GCP AGENTIC AI MASTERCLASS

Designing, Building & Orchestrating AI Agents

A complete guide to the Agent Development Kit (ADK) and Vertex AI.

DESIGN BUILD DEPLOY

Course Roadmap

1

Design

Architecture & Roles

2

Build

ADK Implementation

3

Orchestrate

Sequential vs Parallel

4

Route

Intelligent Flow

5

State

Session Mgmt

6

Tools

SQL & MCP

7

RAG

Vertex AI Search

Project

Banking System

The Tech Stack

Core Components

We leverage a modern, cloud-native stack.

- **Google ADK:** The Python-based Agent Development Kit.
- **Vertex AI:** Access to Gemini 1.5 Flash/Pro models.
- **MCP:** Model Context Protocol for tool interfaces.
- **Cloud SQL:** Persistent MySQL storage.

Hybrid Architecture

Deterministic Tools (SQL) + Probabilistic Reasoning (LLMs).

LESSON 01

Designing Multi-Agent Systems

Planning before Coding

Core Concepts

Why Multi-Agent?

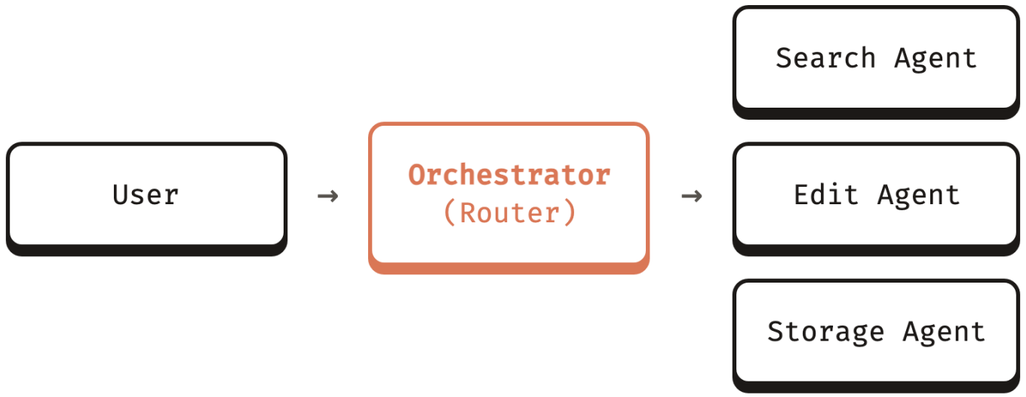
Single agents struggle with broad, complex contexts. Multi-agent systems break problems down into specialized units.

Strategy: Orchestrator & Workers

1. Orchestrator receives user intent.
2. Delegates to specialized Worker.
3. Worker executes and returns result.

Case Study: Google Photos

How a complex application is decomposed into agents.

