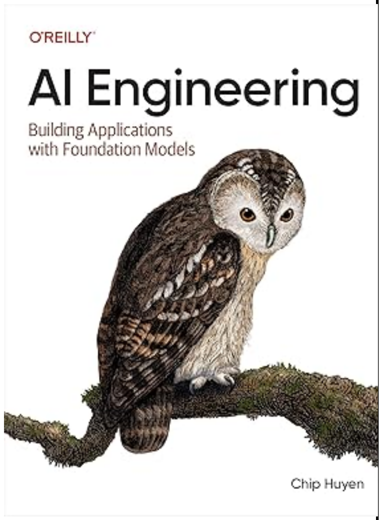
Basics

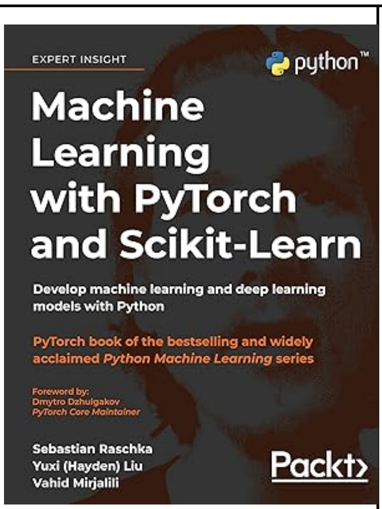
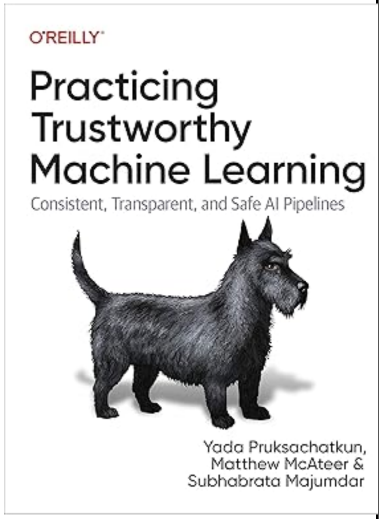
Agentic AI Development - Complete Learning Resources

Author: Prashant Kulkarni, Google

Table of Contents

Books

Title	Cover	Purpose
Building AI Agents with LLMs, RAG, and Knowledge Graphs: A practical guide to autonomous and modern AI agents		First step towards building Agentic AI
AI Engineering: Building Applications with Foundation Models		Go Deeper into various application that can be built with Foundational Models

Machine Learning with PyTorch and Scikit-Learn: Develop machine learning and deep learning models with Python		Go even more deep into how to code them
Build a Large Language Model (From Scratch)		Go to the pisces for LLM development - Gold Standard
Practicing Trustworthy Machine Learning: Consistent, Transparent, and Fair AI Pipelines		Trustworthy ML - the title is enough!

What are Agents?

An AI agent is an autonomous system that can perceive its environment, reason about goals, make decisions, and take actions to achieve those goals. Unlike traditional software that follows predetermined logic, agents can adapt their behavior based on context and feedback.

Core Agent Characteristics

- **Autonomy**: Can operate independently without constant human guidance
- **Reactivity**: Responds to changes in environment or input
- **Proactivity**: Takes initiative to achieve goals
- **Social Ability**: Can communicate and coordinate with other agents or humans

Levels of Autonomy on AI Agents. https://arxiv.org/abs/2506.12469

Autonomy

Definition: The ability to operate independently without constant human guidance.

Levels of Autonomy:

- **Level 0 - No Autonomy**: Human performs all tasks
- **Level 1 - Assistance**: Agent provides suggestions, human decides
- **Level 2 - Partial Autonomy**: Agent handles specific subtasks
- **Level 3 - Conditional Autonomy**: Agent operates independently in defined scenarios
- **Level 4 - High Autonomy**: Agent handles most situations, requests help rarely
- **Level 5 - Full Autonomy**: Complete independence in all scenarios

Reactivity

Definition: The ability to perceive and respond to changes in the environment in a timely manner.

