Intro to Agents and ADK

Yogesh Haribhau Kulkarni

Outline

1. Introduction

2. Implementation

YHK

Introduction

What Makes an AI Agent Different?

- Unlike LLMs that just respond to prompts, agents are autonomous
- Can look at their environment and analyze the situation
- Make comprehensive plans to achieve specific goals
- Actually take action to execute those plans
- Agents bridge the gap between answering and doing

Welcome to AI Agents

- AI agents represent one of the most exciting frontiers in AI
- Not just everyday chatbots - systems that reason, plan, and take action
- Move beyond AI that just answers questions to AI that does things
- Can take on complex multi-step tasks autonomously
- The core promise: AI that accomplishes goals independently
- Technology is advancing rapidly from conversational to agentic AI

Meet Suresh - The Impossible Job

- Suresh's boss tasks him with planning a massive get-together
- Must research a huge guest list and plan fancy menu
- Needs to find entertainment for the event
- A simple chatbot cannot handle these complex requirements
- Suresh needs an AI agent - not just responses, but actions
- Perfect example of why we need more than conversational AI

Introduction to AI Agents

- 2025 is expected to be the year of AI agents.
- AI agents combine multiple components to solve complex problems.
- Shifting from monolithic models to compound AI systems.
- Compound AI systems use system design for better problem solving.
- AI agents improve with reasoning, acting, and memory components. (ReAct = Reasoning + Acting)

The Evolution of AI Capabilities

- **Traditional Programming:** Needed code to operate
- **Traditional ML:** Needed feature engineering
- **Deep Learning:** task-specific model
- **ChatGPT (2022):** Many tasks single model
  - Zero-shot learning (no examples needed)
  - In-context learning (understands from instructions)
- **Agents (2024 . . .):** Can actually **do things**, not just talk

Why Does "Taking Action" Matter?

- In 2022, ChatGPT was revolutionary because AI felt conversational
- By 2024, people wanted more than conversation, they wanted **execution**
- Examples of what users now expect:
  - Instead of listing leads ? **email them directly**
  - Instead of summarizing docs ? **file and create workflow tasks**
  - Instead of suggesting products ? **customize landing pages**
- This shift from **information** to **action** defines the agent era

How Agents Work?

- Agent acts, take you from one state to the other state, provides value by workflow automation. (ReAct paper: Reasoning and Action), it can plan and make decisions.
- Agents have access to tools (ToolFormer paper) e.g. Search APIs, booking, send email etc.
- Interacting of external environment and other Agents, etc.
- Memory to keep the history of conversations/actions done so far.
- May have human-in-loop to keep it sane in the wild-world.
- Agents were there from 1950's but they are effective because of LLMs.
- Agents are systems where LLMs dynamically direct their own processes and tool usage
- Can operate autonomously over extended periods using various tools
- Distinct from workflows: agents have dynamic control vs. predefined code paths
- Essential component in modern AI systems with varying degrees of autonomy

The Agent's Fundamental Game Loop

- Not a one-and-done action, but a continuous reasoning loop
- Similar to a programming while loop that keeps iterating
- **Thought**: Analyzes situation and plans next step
- **Action**: Calls specific tools to execute the plan
- **Observation**: Examines results of the action taken
- Cycle repeats: Thought → Action → Observation
- Continues until the task is completely accomplished
- This loop enables continuous adaptation and problem-solving

Agent's Inner Monologue

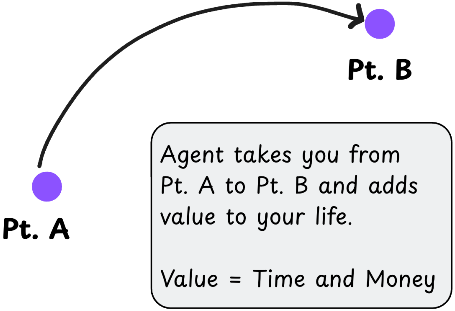
- Agents have visible thought processes before taking action
- Example: "User wants weather in New York. I have a tool for that."
- "My first move is to call the weather API"
- Internal planning step makes agents more than reactive programs
- Reasoning through problems before execution
- This deliberation distinguishes agents from simple scripts
- Shows intelligent decision-making rather than blind execution

How Do Agents Take Action?

- The magic lies in **tools** and **function calling**
- Agents are paired with APIs, plugins, or external systems
- Instead of just text responses, LLMs output structured commands:
  - ”Call the send_email() function with these inputs...”
  - ”Fetch records from CRM using this query...”
  - ”Schedule a meeting for Tuesday at 2PM...”
- **Mental model:** LLM = brain, Tools = hands
- Without tools, agents just talk. With tools, they act.

Defining AI Agents with an example

- Planning a trip involves many complex tasks
- Point A: Just discussing the trip
- Point B: All bookings and itinerary ready
- AI Agents aim to take you from A to B
- First idea: Agent adds value by saving time/money



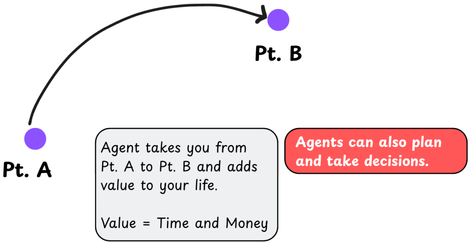
Agent takes you from Pt. A to Pt. B and adds value to your life.

