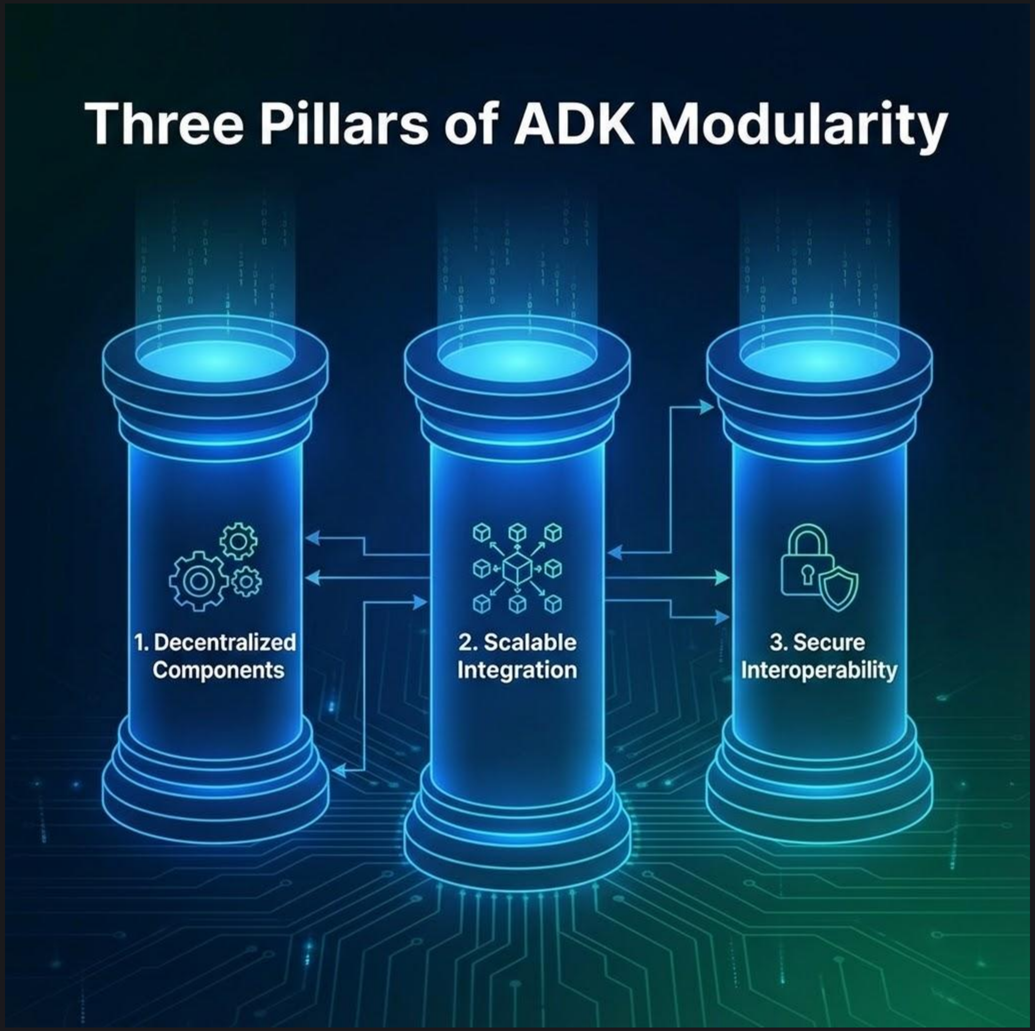
Each agent (super or sub) must be a self-contained Python package, enabling isolated development and ownership. This allows developers to work on different agents simultaneously.

2. Shared vs. Specific Tools

Clearly distinguish between generic tools (placed in a central `shared/` directory) and tools tightly coupled to a specific agent (placed within that agent's folder).

3. Explicit Interfaces

Define clear input/output contracts (data formats, expected states) between Super-Agents and their Sub-Agents to ensure reliable orchestration.


Three Pillars of ADK Modularity
1. Decentralized Components
2. Scalable Integration
3. Secure Interoperability

Agent Modularity: The Foundation of Collaboration

- **Isolate Codebase:** By making each agent a self-contained package, cross-package dependencies are minimized, reducing merge conflicts.
- **Clear Ownership:** A single developer or team can own a single agent directory, including its code, configuration, and agent-specific tools.
- **Easy Testing:** Allows for seamless unit testing of individual agent logic without needing to mock the entire multi-agent system.
- **CLI Discoverability:** ADK's CLI automatically discovers agents in directories containing an `__init__.py` and an `agent.py` file defining a `root_agent`.