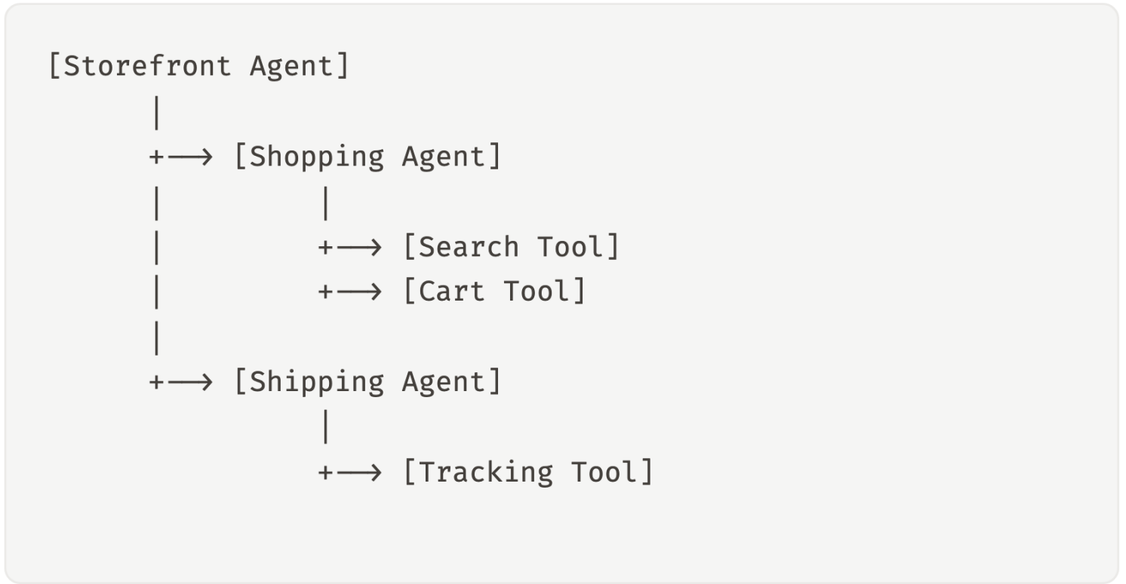


Exercise: E-Commerce Design

The Task

Design an agent system for an online store.

- **Storefront:** The greeter/router.
- **Shopping:** Product search & cart.
- **Inventory:** Stock checks.
- **Shipping:** Status & fulfillment.



LESSON 02

Implementation with ADK

From Diagrams to Python Code

Defining an Agent

The ADK Class

The `Agent` class wraps the model, prompts, and tools.

- `name`: Unique identifier.
- `model`: e.g., gemini-2.5-flash.
- `instruction`: The System Prompt.

```
from google.adk.agents import Agent

shipping_agent = Agent(
    name="shipping_agent",
    description="Handles shipping requests.",
    model="gemini-2.5-flash",
    instruction="You are a shipping agent...",
    tools=[place_order, track_package],
)
```

Delegation via Sub-Agents

```
root_agent = Agent(
    name="orchestrator",
    # ...
    sub_agents=[
        shipping_agent,
        inquiry_agent
    ]
)
```

How Delegation Works

The ADK automatically converts `sub_agents` into tools that the LLM can call.

> If the user asks "Where is my order?", the Orchestrator sees a tool named `delegate_to_shipping_agent` and calls it.

Demo: Shipping Orchestrator

Orchestrator

Receives request. Decides if it's an inquiry or a new order.

Inquiry Agent

Read-only access. Checks order status. Answers FAQs.

Shipping Agent

Write access. Places orders. Updates addresses.

LESSON 03

Orchestration Patterns

Sequential vs Parallel Execution

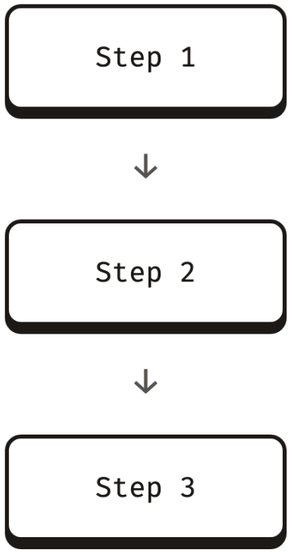
Sequential Orchestration

Dependency Chains

Use `SequentialAgent` when Step B requires data from Step A.

Example: "Add to Cart" workflow.

1. Get active order ID.
2. Check Inventory for item.
3. Add item to cart.



Parallel Orchestration

Speed & Efficiency

Use `ParallelAgent` when tasks are independent.

Example: Calculating Costs

- Calculate Shipping (based on weight).
- Calculate Taxes (based on location).

Code Implementation

Sequential

```
cart_agent = SequentialAgent(
    name="cart_workflow",
    sub_agents=[
        get_order_agent,
        check_inventory_agent,
        add_item_agent
    ]
)
```

Parallel

```
costs_agent = ParallelAgent(
    name="cost_calculator",
    sub_agents=[
        shipping_cost_agent,
        tax_calculator_agent
    ]
)
```

LESSON 04

Data Routing Strategies

Controlling the Flow of Information

Intent-Based Routing

The "Traffic Cop"

Standard routing uses an LLM to decide where to go based on the user's natural language request.

This is probabilistic. "I want to buy a car" → Maps to `Loan Agent`.

LlmAgent

The default router in ADK. Flexible, easy to set up, but can be non-deterministic.