Value = Time and Money

(Ref: Vizuara AI Agents Bootcamp)

Evolving Definition of Agents

- Not all tools from A to B are agents (e.g., cars)
- Agents must plan and make decisions
- Second definition includes decision-making ability
- Example: Choosing flights based on budget
- Planning daily itinerary needs contextual judgment



Agent takes you from Pt. A to Pt. B and adds value to your life.

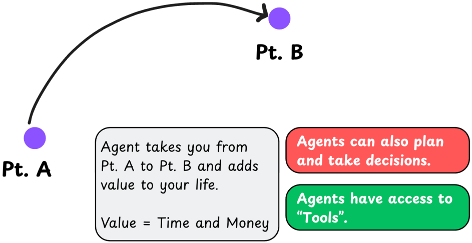
Value = Time and Money

Agents can also plan and take decisions.

(Ref: Vizuara AI Agents Bootcamp)

Agents Need Tools

- Even self-driving cars plan but are not agents
- Agents need access to external tools
- Tools = Access to services (e.g., Gmail, Booking)
- Agents perform tasks using these tools
- Third definition adds tool access to capabilities



Agent takes you from Pt. A to Pt. B and adds value to your life.

Value = Time and Money

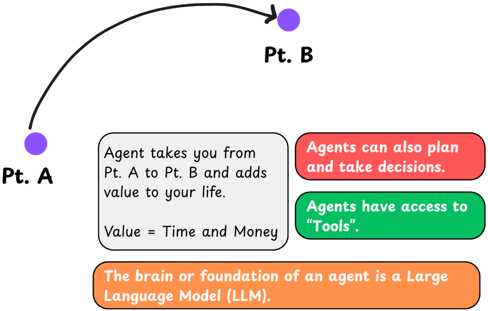
Agents can also plan and take decisions.

Agents have access to "Tools".

(Ref: Vizuara AI Agents Bootcamp)

Rise of LLMs in Agents

- Transformers (2017) enabled powerful LLMs
- LLMs understand and generate human language
- Agents use LLMs for reasoning and planning
- LLMs enable understanding of webpages and writing emails
- Fourth definition: Agents are LLMs with tools and planning ability



Agent takes you from Pt. A to Pt. B and adds value to your life.

Value = Time and Money

Agents can also plan and take decisions.

Agents have access to “Tools”.

The brain or foundation of an agent is a Large Language Model (LLM).

(Ref: Vizuara AI Agents Bootcamp)

What Is an Agent? (Technical Definition)

- Agent acts and takes you from one state to another, providing value through workflow automation
- Based on ReAct paradigm: **Reasoning + Acting**
- Key capabilities:
  - Can plan and make decisions
  - Has access to tools (search APIs, booking, email, etc.)
  - Interacts with external environments and other agents
  - Maintains memory of conversations and actions
  - May include human-in-the-loop for safety
- Agents existed since the 1950s but are now effective because of LLMs

Two Ways to Define Agents

Technical View:

- LLM (brain)
- + Tools (hands)
- + Planning (strategy)
- + Memory (context)
- + State management

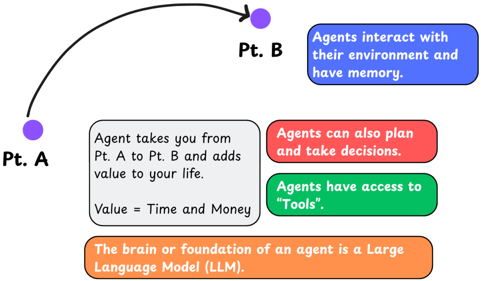
Business View:

- Systems that complete tasks end-to-end
- Focus on outcomes, not components
- Solve real-world problems
- Provide measurable value

Important: Today's agents are **engineering wrappers** around AI models, the intelligence comes from the LLMs, agents help act on that intelligence.

Final Definition of Agents

- Agents can learn from feedback and environment
- Agents interact with tools, humans, and websites
- They improve with experience (memory)
- Fifth definition: LLMs + Tools + Planning + Learning
- Agents evolve over time via memory and feedback



Agents interact with their environment and have memory.

Agent takes you from Pt. A to Pt. B and adds value to your life.

Value = Time and Money

Agents can also plan and take decisions.

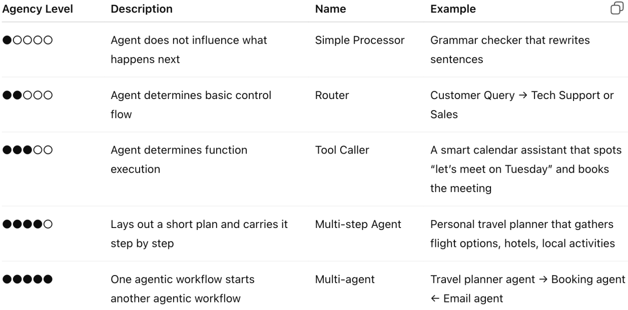
Agents have access to “Tools”.