Rationale: Why Use Multi-Agents?

- **Complexity Management:** Decompose a large, complex problem into smaller, manageable sub-tasks. This is difficult for a single, monolithic agent.
- **Reusability & Modularity:** Specialized agents (e.g., Research, Coding) can be reused across different applications without having to rewrite their logic or tools.
- **Higher Quality Output:** A "society of mind" where experts handle specific domains leads to better reasoning, fewer hallucinations, and more precise tool use.

LLM-Driven Delegation (Agent Transfer)

The Orchestrator's LLM dynamically routes tasks to a sub-agent using the `transfer_to_agent` mechanism based on the sub-agent's precise **`description`** field.

Clear and distinct descriptions are vital for effective dynamic routing.

Best Practice: Model Selection for Scale

Prioritize Flash

The sample uses `gemini-2.5-flash` for all agents. This is the best practice default for multi-agent systems to minimize latency and manage operational costs.

Cost & Latency

Since MAS involves multiple LLM calls per request, the overhead in tokens and time accumulates quickly. Flash is optimized for speed and cost-effectiveness in these scenarios.

When to Use Pro

Reserve larger models (e.g., Gemini 2.5 Pro) for single agents responsible for complex, non-latency-sensitive tasks like final synthesis or complex multi-step reasoning.

The Recommended Structure

A scalable, modular directory layout designed for the Agent Development Kit.

Project Structure & Organization

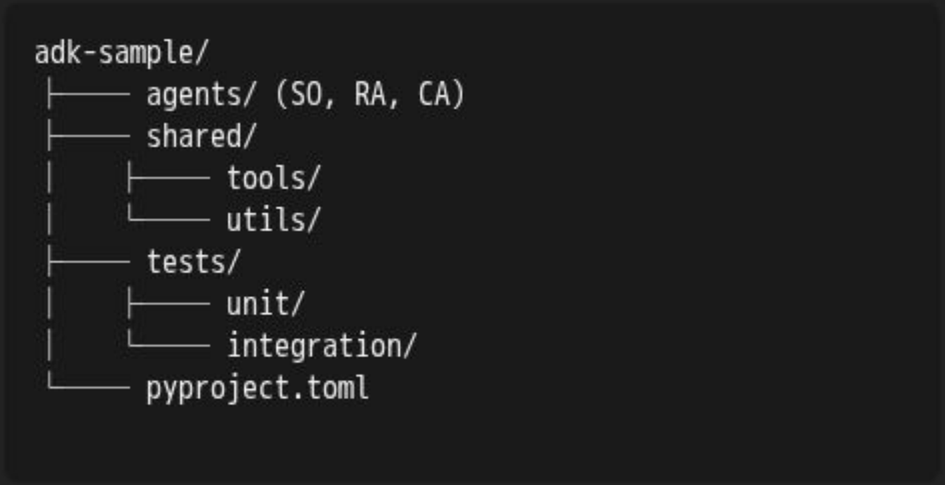
Clear Separation of Concerns

Agents: Dedicated directories for each agent's definitions and implementation logic (Orchestrator, Research, Coding).

Shared: Resources common to all agents, ensuring code reuse and a single source of truth for tools and utilities.

Tests: Separate suites for Unit tests (tool functions) and Integration tests (agent configurations).

The Hierarchy (Simplified)



```
adk-sample/
├── agents/ (SO, RA, CA)
├── shared/
│   ├── tools/
│   └── utils/
├── tests/
│   ├── unit/
│   └── integration/
└── pyproject.toml
```