Custom Routing Logic

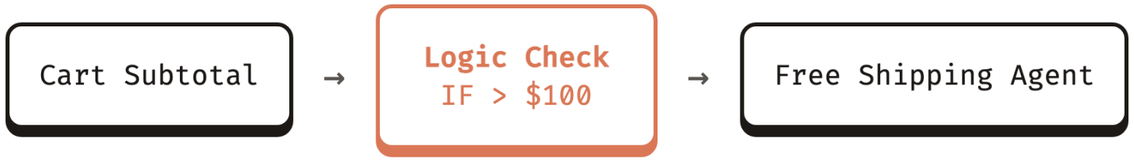
```
class ShippingRouter(BaseAgent):
    async def _run_async_impl(self, ctx):
        total = ctx.state.get("total")

        if total > 100:
            return self.free_shipping
        else:
            return self.standard_shipping
```

Deterministic Logic

Sometimes you need hard rules, not AI guesses.

- **Business Rules:** Free shipping thresholds.
- **A/B Testing:** Randomly routing 50% of users.
- **Security:** Restricting access based on tiers.

Demo: Free Shipping Threshold



LESSON 05

State Management

Memory, Context, and Persistence

The Invocation Context

The "Brain" of the Request

Every agent execution in ADK receives an `InvocationContext`.

- **Session ID:** Identifies the user conversation.
- **History:** Previous turns (chat log).
- **State Dictionary:** A shared key-value store for that session.

Types of State

Implicit: Chat history ("I want that one").
Explicit: Code variables (`ctx.state['id'] = 1`).

Implementing State Code

```
def get_order(tool_context: ToolContext):
    # Check if order_id exists in session state
    order_id = tool_context.state.get("order_id")

    if order_id is None:
        # Create new if missing
        order_id = generate_new_id()
        # Save to state for next turn
        tool_context.state["order_id"] = order_id

    return {"order_id": order_id}
```

Demo: Persistent Cart

Stateless

User: "Add apples"
Agent: "Added."
User: "Checkout"
Agent: "Checkout what?"

Stateful (ADK)

User: "Add apples"
Agent: "Added." [State: cart=['apples']]
User: "Checkout"
Agent: "Checking out 1 item: apples."

LESSON 06

Tools & MCP

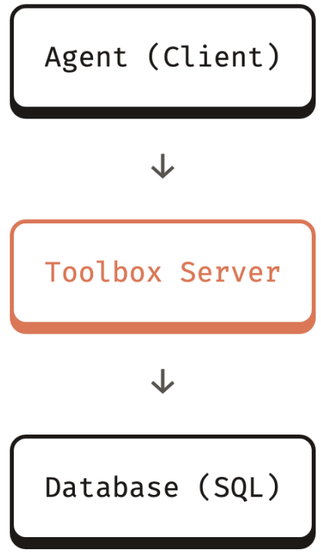
Connecting AI to the Real World

Model Context Protocol (MCP)

The Bridge

MCP is a standard for connecting AI models to data sources and tools.

Instead of hardcoding Python functions inside the agent, we define them in a server (Toolbox) and the agent connects as a client.



Defining Tools: tools.yaml

```
tools:
  get-balance:
    kind: mysql-sql
    source: deposit
    description: Get current balance
    parameters:
      - name: account_name
        type: string
    statement: SELECT bal FROM accounts WHERE name = ?
```

Declarative Safety

We define tools in YAML.

- **Safe:** Parameterized SQL prevents injection.
- **Portable:** Can be used by different agents.
- **Clear:** Descriptions help the LLM know *when* to use it.

Structured Outputs with Pydantic

Strict Schemas

Ensure the LLM returns data that code can understand, not just chatty text.

This forces Gemini to return valid JSON.

```
class InventoryData(BaseModel):
    product_id: str
    in_stock: bool
    count: int

agent = LlmAgent(
    ...,
    output_schema=InventoryData
)
```

LESSON 07

Retrieval Augmented Generation

Handling Unstructured Data

Vertex AI Search

Vector Search

Databases (SQL) are good for numbers. Vector Search is good for **meaning**.

It converts text (PDFs, Manuals) into mathematical vectors (embeddings) to find "concepts" rather than just keywords.

Capabilities

- Ingest PDFs/HTML.
- Semantic Search.
- Auto-summarization.

Multi-Agent RAG Strategy

1. User Query

"Do you have the waterproof speaker in stock?"