Types of Reactive Behaviors:

- **Immediate Response**: Real-time reaction to stimuli
- **Event-Driven**: Triggered by specific occurrences
- **Threshold-Based**: Action when metrics exceed limits
- **Pattern Recognition**: Response to detected patterns

Proactivity

Definition: Goal-directed behavior where agents take initiative rather than just responding to events.

Proactive Capabilities:

- **Goal Setting**: Establishing objectives based on high-level directives
- **Planning**: Creating action sequences to achieve goals
- **Opportunity Recognition**: Identifying beneficial actions without prompting
- **Preventive Actions**: Anticipating and preventing problems

Social Ability

Definition: The capacity to interact with other agents and humans through communication and coordination.

Social Interaction Patterns:

- **Cooperation**: Working together toward shared goals
- **Negotiation**: Resolving conflicts and reaching agreements
- **Competition**: Engaging in competitive scenarios
- **Teaching/Learning**: Knowledge transfer between agents

Agent Tools, MCP & Agent-to-Agent Communication

1. Agent Tools Ecosystem

Core Tool Categories

Information Retrieval

- **Web Search**: Google Search API, Bing Search, SerpAPI
- **Knowledge Bases**: Wikipedia API, Wolfram Alpha, specialized databases
- **Document Processing**: PDF parsers, OCR, document summarization
- **Code Search**: GitHub API, Stack Overflow integration

Code Execution & Development

- **Code Interpreters**: Python REPL, Jupyter kernels, sandboxed execution
- **Version Control**: Git operations, repository management
- **Development Tools**: Linting, testing, deployment automation
- **Resources**:

    - E2B Code Interpreter
    - Replit Agent
    - GitHub Copilot Workspace

Data & Analytics

- **Database Connections**: SQL execution, NoSQL queries
- **API Integrations**: REST/GraphQL clients, authentication handling
- **Data Processing**: ETL operations, data validation, visualization
- **Business Intelligence**: Report generation, dashboard creation

Communication & Collaboration

- **Email**: Send/receive, calendar integration
- **Messaging**: Slack, Discord, Teams integration
- **File Management**: Cloud storage, document sharing
- **Project Management**: Jira, Asana, Notion integration

Tool Development Best Practices

Tool Design Principles

- **Atomic Operations**: Single responsibility per tool
- **Error Handling**: Graceful failure and retry mechanisms
- **Schema Validation**: Clear input/output specifications
- **Security**: Authentication, authorization, input sanitization

Tool Integration Patterns

```python
# Example tool structure
{
  "name": "search_documents",
  "description": "Search through document collection",
  "parameters": {
    "type": "object",
    "properties": {
      "query": {"type": "string"},
      "max_results": {"type": "integer", "default": 10}
    },
    "required": ["query"]
  }
}
```

2. Model Context Protocol (MCP)

What is MCP?

The **Model Context Protocol (MCP)** is an open standard developed by Anthropic that provides a unified way for AI assistants to connect with external data sources and tools. It establishes a secure, standardized communication layer between AI models and the resources they need to access.

Key Components

MCP Servers

- **Purpose**: Expose resources (tools, prompts, context) to AI models
- **Types**: Local servers (file systems, databases), remote servers (APIs, cloud services)
- **Examples**: File system access, database queries, API integrations

MCP Clients

- **Purpose**: AI applications that consume MCP server resources
- **Integration**: Built into Claude Desktop, can be integrated into custom applications
- **Security**: Controlled access through permission models

Resource Types

- **Tools**: Executable functions (file operations, API calls)
- **Prompts**: Reusable prompt templates with parameters
- **Resources**: Static content (documents, images, data)

MCP Implementation

Setting Up MCP Servers

```JSON
{
  "mcpServers": {
    "filesystem": {
      "command": "npx",
      "args": ["-y", "@modelcontextprotocol/server-filesystem",
"/path/to/allowed/files"]
    },
    "postgres": {
      "command": "uvx",
      "args": ["mcp-server-postgres", "--connection-string",
"postgresql://..."]
    }
  }
}
```

Available MCP Servers

- **Filesystem**: Local file operations
- **Database**: PostgreSQL, SQLite, MySQL connections
- **Git**: Repository operations
- **Google Drive**: Cloud file access
- **Slack**: Team communication
- **Brave Search**: Web search capabilities

Resources for MCP

- MCP Documentation
- MCP Server Examples
- Building Custom MCP Servers
- MCP Security Guide

3. Agent-to-Agent (A2A) Communication

Agent2Agent (A2A) is a new, open protocol with support and contributions from more than 50 technology partners like Atlassian, Box, Cohere, Intuit, Langchain, MongoDB, PayPal, Salesforce, SAP, ServiceNow, UKG and Workday.