Project Root Layout: A High-Level View

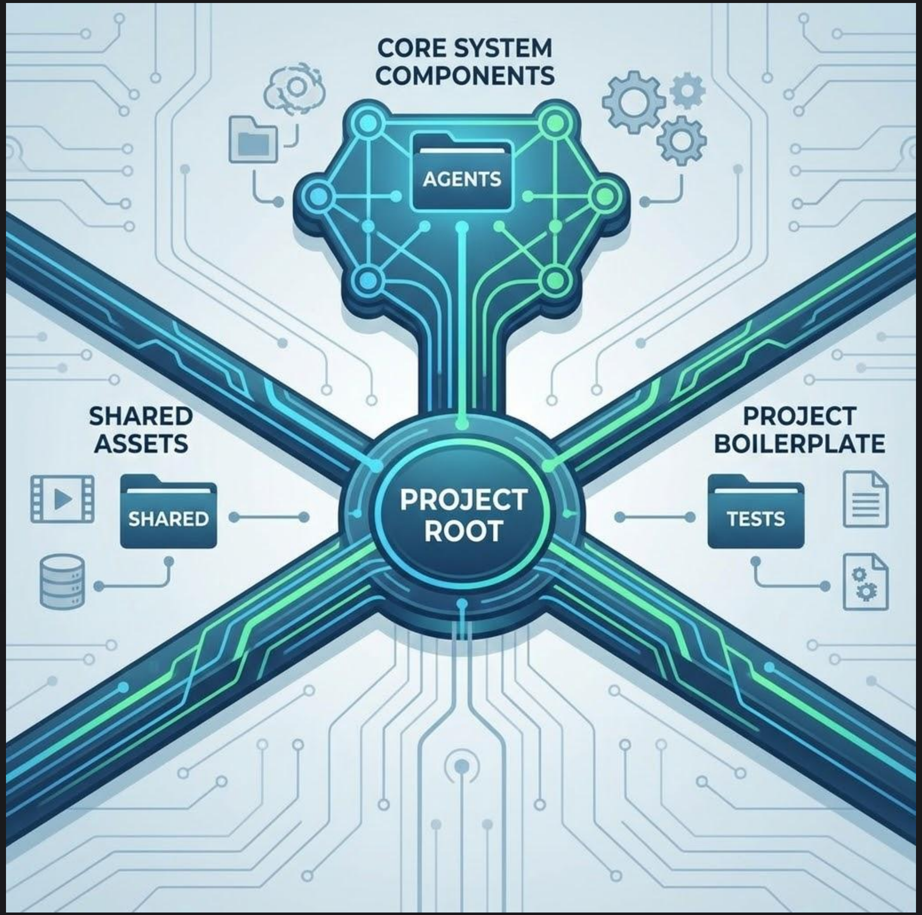
The Top-Level Breakdown

The project root acts as the central hub, clearly dividing core system components, shared assets, and project boilerplate.

Key Folders:

- `agents/`: All modular agents.
- `shared/`: Reusable tools and utilities.
- `tests/`: Mirrored structure for unit and integration testing.

Files: `.env`, `main.py`, `requirements.txt`.



Key Components: Agent Roles

Super Orchestrator

Top-level agent. Delegates tasks to sub-agents. Model: `gemini-2.5-flash`.

Research Subagent

Specialized in info gathering. Tools: `brave_search` (shared), `query_knowledge_base` (local).

Coding Subagent

Specialized in code analysis and linting. Primary tool is `run_linter` (local).

Best Practice: Unified Tool Architecture

- **Reduce LLM Workload:** Consolidate multiple small tool functions into a single, unified tool where appropriate (e.g., a "Code Analyzer" tool that runs linter, static analysis, and formatter internally).
- **Lower Cost:** Calling one comprehensive tool instead of three separate ones significantly reduces total LLM tokens used for reasoning, leading to measurable cost and latency savings.
- **Structured Output:** Unified tools can provide consistent, structured outputs that the LLM finds easier to parse, improving success rate even if the output is technically longer.

Best Practice: Context & State Management

- **Shared Session State:** Agents exchange information and maintain conversation flow via the shared `session.state`. This is the primary "knowledge sharing" mechanism.
- **Context Preservation:** The Orchestrator can inject relevant state (e.g., intermediate results, user tier) into the sub-agent's `InvocationContext` during transfer to maintain the thread of thought.
- **Scoped Storage:** ADK supports state scoping (`temp:`, `session:`, `user:`, `app:`) for managing data lifespan, which is critical for scalable, enterprise-ready systems.

Tool Use: Shared vs. Local

Shared Tools (`shared/tools/`)

- Multiple agents need the exact same functionality (e.g., search, time).
- Promotes **code reusability** and ensures a single source of truth.
- Best for generic utilities to simplify maintenance.

Local Tools (`agents/agent/tools.py`)

- The tool is specific to one agent's domain (e.g., `run_linter`).
- Tool logic is tightly coupled to the agent's role.
- Ensures **modularity** and keeps agent code highly self-contained.