2. RAG Agent

Searches Manuals: "Yes, Product P003 is waterproof."

3. SQL Agent

Checks DB: "P003 Count: 0"

RAG Tool Implementation

```
def datastore_search_tool(search_query: str):
    client = discoveryengine.SearchServiceClient()
    # Query Vertex AI Data Store
    response = client.search(
        serving_config=config_id,
        query=search_query
    )
    return response.results
```

This tool is given to the `product_qa_agent`, allowing it to "read" the documents.

DEEP DIVE

Complete Execution Flow

"Add wireless headphones to my cart"

The Six Phases of Execution

1

Initialization

Request handling & orchestrator setup

2

Search

Product discovery via MCP tools

3

Cart Prep

Parallel order & inventory checks

4

Add to Cart

Sequential item addition

5

Response

Event aggregation & streaming

6

Persistence

State & session management

PHASE 1

Initialization & Request Handling

From user input to orchestrator decision

Step 1-2: Request & Root Agent

User Request Submission

Input: "Add wireless headphones to my cart"

Request received by ADK runtime and wrapped in an `InvocationContext` object.

Root Agent Instantiation

```
shopping_orchestrator = Agent(
    name="shopping_orchestrator",
    model="gemini-2.5-flash",
    instruction="agent-prompt.txt",
    sub_agents=[
        search_agent,
        inventory_agent,
        cart_agent,
        product_qa_agent
    ]
)
```

Step 3: Orchestrator Reasoning

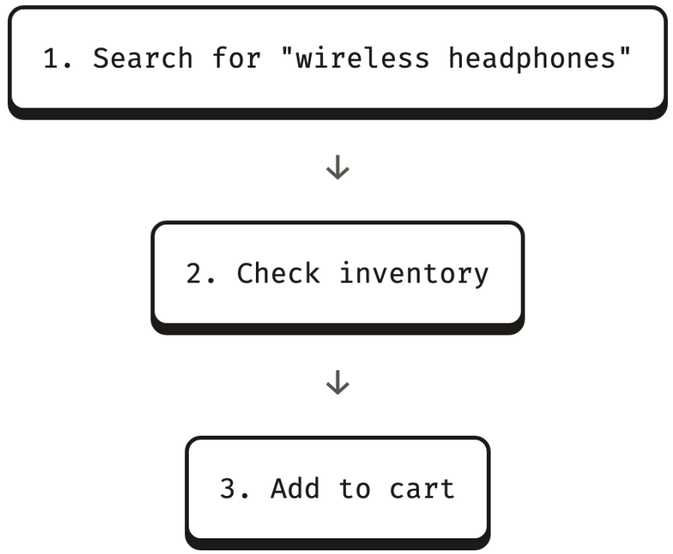
LLM Decision Process

Input to Gemini-2.5-Flash:

- User message
- Orchestrator instructions
- Available sub-agents list

Analysis: User wants to "Add...to cart"

Determined Workflow



PHASE 2

Search Agent Execution

A/B Testing Router → MCP Toolbox → MySQL

Step 4-5: SearchRouter (A/B Testing)

Custom Router Logic

```
class SearchRouter(BaseAgent):
    agent_a = search_agent_exact  # Phrase match
    agent_b = search_agent_broad  # Any-word match
    agent_b_rate = 0.5  # 50% probability

    async def _run_async_impl(self, ctx):
        if random.random() < (1 - self.agent_b_rate):
            return self.agent_a
        else:
            return self.agent_b
```

Probabilistic Routing

50% → Exact Search
50% → Broad Search

Useful for testing which approach performs better.

Step 6–7: Search Agent & MCP Toolbox

Agent Instruction: "Your job is to help users find products based on their queries..."

Tool Loaded: `search-products` from `tools.yaml`

Step 8–9: Database Query Execution

Tool Definition (tools.yaml)

```
search-products:
  kind: mysql-sql
  source: storefront
  statement: |
    SELECT * FROM products
    WHERE CONCAT(name, ' ', description)
    LIKE CONCAT('%', ?, '%')
```

Query Result

```
| product_id | name                | price  |
├────────────┼─────────────────────┼────────┤
| P001       | Wireless Headphones| 299.99 |

description: "Noise cancelling
              over-ear headphones"
```

Step 10-12: Response Generation

Tool Result → LLM

```
{
  "product_id": "P001",
  "name": "Wireless Headphones",
  "description": "Noise cancelling ...",
  "price": 299.99
}
```

Agent Output

"I found **Wireless Headphones** - Noise cancelling over-ear headphones at **$299.99**. Would you like to add it to your cart?"