Communication Patterns

Direct Communication

- **Synchronous**: Request-response patterns, real-time interaction
- **Asynchronous**: Message queues, event-driven communication
- **Use Cases**: Task delegation, information sharing, coordination

Mediated Communication

- **Message Brokers**: RabbitMQ, Apache Kafka, Redis
- **API Gateways**: Centralized routing and management
- **Service Mesh**: Istio, Linkerd for microservices communication

Broadcast/Multicast

- **Event Streams**: Publishing events to multiple subscribers
- **Coordination**: Distributed consensus, leader election
- **Use Cases**: Status updates, global state changes

A2A Protocols & Standards

Communication Formats

```json
{
  "message_type": "task_delegation",
  "sender_id": "agent_1",
  "recipient_id": "agent_2",
  "payload": {
    "task": "analyze_data",
    "parameters": {...},
    "priority": "high"
  },
  "timestamp": "2025-01-15T10:30:00Z"
}
```

Protocol Layers

- **Transport**: HTTP/REST, WebSocket, gRPC
- **Message Format**: JSON, Protocol Buffers, MessagePack
- **Semantic Layer**: Task ontologies, shared vocabularies

Multi-Agent Coordination

Coordination Strategies

- **Centralized**: Single coordinator manages all agents
- **Hierarchical**: Tree-like delegation structure
- **Peer-to-Peer**: Distributed coordination without central authority
- **Hybrid**: Combines multiple approaches based on context

Consensus Mechanisms

- **Voting**: Democratic decision making among agents
- **Leader Election**: Dynamic selection of coordinating agent
- **Conflict Resolution**: Handling disagreements and failures

State Management

- **Shared State**: Distributed databases, state machines
- **Event Sourcing**: Audit trail of all agent actions
- **CRDT**: Conflict-free replicated data types for consistency

Implementation Frameworks

Multi-Agent Platforms

- **JADE (Java)**: Mature platform with FIPA compliance
- **SPADE (Python)**: Modern multi-agent system framework
- **Mesa (Python)**: Agent-based modeling framework
- **Resources**:
    - JADE Documentation
    - SPADE Tutorial
    - Mesa Examples

Communication Libraries

- **ZeroMQ**: High-performance messaging library
- **Apache Kafka**: Distributed streaming platform
- **Redis Streams**: Real-time messaging
- **gRPC**: High-performance RPC framework

4. Integration Patterns

Tool Chaining

- **Sequential**: Output of one tool becomes input to next
- **Parallel**: Multiple tools executed simultaneously
- **Conditional**: Tool selection based on runtime conditions
- **Error Recovery**: Fallback tools and retry mechanisms

Context Sharing

- **Session State**: Maintaining context across tool calls
- **Cross-Agent Memory**: Shared knowledge bases
- **Event History**: Audit trails and learning from past actions

Security & Governance

- **Access Control**: Role-based permissions for tools and agents
- **Audit Logging**: Tracking all agent actions and communications
- **Rate Limiting**: Preventing abuse and managing costs
- **Data Privacy**: Ensuring sensitive information protection

Monitoring & Observability

- **Agent Behavior Tracking**: Understanding decision-making patterns
- **Performance Metrics**: Latency, success rates, resource usage
- **Communication Analysis**: Message flow and bottleneck identification
- **Debugging Tools**: Step-by-step execution tracing

5. Advanced Integration Scenarios

Enterprise Integration

- **API Management**: Centralized tool catalogs and versioning
- **Service Discovery**: Dynamic tool registration and lookup
- **Load Balancing**: Distributing agent workloads
- **Disaster Recovery**: Failover and backup strategies