The Foundation: Comprehensive Testing

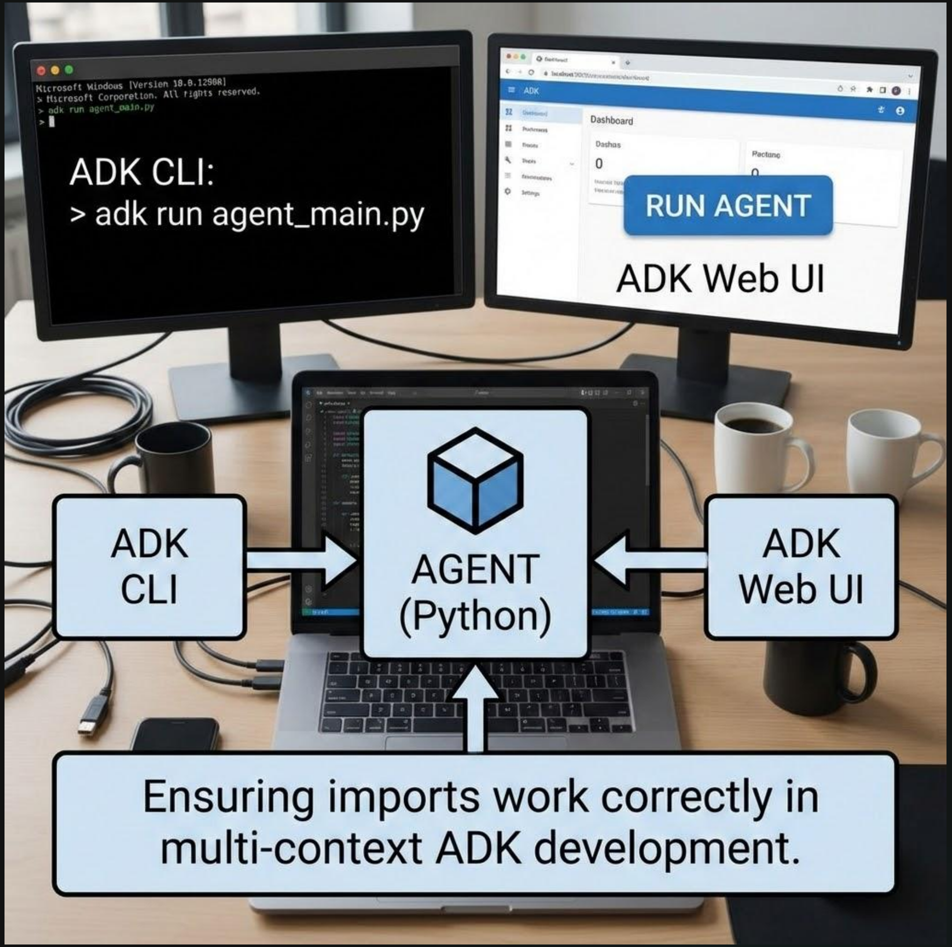
- **Unit Tests:** Verify individual tool functions in isolation, ensuring tool return values, behavior, and error handling are correct.
- **Integration Tests:** Verify agent configuration, ensuring the correct tools are attached and agent metadata (name, model) is set properly.
- **Command:** Run the entire test suite from the project root using: python3 -m unittest discover tests -v

Technical Best Practice: Robust Imports

Ensuring imports work correctly is critical in multi-context ADK development. Agents must be runnable via:

- **ADK CLI:** adk run agents/X
- **ADK Web UI:** adk web agents

The solution is to manage sys.path in agent files to ensure the project root is always resolvable.



```
Microsoft Windows [Versien 18.0.1290R]
> Microsoft Corporetion. All rights reserved.
> adk run agent_main.py
> 
```

Ensuring imports work correctly in multi-context ADK development.

Running the Agents

Run Individual Agents (CLI)

Run the top-level orchestrator agent directly from the command line, which automatically includes its sub-agents for delegation.

```
adk run agents/super_orchestrator
```

Run with Web UI

Launch all agents simultaneously with a user-friendly web interface for interaction and inspection (accessible at http://127.0.0.1:8000).

```
adk web agents
```