Response wrapped in `Event` object and yielded back to orchestrator.

PHASE 3

Cart Agent Execution

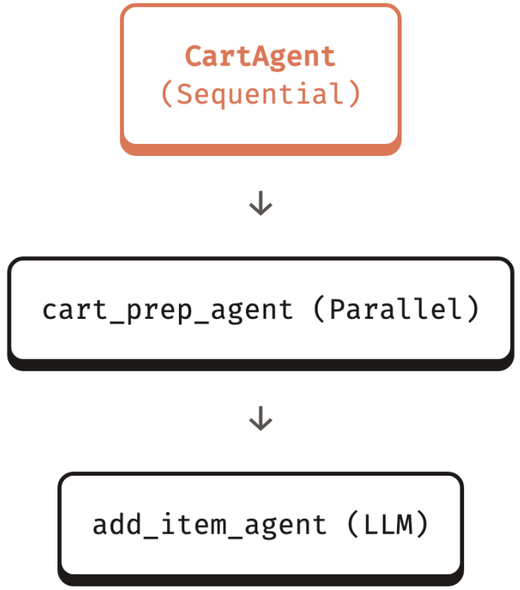
SequentialAgent → ParallelAgent → Concurrent Tasks

Step 13-14: SequentialAgent Workflow

Cart Agent Structure

```
cart_agent = SequentialAgent(
    name="cart_agent",
    sub_agents=[
        cart_prep_agent,   # Phase 1
        add_item_agent     # Phase 2
    ]
)
```

Not an LlmAgent! This is deterministic sequential execution.



Step 15: CartPrepAgent (Parallel)

```
cart_prep_agent = ParallelAgent(
    name="cart_prep_agent",
    sub_agents=[
        get_order_agent,      # Get/create order
        inventory_data_agent  # Check stock
    ]
)
```

Why Parallel?

These tasks are **independent**:

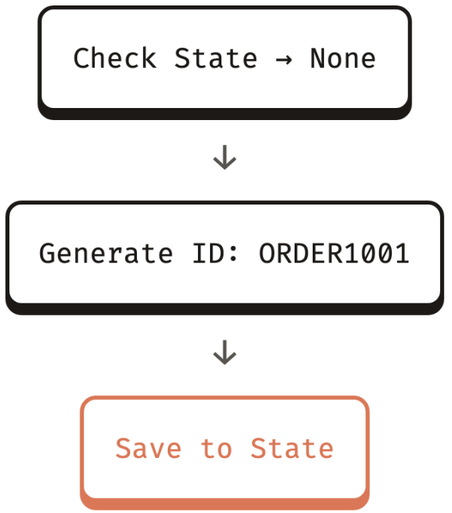
- Creating an order doesn't need inventory data
- Checking inventory doesn't need order ID

Performance Win: Both execute simultaneously!

Step 15a: GetOrderAgent (Track A)

```
def get_order(tool_context: ToolContext):
    # Check if order exists
    order_id = tool_context.state.get("order_id")

    if order_id is None:
        # Create new order
        order_id = get_next_order_id()  # "ORDER1001"
        tool_context.state["order_id"] = order_id

        # In-memory storage
        orders[order_id] = {
            "cart": [],
            "address": None,
            "order_status": None
        }

    return {"order_id": order_id}
```

State Flow



Step 15b: InventoryDataAgent (Track B)

Structured Output Schema

```
class InventoryData(BaseModel):
    product_id: str
    in_stock: bool
    count: int

inventory_data_agent = LlmAgent(
    name="inventory_data_agent",
    output_schema=InventoryData,
    tools=[check_inventory]
)
```

SQL Query & Result

```
-- check-inventory tool
SELECT
  product_id,
  quantity > 0 as in_stock,
  quantity as count
FROM inventory
WHERE product_id = 'P001'

-- Result:
{
  "product_id": "P001",
  "in_stock": true,
  "count": 50
}
```

Step 15c: Parallel Completion

Track A Complete
`Order: ORDER1001`

Track B Complete
`P001: 50 units in stock`

Synchronization Point: Both results merged into `InvocationContext` for the next step.