Cloud-Native Patterns

- **Containerization**: Docker containers for agent and tool isolation
- **Orchestration**: Kubernetes for scaling and management
- **Serverless**: Function-as-a-Service for event-driven tools
- **Edge Computing**: Distributed agent deployment

Real-Time Systems

- **Stream Processing**: Handling continuous data flows
- **Event-Driven Architecture**: Reactive system design
- **Low-Latency Communication**: Optimizing for speed
- **Fault Tolerance**: Handling failures gracefully

Resources for Tools, MCP & A2A

Essential Reading

- MCP Specification
- Multi-Agent Systems Book
- Distributed Systems Patterns

Practical Tutorials

- Building MCP Servers Tutorial

- Multi-Agent System Design
- Agent Communication Patterns

Communities & Forums

- MCP Discord Community
- Multi-Agent Systems Conference
- Agent Communication Research Group

The Agentic AI Landscape

What is Agentic AI?

Agentic AI represents a paradigm shift from reactive AI systems to proactive, autonomous agents that can plan, reason, and act independently to achieve goals. Think of the difference between a calculator (responds to input) and a personal assistant (proactively manages your schedule).

The Evolution Timeline

- **Traditional AI**: Pattern recognition, classification
- **Conversational AI**: Chat interfaces, Q&A systems
- **Tool-using AI**: LLMs with function calling capabilities
- **Agentic AI**: Autonomous planning, multi-step reasoning, goal-oriented behavior
- **Multi-Agent Systems**: Coordinated networks of specialized agents

Why Agentic AI Matters Now

- **LLM Capabilities**: Advanced reasoning and planning abilities
- **Tool Integration**: Seamless API and system interactions
- **Cost Efficiency**: Automation of complex knowledge work
- **Scalability**: Handling tasks too complex for traditional automation

---

Foundational Knowledge

Essential Prerequisites

1. Machine Learning Fundamentals

- **Concepts**: Supervised/unsupervised learning, neural networks, training/inference

- **Resources**:
    - Machine Learning Crash Course (Google)
    - Coursera ML Course (Andrew Ng)
    - Fast.ai Practical Deep Learning

2. Large Language Models (LLMs)

- **Key Topics**: Transformers, attention mechanisms, prompting, fine-tuning
- **Resources**:
    - The Illustrated Transformer
    - Attention Is All You Need (Paper)
    - LLM University (Cohere)
    - OpenAI GPT Guide

3. Programming & Software Engineering

- **Languages**: Python (primary), JavaScript/TypeScript (web apps)
- **Concepts**: APIs, microservices, containerization, version control
- **Resources**:
    - Python for AI Development
    - API Design Best Practices
    - Docker Fundamentals

Core Concepts & Architectures

1. Agent Architectures

Agent Architectures & Classifications

1. Reactive Agents

- **Definition**: Respond directly to stimuli without internal state
- **Behavior**: Stimulus → Response (no planning or memory)
- **Use Cases**: Simple automation, trigger-based actions
- **Example**: Chatbot that answers FAQ questions
- **Resources**:
    - Reactive Agent Patterns
    - Behavior-Based Robotics

2. Deliberative (Planning) Agents

- **Definition**: Use internal models to plan and reason about actions
- **Behavior**: Sense → Plan → Act

- **Components**: Beliefs, Desires, Intentions (BDI architecture)
- **Use Cases**: Complex task planning, strategic decision making
- **Example**: Travel planning agent that considers multiple constraints
- **Resources**:
  - BDI Agent Architecture
  - Planning in AI (Russell & Norvig)

3. Hybrid Agents

- **Definition**: Combine reactive and deliberative capabilities
- **Behavior**: Fast reactions for urgent situations, planning for complex goals
- **Architecture**: Layered approach with reactive and deliberative layers
- **Use Cases**: Autonomous vehicles, game AI, robotic systems
- **Example**: Customer service agent that handles simple queries reactively but escalates complex issues to planning layer