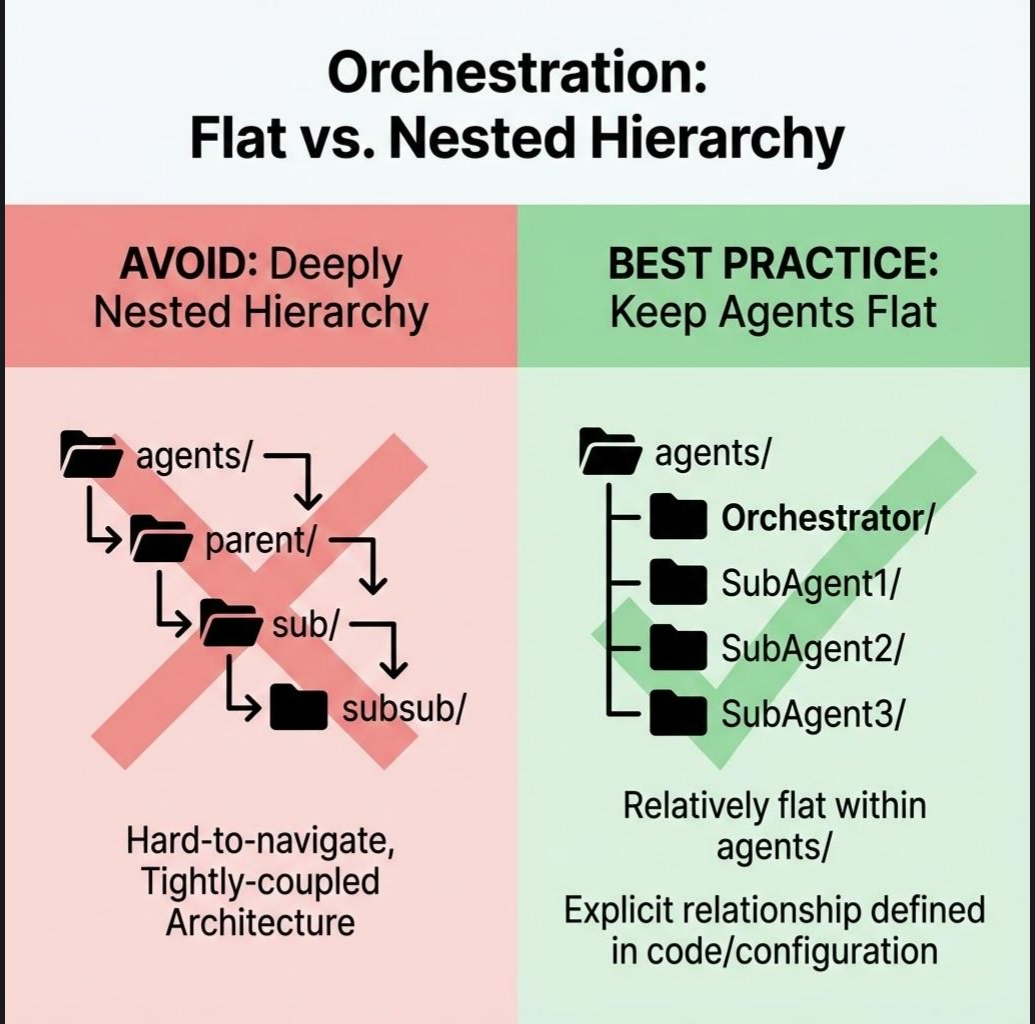
Orchestration: Flat vs. Nested Hierarchy

Keep the Agent Structure Flat

AVOID: Deeply nested folder structures like `agents/parent/sub/subsub/`. This leads to hard-to-navigate and tightly-coupled architectures.

BEST PRACTICE: Keep agents relatively flat within the `agents/` directory.

The relationship (e.g., Orchestrator calls Sub-agent) should be defined explicitly in the **code** (in the orchestrator `agent.py`) or via **configuration**, not implicitly by folder nesting.



Orchestration:
Flat vs. Nested Hierarchy
AVOID: Deeply Nested Hierarchy
BEST PRACTICE: Keep Agents Flat
agents/
parent/
sub/
subsub/
Hard-to-navigate, Tightly-coupled Architecture
agents/
Orchestrator/
SubAgent1/
SubAgent2/
SubAgent3/
Relatively flat within agents/
Explicit relationship defined in code/configuration

Parallel Development & Testing

- **Parallel Workstreams:** Developer A works solely in `agents/research_subagent/` while Developer B works in `agents/coding_subagent/`.

- **Minimal Cross-Paths:** Teams only cross paths when updating the contracts (interfaces) or modifying code within `shared/tools/`.

- **Isolated Testing:** Run unit tests for just one developer's agent without affecting the others: pytest tests/unit/agents//.

- **Scalability:** This modular approach ensures that adding new, specialized agents (e.g., a `deployment_subagent/`) does not introduce complexity to existing agents.

Questions?

Thank you for exploring the ADK Multi-Agent Best Practices.

Google ADK Documentation