The brain or foundation of an agent is a Large Language Model (LLM).

(Ref: Vizuara AI Agents Bootcamp)

Understanding Agency

- Agency = Level of autonomy an agent has
- Low agency → less value
- High agency → high value
- More autonomous agents can handle complex tasks
- Agency is key to measuring agent usefulness



Agency Level	Description	Name	Example
●○○○○	Agent does not influence what happens next	Simple Processor	Grammar checker that rewrites sentences
●●○○○	Agent determines basic control flow	Router	Customer Query → Tech Support or Sales
●●●○○	Agent determines function execution	Tool Caller	A smart calendar assistant that spots "let's meet on Tuesday" and books the meeting
●●●●○	Lays out a short plan and carries it step by step	Multi-step Agent	Personal travel planner that gathers flight options, hotels, local activities
●●●●●	One agentic workflow starts another agentic workflow	Multi-agent	Travel planner agent → Booking agent ← Email agent

(Ref: Vizuara AI Agents Bootcamp)

Tools: The Agent's Hands

- LLM is the agent's brain; tools are its hands
- Tools are functions agents call to interact with the world
- Can search web, run calculations, or query databases
- Bridge between thinking and doing in the real world
- Enable agents to move from planning to execution
- Without tools, agent thoughts would be useless
- Tools provide the interface to external systems and data

Two Ways Agents Use Tools

- **JSON Agent**: Writes structured work orders for other systems
- JSON approach requires external system to read and execute
- **Code Agent**: Directly writes and runs code blocks
- Code approach is more direct and powerful
- Code is naturally more expressive than JSON
- Can handle complex logic like loops and conditionals
- Modular, easier to debug, and taps into existing libraries
- Code agents can access thousands of APIs directly

Code Agent in Action

- Alfred needs a gala menu - agent has ”suggest_menu” tool
- Agent doesn’t just make up suggestions randomly
- Generates and runs actual code to call the specific tool
- Gets real results from the tool execution
- Super direct, efficient, and powerful way to take action
- Code generation enables precise tool interaction
- Results are based on actual tool capabilities, not hallucination

Advanced Pattern: Agentic RAG

- Traditional RAG: Retrieval Augmented Generation fetches info before answering
- Agentic RAG supercharges this with intelligent multi-step processes
- Turns retrieval itself into an agent-driven task
- Like having a master researcher on staff
- Doesn't just do one search - runs complete research processes
- Rewrites queries for better results and runs multiple searches
- Uses findings to inform next searches and validates accuracy
- Pulls from both private data and public web sources

Multi-Agent Systems: Digital Teams

- Complex problems like finding the missing Batmobile need teams
- Single agents can't handle web searches, calculations, and visualization
- Solution: Build teams of specialized agents
- Manager agent acts as project lead breaking down big tasks
- Delegates work to specialist agents with specific skills
- Web agent handles online searching while manager coordinates
- Manager focuses on big picture and final integration
- Digital division of labor for complex problem solving

The GAIA Benchmark Reality Check

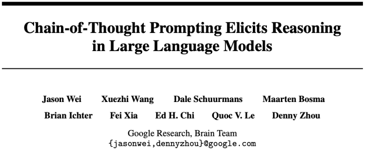
- GAIA benchmark tests real-world multi-step problems
- Measures how well systems handle tricky, complex tasks
- Results are eye-opening and show current limitations
- Humans solve these tasks with 92% accuracy
- Today's most advanced AI models: only 15% accuracy
- Massive 77% gap between human and AI performance
- This gap is exactly what agentic systems aim to close
- Shows the enormous potential for improvement

The Fundamental Shift

- Moving from conversational AI to agentic AI era
- Old paradigm: Ask questions, get answers
- New paradigm: State goals, systems plan and accomplish them
- Represents fundamental change in human-computer interaction
- Technology for building personal AI assistants advancing rapidly
- Not a question of ”if” but ”when” this becomes reality
- The future Alfred is closer than we think
- Prepare for AI that can handle ”impossible” complex tasks

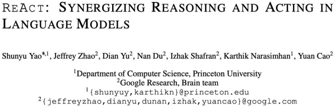
Papers that Shaped AI Agents

- Core research papers laid the foundation
- Introduced key frameworks and architectures
- Sparked recent boom in agent development
- Include Transformer and Agentic frameworks
- Major driving force in LLM-based agent systems



Chain-of-Thought Prompting Elicits Reasoning in Large Language Models

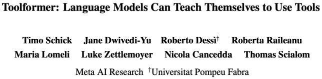
Jason Wei Xuezhi Wang Dale Schuurmans Maarten Bosma
Brian Ichter Fei Xia Ed H. Chi Quoc V. Le Denny Zhou
Google Research, Brain Team
{jasonwei,dennyzhou}@google.com