PHASE 4

Add to Cart Execution

Tool invocation with state management

Step 16-17: AddItemAgent Execution

Agent Configuration

```
add_item_agent = LlmAgent(
    name="add_item_agent",
    instruction="""
    Your job is to add a product to cart.
    Assume order exists and inventory
    has been checked.

    If sufficient inventory, use the
    add_to_cart tool. If not, alert
    the customer.
    """,
    tools=[add_to_cart]
)
```

Context Available

- **From Search:** product_id = "P001"
- **From Track A:** order_id = "ORDER1001"
- **From Track B:** in_stock = true, count = 50

Agent has full context to make decision → **Proceed with add_to_cart**

Step 18-19: add_to_cart Function

```
def add_to_cart(product_id: str, tool_context: ToolContext):
    # Step 19.1: Extract order_id from context
    order_id = tool_context.state.get("order_id")  # Returns "ORDER1001"

    # Step 19.2: Validate order exists
    order = orders[order_id]  # Retrieves from in-memory store

    # Step 19.3: Check order status
    if order["order_status"] is not None:
        return {"error": "Order cannot be modified..."}

    # Step 19.4: Add product to cart array
    order["cart"].append(product_id)  # cart = ["P001"]

    # Step 19.5: Return success response
    return {
        "status": "success",
        "message": f"Added {product_id} to cart.",
        "cart": order["cart"]  # ["P001"]
    }
```

PHASE 5

Response Aggregation

Event collection and streaming

Step 21-22: Event Collection

1

Search

"I found Wireless Headphones..."

2

Order

"Order created: ORDER1001"

3

Inventory

"P001 in stock: 50 units"

4

Cart

"Successfully added..."

Events Queue: All responses stored in `invocation_context.session.events`

Streaming: Events yielded back through async generator in sequence of execution.

Step 23-24: Final User Response

Combined Output

Search Result: I found Wireless Headphones - Noise cancelling over-ear headphones at $299.99

Order Status: Order created with ID ORDER1001

Inventory: P001 in stock (50 units available)

Cart Update: Successfully added Wireless Headphones to your cart!

Final State

```
# Session State
{
  "order_id": "ORDER1001"
}

# In-Memory Orders
orders["ORDER1001"] = {
  "cart": ["P001"],
  "address": None,
  "order_status": None
}
```

PHASE 6

State Persistence

Session continuity for future requests

Step 25-26: Future Invocation Handling

Next User Request

User: "Add USB-C cable"

GetOrderAgent checks:

```
order_id = tool_context.state.get("order_id")
# Returns "ORDER1001" (NOT None!)

# Reuses existing order
# No new order created
```

Updated Cart

```
# Before
orders["ORDER1001"]["cart"] = ["P001"]

# After adding USB-C cable (P004)
orders["ORDER1001"]["cart"] = [
    "P001",  # Wireless Headphones
    "P004"   # USB-C Cable
]
```

Cart persists across requests!