4. Learning Agents

- **Definition**: Improve performance through experience
- **Components**: Learning element, performance element, critic, problem generator
- **Types**: Supervised, unsupervised, reinforcement learning
- **Use Cases**: Personalization, adaptation to user preferences
- **Example**: Recommendation agent that learns user preferences over time
- **Resources**:
  - Reinforcement Learning: An Introduction
  - Multi-Agent Reinforcement Learning

Specialized Agent Types

1. Conversational Agents

- **Purpose**: Natural language interaction with users
- **Components**: NLU, dialogue management, NLG, knowledge base
- **Patterns**: Rule-based, retrieval-based, generative
- **Technologies**: Large Language Models, intent recognition, entity extraction
- **Resources**:
  - Conversational AI Guide
  - Dialogue Systems Book

2. Task-Oriented Agents

- **Purpose**: Complete specific objectives (booking, scheduling, analysis)
- **Components**: Goal decomposition, task planning, execution monitoring
- **Patterns**: Workflow-based, goal-oriented, constraint satisfaction
- **Example**: Travel booking agent, data analysis agent
- **Resources**:

    - Task-Oriented Dialogue Systems
    - Goal-Oriented AI Planning

3. Collaborative Agents

- **Purpose**: Work together to achieve shared or complementary goals
- **Components**: Communication protocols, coordination mechanisms, shared context
- **Patterns**: Leader-follower, peer-to-peer, hierarchical
- **Example**: Software development team with specialist agents
- **Resources**:
    - Multi-Agent Systems Book
    - Distributed AI Research

4. Monitoring & Control Agents

- **Purpose**: Oversee systems, detect anomalies, maintain operations
- **Components**: Sensors, event processing, alerting, corrective actions
- **Patterns**: Observer, guardian, supervisor
- **Example**: System monitoring agent, security surveillance agent
- **Resources**:
    - Autonomous Systems Engineering
    - Control Theory for Computer Scientists

2. Core Components

Tool Usage & Function Calling

- **Concepts**: API integration, function schemas, error handling
- **Implementations**: OpenAI Functions, Anthropic Tools, Google Function Calling
- **Resources**:
    - OpenAI Function Calling Guide
    - Tool Use Best Practices

Memory & Context Management

- **Types**: Short-term (conversation), long-term (episodic/semantic), working memory
- **Techniques**: Vector databases, summarization, retrieval-augmented generation
- **Resources**:
    - Memory in LLM Agents
    - Vector Database Comparison

Agent Communication

- **Patterns**: Message passing, shared memory, event-driven

- **Protocols**: REST APIs, message queues, real-time communication
- **Resources**:
  - Agent Communication Languages
  - Microservices Patterns

Agent Communication & Coordination

Communication Patterns

- Direct Communication: Point-to-point messaging
- Broadcast: One-to-many messaging
- Publish-Subscribe: Event-driven communication
- Blackboard: Shared information space

Coordination Mechanisms

- Contract Net Protocol: Task assignment through bidding
- Consensus Algorithms: Distributed agreement
- Market Mechanisms: Economic coordination
- Organizational Hierarchies: Structured coordination

Agent Design Principles

1. Single Responsibility Principle

- Each agent should have one clear purpose or domain of expertise
- Avoid "super agents" that try to do everything
- Enable better testing, maintenance, and reusability

2. Autonomy vs Coordination Balance

- Agents should be autonomous but not isolated
- Design clear interfaces for inter-agent communication
- Establish protocols for conflict resolution

3. Robustness & Fault Tolerance

- Design for graceful degradation when components fail
- Implement retry mechanisms and fallback strategies
- Monitor agent health and performance

4. Scalability Considerations

- Design agents to handle increasing load
- Consider horizontal scaling patterns
- Minimize shared state and dependencies

Development Frameworks & Tools

Enterprise-Grade Frameworks

Google ADK (Agent Development Kit)

- **Strengths**: Production-ready, Google ecosystem integration, built-in evaluation
- **Use Cases**: Enterprise applications, multi-agent systems, scalable deployment
- **Resources**:
  - Official Documentation
  - Sample Agents
  - Introduction Video