REACT: SYNERGIZING REASONING AND ACTING IN LANGUAGE MODELS

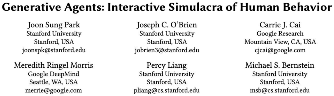
Shunyu Yao*[,1], Jeffrey Zhao[2], Dian Yu[2], Nan Du[2], Izhak Shafran[2], Karthik Narasimhan[1], Yuan Cao[2]

[1]Department of Computer Science, Princeton University
[2]Google Research, Brain team
[1]{shunyuy,karthikn}@princeton.edu
[2]{jeffreyzhao,dianyu,dunan,izhak,yuancao}@google.com



Toolformer: Language Models Can Teach Themselves to Use Tools

Timo Schick Jane Dwivedi-Yu Roberto Dessì† Roberta Raileanu
Maria Lomeli Luke Zettlemoyer Nicola Cancedda Thomas Scialom
Meta AI Research †Universitat Pompeu Fabra



Generative Agents: Interactive Simulacra of Human Behavior

Joon Sung Park
Stanford University
Stanford, USA
joonspk@stanford.edu

Joseph C. O'Brien
Stanford University
Stanford, USA
jobrien3@stanford.edu

Carrie J. Cai
Google Research
Mountain View, CA, USA
cjcai@google.com

Meredith Ringel Morris
Google DeepMind
Seattle, WA, USA
merrie@google.com

Percy Liang
Stanford University
Stanford, USA
pliang@cs.stanford.edu

Michael S. Bernstein
Stanford University
Stanford, USA
msb@cs.stanford.edu

(Ref: Vizuara AI Agents Bootcamp)

When to Use Agents?

- Best suited for tasks requiring flexibility and model-driven decision-making
- Consider tradeoffs: agents increase latency and cost for better task performance
- Recommended for open-ended problems with unpredictable steps
- Simple solutions preferred - single LLM calls with retrieval often sufficient

Future AI Applications

- What are future AI applications like?
  - **Generative:** Generate content like text and images
  - **Agentic:** Execute complex tasks on behalf of humans
- How do we empower every developer to build them?
  - **Co-Pilots:** Human-AI collaboration
  - **Autonomous:** Independent task execution
- 2024 is expected to be the year of AI agents

The Big Question

- Agentic AI technology is moving incredibly fast
- Personal AI assistants will soon be capable of complex tasks
- Think beyond simple queries to multi-step accomplishments
- Consider what ”impossible” tasks you want to delegate
- What complex, party-of-the-century level challenge will you tackle?
- The era of AI that truly does rather than just discusses
- Prepare for AI assistants that can handle your biggest challenges

Google ADK (Agent Development Kit)

Introduction to ADK Framework

- ADK (Agent Development Kit) is Google's framework for building sophisticated AI agents and agentic applications
- Official documentation available at: `https://google.github.io/adk-docs/`
- Designed for production-ready agentic systems with focus on scalability and reliability
- Supports multi-agent systems, tool integration, and flexible deployment options
- Built on Google's Gemini models with support for multiple LLM providers
- Framework emphasizes type safety, async operations, and enterprise-grade features

Key Components

- **Agents:** Core building blocks with instructions, tools, and model configuration
- **Tools:** Functions that agents can call to interact with external systems and APIs
- **Sessions:** Manage conversation context and state across multiple interactions
- **Runners:** Execute agent logic with support for streaming and async operations
- **Memory:** Store and retrieve conversation history and context
- **Multi-Agent Support:** Coordinate multiple specialized agents working together
- **Deployment:** Built-in support for FastAPI and serverless deployment

ADK Architecture Levels

- **Basic Agents:** Single agent with tools and instructions
- **Stateful Agents:** Agents with session management and memory
- **Multi-Agent Systems:** Multiple specialized agents collaborating
- **Agentic Workflows:** Complex orchestration with control flow
- **Production Systems:** Deployed services with monitoring and scaling
- Each level provides additional capabilities for building sophisticated AI applications

Key Features - Model Support & Flexibility

- **Gemini Integration:** Native support for Google's Gemini models (1.5 Pro, 2.0 Flash)
- **Multi-Provider:** Works with OpenAI, Anthropic, and other LLM providers
- **Type Safety:** Full TypeScript/Python type annotations for reliability
- **Async-First:** Built on async/await for high-performance concurrent operations
- **Streaming Support:** Real-time response streaming for better UX
- **Tool Calling:** Native function calling with structured input/output
- **Production Ready:** Built-in FastAPI integration and deployment patterns

Advanced Capabilities - Grounding & Multi-Modal

- **Google Search Grounding:** Connect agents to real-time web search results
- **Multi-Modal Input:** Support for text, images, audio, and video
- **Code Execution:** Built-in code interpreter for dynamic computation
- **Vertex AI Integration:** Enterprise features like RAG and vector search
- **Function Calling:** Structured tool execution with automatic parameter extraction
- **Safety Filters:** Built-in content safety and harm prevention

Enterprise Features - State & Deployment

- **Session Management:** Persistent conversation state across interactions
- **Cloud Integration:** Native support for Google Cloud services
- **Structured Output:** Pydantic models for type-safe responses
- **Error Handling:** Robust retry logic and error recovery
- **Monitoring:** Integration with Cloud Logging and Tracing
- **Authentication:** Built-in OAuth and API key management
- **Scalability:** Designed for high-throughput production workloads