SUMMARY

Key Technical Layers

The architecture that powers it all

The Four-Layer Architecture

ADK Framework

- `InvocationContext`
- `ToolContext`
- `Event` objects

Agent Layer

- `LlmAgent`
- `SequentialAgent`
- `ParallelAgent`
- `SearchRouter`

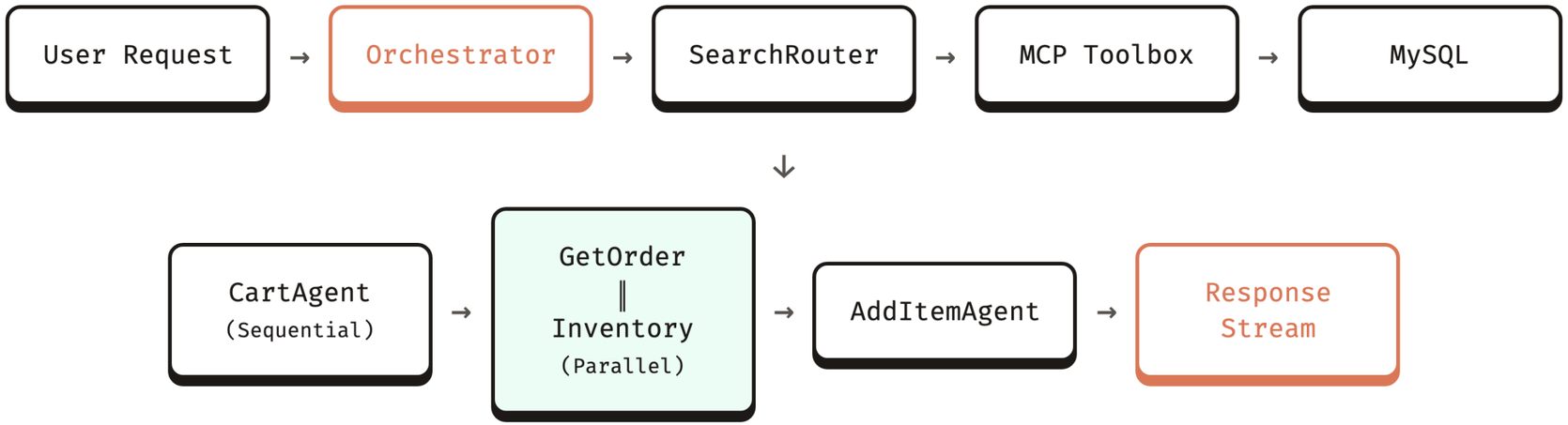
Tool Layer (MCP)

- `ToolboxSyncClient`
- SQL tools in YAML
- Pydantic schemas

Data Layer

- MySQL Database
- Vertex AI Search
- In-Memory State

Complete Execution Flow



FINAL PROJECT

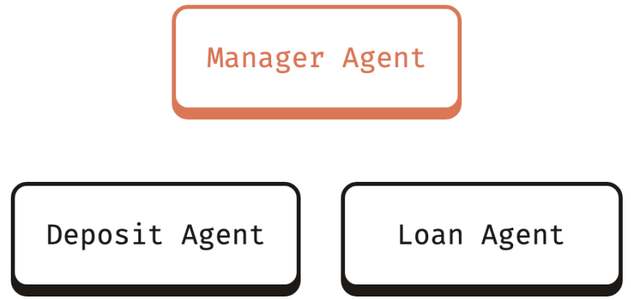
Intelligent Banking System

Putting it all together

System Architecture

Three Specialized Agents

- **Manager Agent:** The router/greeter.
- **Deposit Agent:** Handles checking/savings.
- **Loan Agent:** Handles mortgages/auto loans.



The Deposit Agent

Responsibilities

- Check Balances.
- List Transactions.
- Verify Minimum Balances.

```
tools:
  - get-accounts
  - get-balance
  - check-minimum-balance (SQL SUM query)
  - get-transactions
```

The Loan Agent: Complex Logic

1

Collect Data

Ask user for Amount and Type (Auto, Home).

2

Check Policy

Read `loan-policy.pdf` via RAG to get rules.

3

Decision

Compare User Data vs Policy Rules → Approve/Deny.

Flow: Applying for a Loan

User: "I want a car loan for $20k"
Manager: Routes to Loan Agent.
Loan Agent: Checks existing debt (SQL).
Loan Agent: Checks policy manual (RAG).
Loan Agent: "Your debt ratio is too high based on policy. Denied."

Project Code Structure

```
project/solution/
├── manager/
│   ├── agent.py
│   └── agent-prompt.txt
├── deposit/
│   ├── agent.py
│   └── tools.yaml (SQL)
└── loan/
    ├── agent.py
    ├── loan.py (Logic)
    └── tools.yaml
```

Key Project Takeaways

Orchestration

Coordinating distinct agents (Manager → Loan) is powerful.

Data Grounding

Real-world agents need real data (SQL) and policies (PDFs).

Course Summary

- **Design First:** Always diagram your agents.
- **Use the ADK:** It handles the glue code (HTTP, Context).
- **Tools are Key:** Agents are only as good as their tools (MCP).
- **State Matters:** Remember the user's cart/session.
- **Hybrid Data:** Combine Vector Search + SQL for complete answers.

Thank You!

Now go build intelligent agents.