Microsoft Semantic Kernel

- **Strengths**: .NET/C# focus, enterprise integration, plugin ecosystem
- **Use Cases**: Microsoft stack integration, enterprise workflows
- **Resources**:
  - Semantic Kernel Docs
  - GitHub Repository

Open Source Frameworks

LangChain/LangGraph

- **Strengths**: Mature ecosystem, extensive integrations, Python/JavaScript
- **Use Cases**: Rapid prototyping, research, complex workflows
- **Resources**:
  - LangChain Documentation
  - LangGraph Guide
  - LangChain Academy

CrewAI

- **Strengths**: Multi-agent focus, role-based design, simple API
- **Use Cases**: Team-based AI workflows, collaborative agents

- **Resources**:
    - CrewAI Documentation
    - GitHub Repository

AutoGen (Microsoft)

- **Strengths**: Conversational agents, multi-agent chat, research-oriented
- **Use Cases**: Research, conversational AI, agent-to-agent communication
- **Resources**:
    - AutoGen Documentation
    - Research Papers

Specialized Tools

Agent Evaluation

- **Tools**: Weights & Biases, MLflow, custom evaluation frameworks
- **Metrics**: Task success rate, reasoning quality, tool usage efficiency
- **Resources**:
    - Agent Evaluation Guide
    - Berkeley Agent Evaluation

Observability & Monitoring

- **Tools**: LangSmith, Arize, Phoenix, custom logging
- **Focus**: Agent behavior tracking, performance monitoring, debugging
- **Resources**:
    - LangSmith Tracing
    - MLOps for Agents

Implementation Pathways

Learning Path 1: Research & Experimentation

Timeline: 2-3 months

1. **Week 1-2**: LLM fundamentals + prompting techniques
2. **Week 3-4**: Simple ReAct agents with LangChain
3. **Week 5-6**: Tool integration and function calling
4. **Week 7-8**: Multi-agent experiments with CrewAI
5. **Week 9-12**: Custom research project

Learning Path 2: Enterprise Development

Timeline: 3-4 months

1. **Month 1**: Foundations + Google ADK basics
2. **Month 2**: Production patterns + deployment
3. **Month 3**: Security, evaluation, and monitoring
4. **Month 4**: Real-world project implementation

Learning Path 3: Academic Research

Timeline: 6+ months

1. **Months 1-2**: Deep theory (papers, algorithms)
2. **Months 3-4**: Novel architecture experimentation
3. **Months 5-6**: Research project + publication

Advanced Topics

1. Agent Safety & Alignment

- **Concepts**: Constitutional AI, red teaming, robustness testing
- **Resources**:
  - Anthropic Safety Research
  - AI Safety Fundamentals

2. Scalable Agent Systems

- **Concepts**: Distributed systems, load balancing, fault tolerance
- **Resources**:
  - Designing Data-Intensive Applications
  - Microservices Architecture

3. Agent Learning & Adaptation

- **Concepts**: Reinforcement learning, online learning, meta-learning
- **Resources**:
  - RL for LLMs
  - Meta-Learning Survey

4. Human-Agent Interaction

- **Concepts**: UI/UX for agents, human-in-the-loop, explainability
- **Resources**:

    - HAI Research
    - Explainable AI Guide

Industry Applications

Current Success Stories

- **Customer Service**: Automated support agents
- **Software Development**: Code generation and review agents
- **Data Analysis**: Autonomous data scientists
- **Content Creation**: Multi-modal content agents
- **Research**: Literature review and synthesis agents

Emerging Applications

- **Scientific Discovery**: Lab automation, hypothesis generation
- **Financial Services**: Trading, risk assessment, compliance
- **Healthcare**: Diagnostic assistance, treatment planning
- **Education**: Personalized tutoring, curriculum design

Learning Path Recommendations

For Beginners (No AI Background)

1. Start with Machine Learning Crash Course
2. Learn Python programming fundamentals
3. Understand LLMs through LLM University
4. Build simple agents with LangChain tutorials
5. Experiment with Google ADK quickstart

For ML Engineers

1. Study agent architectures (ReAct, planning)
2. Learn LangChain/LangGraph deeply
3. Explore Google ADK for production use
4. Focus on evaluation and monitoring
5. Build multi-agent systems