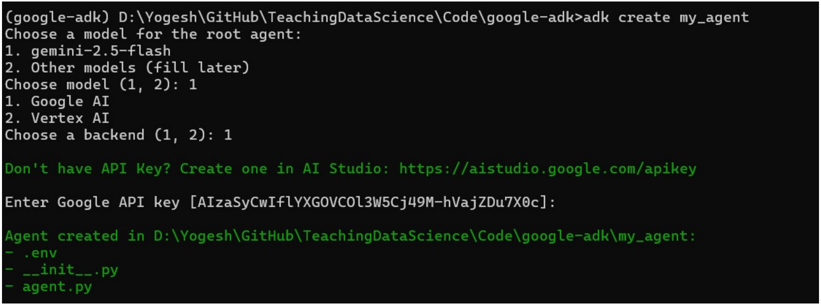
Installation & Setup

- **Simple Installation:** Install via pip with minimal dependencies
- **CLI Tools:** Built-in commands for project creation and management
- **Project Generation:** Automatic scaffolding with best practices
- **API Key Creation:**Get the API key from `https://aistudio.google.com/api-keys`
- **API Key Setup:** Configure Gemini API key via environment variable
- **Quick Start:** Start building agents immediately after installation

```
pip install google-adk
pip install google-adk --use-deprecated=legacy-resolver

# Create a new agent project
adk create my_agent

# Set API key in .env file
echo 'GOOGLE_API_KEY="YOUR_API_KEY"' > my_agent/.env # write your API key into an .env

```

Use CLI not VS code

Use Anaconda prompt with Admin permissions.



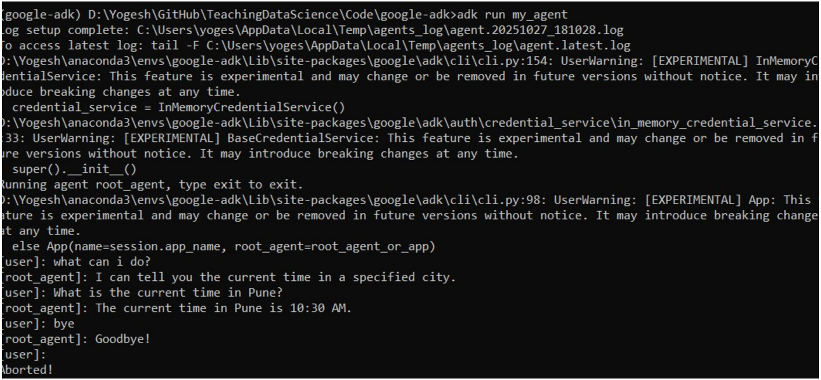
```
(google-adk) D:\Yogesh\GitHub\TeachingDataScience\Code\google-adk>adk create my_agent
Choose a model for the root agent:
1. gemini-2.5-flash
2. Other models (fill later)
Choose model (1, 2): 1
1. Google AI
2. Vertex AI
Choose a backend (1, 2): 1

Don't have API Key? Create one in AI Studio: https://aistudio.google.com/apikey

Enter Google API key [AIzaSyCwIflYXGOVCOl3W5Cj49M-hVajZDu7X0c]:

Agent created in D:\Yogesh\GitHub\TeachingDataScience\Code\google-adk\my_agent:
- .env
- __init__.py
- agent.py
```

Explore the agent project

- The created agent project has the following structure, with the agent.py file containing the main control code for the agent.
- The agent.py file contains a root_agent definition which is the only required element of an ADK agent. You can also define tools for the agent to use. Update the generated agent.py
- Run your agent using the adk run command-line tool.
- The ADK framework provides web interface you can use to test and interact with your agent.

```
my_agent/
    agent.py      # main agent code
    .env          # API keys or project IDs
    __init__.py

adk run my_agent

adk web --port 8000 my_agent
```

Wrong time!!



```
(google-adk) D:\Yogesh\GitHub\TeachingDataScience\Code\google-adk>adk run my_agent
Log setup complete: C:\Users\yoges\AppData\Local\Temp\agents_log\agent.20251027_181028.log
To access latest log: tail -F C:\Users\yoges\AppData\Local\Temp\agents_log\agent.latest.log
D:\Yogesh\anaconda3\envs\google-adk\Lib\site-packages\google\adk\cli\cli.py:154: UserWarning: [EXPERIMENTAL] InMemoryC
dentialService: This feature is experimental and may change or be removed in future versions without notice. It may in
oduce breaking changes at any time.
  credential_service = InMemoryCredentialService()
D:\Yogesh\anaconda3\envs\google-adk\Lib\site-packages\google\adk\auth\credential_service\in_memory_credential_service.
:33: UserWarning: [EXPERIMENTAL] BaseCredentialService: This feature is experimental and may change or be removed in f
ure versions without notice. It may introduce breaking changes at any time.
  super().__init__()
Running agent root_agent, type exit to exit.
D:\Yogesh\anaconda3\envs\google-adk\Lib\site-packages\google\adk\cli\cli.py:98: UserWarning: [EXPERIMENTAL] App: This
ature is experimental and may change or be removed in future versions without notice. It may introduce breaking change
at any time.
  else App(name=session.app_name, root_agent=root_agent_or_app)
[user]: what can i do?
[root_agent]: I can tell you the current time in a specified city.
[user]: What is the current time in Pune?
[root_agent]: The current time in Pune is 10:30 AM.
[user]: bye
[root_agent]: Goodbye!
[user]:
Aborted!
```

Basic Agent Example - Simple Setup

- Create a basic agent with Gemini 2.0 Flash model
- Define tools using Python functions with type hints
- Agent automatically handles function calling and response generation
- Clean separation between tool definition and agent logic
- Synchronous execution for simple use cases

```python
from adk import Agent
from adk.models import GeminiModel
import yfinance as yf

def get_stock_price(symbol: str) -> dict:
    """Get current stock price for a given symbol."""
    # Implementation using yfinance or similar
    stock = yf.Ticker(symbol)
    return {"symbol": symbol, "price": stock.info.get('currentPrice')}

agent = Agent(
    model=GeminiModel(model_name="gemini-2.0-flash-exp"),
    tools=[get_stock_price],
    instructions="You are a helpful financial assistant. Use tools when needed."
)

response = agent.run("What is NVIDIA's current stock price?")
print(response.text)
```

Agent with Multiple Tools

- Define multiple specialized tools for the agent with docstring
- Agent selects appropriate tools based on dcostring and user query

```python
from adk import Agent
from adk.models import GeminiModel
import yfinance as yf

def get_stock_price(symbol: str) -> float:
    """Get current stock price."""
    return yf.Ticker(symbol).info.get('currentPrice', 0)

def get_company_info(symbol: str) -> dict:
    """Get company information."""
    ticker = yf.Ticker(symbol)
    return {"name": ticker.info.get('longName'),
            "sector": ticker.info.get('sector'),
            "summary": ticker.info.get('longBusinessSummary')}

def get_analyst_recommendations(symbol: str) -> str:
    """Get analyst recommendations."""
    return yf.Ticker(symbol).recommendations.to_string()

agent = Agent(model=GeminiModel(model_name="gemini-2.0-flash-exp"),
    tools=[get_stock_price, get_company_info, get_analyst_recommendations],
    instructions="Provide detailed financial analysis using available tools.")

for chunk in agent.run_stream("Write a report on NVIDIA stock"):
    print(chunk.text, end="", flush=True)
```

Agent with Session Management

- Sessions maintain conversation history and context
- Enable multi-turn conversations with memory
- Store and retrieve previous interactions
- Support for persistent storage backends

```python
from adk import Agent, Session
from adk.models import GeminiModel
from adk.storage import InMemoryStorage
import yfinance as yf

def get_stock_price(symbol: str) -> float:
    """Get current stock price."""
    return yf.Ticker(symbol).info.get('currentPrice', 0)

def get_company_info(symbol: str) -> dict:
    """Get company information."""
    ticker = yf.Ticker(symbol)
    return {"name": ticker.info.get('longName'),
            "sector": ticker.info.get('sector'),
            "summary": ticker.info.get('longBusinessSummary')}

# Create storage for session history
storage = InMemoryStorage()

agent = Agent(
    model=GeminiModel(model_name="gemini-2.0-flash-exp"),
    tools=[get_stock_price, get_company_info],
    instructions="You are a financial advisor. Remember previous conversations."
)
```

Agent with Session Management

```python
# Create a session for this conversation
session = Session(
    agent=agent,
    storage=storage,
    session_id="user-123"
)

# Multi-turn conversation
response1 = session.run("What's NVIDIA's stock price?")
print(response1.text)

# Agent remembers previous context
response2 = session.run("How about their competitor AMD?")
print(response2.text)

# Session history is maintained
response3 = session.run("Compare both companies")
print(response3.text)

```

Agent with Google Search Grounding

- Google Search grounding provides real-time web information
- Automatically cites sources in responses
- Reduces hallucination with factual grounding
- Requires Vertex AI setup for production use

```
from adk import Agent
from adk.models import GeminiModel
from adk.extensions import GoogleSearchGrounding

agent = Agent(
    model=GeminiModel(
        model_name=”gemini−2.0−flash−exp”,
        extensions=[GoogleSearchGrounding()]
    ),
    instructions=”Provide well−researched answers with citations.”
)

response = agent.run(
    ”What are the latest developments in AI semiconductor technology?”
)
print(response.text)
# Response will include citations from search results

```

Structured Output with Pydantic

- Define expected output schema using Pydantic models
- Ensures type-safe and structured responses
- Automatic validation of agent outputs
- Ideal for integration with downstream systems

```python
from adk import Agent
from adk.models import GeminiModel
from pydantic import BaseModel, Field

def get_stock_price(symbol: str) -> float:
    :

def get_company_info(symbol: str) -> dict:
    :


class StockAnalysis(BaseModel):
    symbol: str = Field(description="Stock ticker symbol")
    current_price: float = Field(description="Current stock price")
    recommendation: str = Field(description="Buy/Hold/Sell recommendation")
    reasoning: str = Field(description="Analysis reasoning")

agent = Agent(
    model=GeminiModel(model_name="gemini-2.0-flash-exp"),
    tools=[get_stock_price, get_company_info],
    output_schema=StockAnalysis,
    instructions="Analyze the stock and provide structured recommendation."
)

result: StockAnalysis = agent.run("Analyze NVIDIA stock")
print(f"Symbol: {result.symbol}")
print(f"Price: ${result.current_price}")
```

Multi-Agent System - Architecture

- **Specialized Agents:** Each agent focuses on specific domain expertise
- **Orchestration:** Coordinator agent manages task delegation
- **Parallel Execution:** Multiple agents can work simultaneously
- **Result Aggregation:** Combine outputs from multiple agents
- **Scalable Design:** Handle complex workflows through collaboration

Multi-Agent Implementation - Part 1

```python
from adk import Agent
from adk.models import GeminiModel

def web_search(query: str) -> str:
    """Search the web for information."""
    # Implementation using search API
    pass

def get_stock_data(symbol: str) -> dict:
    """Get comprehensive stock data."""
    import yfinance as yf
    ticker = yf.Ticker(symbol)
    return {
        "price": ticker.info.get('currentPrice'),
        "recommendations": ticker.recommendations.tail(5).to_dict()
    }

web_agent = Agent(
    name="Web Agent",
    model=GeminiModel(model_name="gemini-2.0-flash-exp"),
    tools=[web_search],
    instructions="Search the web and provide sourced information."
)

finance_agent = Agent(
    name="Finance Agent",
    model=GeminiModel(model_name="gemini-2.0-flash-exp"),
    tools=[get_stock_data],
    instructions="Analyze financial data and present in clear tables."
)
```

Multi-Agent Implementation - Part 2

- Coordinator agent orchestrates multiple specialized agents
- Delegates tasks to appropriate agents based on requirements
- Aggregates and synthesizes results from multiple sources

```python
from adk import Agent, MultiAgentOrchestrator
from adk.models import GeminiModel

orchestrator = MultiAgentOrchestrator(
    agents=[web_agent, finance_agent],
    coordinator=Agent(
        model=GeminiModel(model_name="gemini-2.0-flash-exp"),
        instructions="""You coordinate a team of specialized agents.
        - Use Web Agent for market news and trends
        - Use Finance Agent for stock data and analysis
        - Synthesize their outputs into comprehensive reports.""" ))

# Run multi-agent workflow
result = orchestrator.run(
    "Provide a comprehensive analysis of AI semiconductor companies "
    "including market outlook and financial performance")

print(result.text)

# Access individual agent outputs
for agent_name, agent_result in result.agent_outputs.items():
    print(f"\n{agent_name} output:")
    print(agent_result.text)
```

FastAPI Deployment

Deploy agents as REST APIs using FastAPI

```
from fastapi import FastAPI, HTTPException
from fastapi.responses import StreamingResponse
from adk import Agent, Session
from adk.models import GeminiModel
from pydantic import BaseModel
import uvicorn

app = FastAPI(title=”ADK Agent API”)

agent = Agent(
    model=GeminiModel(model_name=”gemini−2.0−flash−exp”),
    tools=[get_stock_price, get_company_info],
    instructions=”You are a financial assistant API.”
)

class QueryRequest(BaseModel):
    message: str
    session_id: str
```

FastAPI Deployment

```python

@app.post("/chat")
async def chat(request: QueryRequest):
    session = Session(agent=agent, session_id=request.session_id)
    response = await session.run_async(request.message)
    return {"response": response.text}

@app.post("/chat/stream")
async def chat_stream(request: QueryRequest):
    session = Session(agent=agent, session_id=request.session_id)

    async def generate():
        async for chunk in session.run_stream_async(request.message):
            yield chunk.text

    return StreamingResponse(generate(), media_type="text/plain")

if __name__ == "__main__":
    uvicorn.run(app, host="0.0.0.0", port=8000)
```

Vertex AI Integration

```
from adk import Agent
from adk.models import VertexAIModel
from adk.extensions import VertexAIVectorSearch
from google.cloud import aiplatform

# Initialize Vertex AI
aiplatform.init(project=”your—project—id”, location=”us—central1”)

# Create agent with Vertex AI backend
agent = Agent(
    model=VertexAIModel(
        model_name=”gemini—2.0—flash—exp”,
        project=”your—project—id”,
        location=”us—central1”
    ),
    extensions=[
        VertexAIVectorSearch(
            index_endpoint=”your—index—endpoint”,
            deployed_index_id=”your—index—id”
        )
    ],
    instructions=”Answer questions using both your knowledge and the vector store.”
)

response = agent.run(”What are our company policies on remote work?”)
print(response.text)
```

Error Handling & Reliability

```python
from adk import Agent
from adk.models import GeminiModel
from adk.exceptions import ToolExecutionError, ModelError
import logging
import yfinance as yf
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

def safe_tool_wrapper(func):
    """Decorator for safe tool execution with fallback."""
    def wrapper(*args, **kwargs):
        try:
            return func(*args, **kwargs)
        except Exception as e:
            logger.error(f"Tool {func.__name__} failed: {e}")
            return {"error": str(e), "fallback": True}
    return wrapper

@safe_tool_wrapper
def get_stock_price(symbol: str) -> dict:
    """Get stock price with error handling."""
    return {"price": yf.Ticker(symbol).info['currentPrice']}

```

Error Handling & Reliability

```
agent = Agent(
    model=GeminiModel(
        model_name="gemini-2.0-flash-exp",
        max_retries=3,
        timeout=30
    ),
    tools=[get_stock_price],
    instructions="Handle errors gracefully and inform users."
)

try:
    response = agent.run("What's the price of NVDA?")
    print(response.text)
except ModelError as e:
    logger.error(f"Model error: {e}")
    print("Sorry, I'm having trouble processing your request.")
```

Framework Comparison: LangGraph vs Agno vs ADK

Feature	LangGraph	Agno	ADK
Primary Focus	Graph-based workflows	Multi-agent systems	Enterprise AI agents
Developer	LangChain AI	Agno (ex-phidata)	Google
Architecture	State machines & graphs	Agent teams	Agent orchestration
Model Support	100+ providers	23+ providers	Gemini + multi-provider
Learning Curve	Steep (graph concepts)	Moderate	Moderate
Setup Time	Complex	Minutes	Minutes
Performance	Good	3μs instantiation	Optimized for Gemini
State Management	Built-in (checkpoints)	Session storage	Session & Cloud
Memory	Memory modules	Built-in drivers	In-memory & Cloud
Multi-Agent	Via subgraphs	Native teams	Orchestrator pattern
Workflow Control	Explicit graphs	Coordinate mode	Coordinator agent
Streaming	Yes	Yes	Yes (native async)
RAG Support	Vector stores	20+ vector DBs	Vertex AI Search
Deployment	Custom	FastAPI built-in	FastAPI + Cloud Run
Monitoring	LangSmith	agno.com	Cloud Console
Production Ready	Yes	Yes	Yes (enterprise)
Best For	Complex workflows	Fast prototyping	Google Cloud users
Unique Feature	Graph visualization	6.5KB/agent memory	Search grounding

Best Practices

- **Start Simple:** Begin with basic agents before adding complexity
- **Clear Instructions:** Provide specific, actionable instructions to agents
- **Tool Design:** Keep tools focused and well-documented with type hints
- **Error Handling:** Always implement proper error handling and retries
- **Testing:** Test agents thoroughly before production deployment
- **Monitoring:** Implement logging and monitoring for production systems
- **Security:** Validate inputs and sanitize outputs
- **Cost Management:** Monitor API usage and implement rate limiting

Getting Started - Next Steps

- **Documentation:** Explore comprehensive guides at `https://google.github.io/adk-docs/`
- **Examples Repository:** Check out example projects and templates
- **Community:** Join Google Cloud community for support
- **Start Building:** Begin with simple agents and iterate
- **Cloud Integration:** Leverage Google Cloud services for production
- **Vertex AI:** Explore enterprise features for advanced use cases
- **Monitoring:** Use Cloud Console for production monitoring

Resources & References

- **Official Documentation:** `https://google.github.io/adk-docs/`
- **GitHub Repository:** `https://github.com/google/adk`
- **Gemini API:** `https://ai.google.dev/`
- **Vertex AI:** `https://cloud.google.com/vertex-ai`
- **Google Cloud:** `https://cloud.google.com/`
- **Community Support:** Google Cloud Community forums

Thanks . . .

- Search **”Yogesh Haribhau Kulkarni”** on Google and follow me on LinkedIn, GitHub, Medium
- Office Hours: Saturdays, 2 to 3 pm (IST); Free-Open to all; email for appointment.
- yogeshkulkarni at yahoo dot com
- Call + 9 1 9 8 9 0 2 5 1 4 0 6

(https://medium.com/@yogeshharibhaukulkarni)

(https://www.linkedin.com/in/yogeshkulkarni/)

(https://www.github.com/yogeshhk/)

Pune AI Community (PAIC)

- Two-way communication:
  - Website puneaicommunity dot org
  - Email puneaicommunity at gmail dot com
  - Call + 9 1 9 8 9 0 2 5 1 4 0 6
  - LinkedIn: https://linkedin.com/company/pune-ai-community
- One-way Announcements:
  - Twitter (X) @puneaicommunity
  - Instagram @puneaicommunity
  - WhatsApp Community: Invitation Link https://chat.whatsapp.com/LluOrhyEzuQLDr25ixZ
  - Luma Event Calendar: puneaicommunity
- Contribution Channels:
  - GitHub: Pune-AI-Community and puneaicommunity
  - Medium: pune-ai-community
  - YouTube: @puneaicommunity

Website

Pune AI Community (PAIC) QR codes

Website

Medium Blogs

Twitter-X

LinkedIn Page

Github Repository

WhatsApp Invite

Luma Events

YouTube Videos

Instagram