For Software Engineers

1. Understand LLM APIs and prompting
2. Learn agent frameworks (ADK, Semantic Kernel)
3. Focus on system design and scalability
4. Study deployment patterns
5. Implement production-ready solutions

For Researchers

1. Read foundational papers on agent architectures
2. Experiment with novel approaches
3. Use research-oriented tools (AutoGen)
4. Contribute to open-source frameworks
5. Publish novel findings

Next Steps

Choose your learning path based on your background and goals, then start with the foundational resources. Remember that agentic AI is a rapidly evolving field - stay connected with the community through conferences, papers, and open-source contributions.

Key Communities:

- r/MachineLearning
- AI Safety Discord
- LangChain Discord
- Twitter AI Research Community

Project

Travel Concierge & Rental Car Agents - ADK Project Guide

Project Overview

Build a multi-agent system using Google ADK that helps users plan trips by coordinating between a Travel Concierge Agent and a Rental Car Agent. This project demonstrates key agentic AI concepts including multi-agent coordination, tool usage, and workflow orchestration.

Key Project Highlights:

Multi-Agent Architecture

- **Travel Concierge Agent**: Coordinator that handles overall trip planning
- **Rental Car Agent**: Specialized agent for vehicle bookings
- **Clear separation of concerns** with agent-to-agent communication

Progressive Learning Structure

- **Phase 1**: Foundation setup with basic tools
- **Phase 2**: Specialized agents and communication
- **Phase 3**: Workflow orchestration and integration
- **Phase 4**: Real API integration and production features

ADK Concept Coverage

- **Tool Usage**: Weather, flight search, car availability tools
- **Agent Communication**: Message passing between agents
- **Workflow Orchestration**: Sequential and parallel task execution
- **Context Management**: Shared state across agents
- **Error Handling**: Retry policies and graceful failures

Real-World Application

- **Practical APIs**: Amadeus, Skyscanner, Booking.com integration
- **Production Patterns**: Error handling, retries, monitoring
- **Deployment Options**: Local, Google Cloud, Vertex AI

Extension Opportunities

- **MCP Integration**: Secure tool access patterns
- **Advanced Coordination**: Auction-based car selection
- **Machine Learning**: User preference learning

The project starts simple (mock data, basic agents) and gradually adds complexity, making it perfect for learning ADK while building something genuinely useful. Each phase has clear deliverables and builds upon previous work.

System Architecture

```None
User Request → Travel Concierge Agent (Coordinator)
                    ↓
                ┌─ Flight Search Tool
                ├─ Hotel Search Tool
                ├─ Weather Tool
                └─ Rental Car Agent
                    ↓
                ┌─ Car Availability Tool
                ├─ Pricing Tool
                └─ Booking Tool
```

Learning Objectives

- **Multi-Agent Coordination**: Agent-to-agent communication and task delegation
- **Tool Integration**: External API usage and custom tool creation
- **Workflow Orchestration**: Sequential and parallel task execution
- **Context Management**: Sharing information between agents
- **Error Handling**: Graceful failure and recovery patterns

Project Phases

Phase 1: Foundation Setup (Week 1)

Goals

- Set up ADK development environment
- Create basic agent structure
- Implement simple tool integration

Tasks

1.1 Environment Setup

Shell
```shell
# Create and activate conda environment
conda create -n travel-agents python=3.11
conda activate travel-agents

# Install core dependencies
pip install google-adk
pip install requests python-dotenv pytest

# For development and testing
pip install black flake8 mypy jupyter

# Set up project structure
mkdir travel-agents
cd travel-agents
mkdir agents tools tests config data notebooks

# Create environment configuration
touch .env
echo "travel-agents" > environment.yml

# Initialize git repository
git init
touch .gitignore
```

1.2 Create Environment Configuration

None
```
# environment.yml
name: travel-agents
```

```
channels:
  - conda-forge
  - defaults
dependencies:
  - python=3.11
  - pip
  - pip:
    - google-adk
    - requests
    - python-dotenv
    - pytest
    - black
    - flake8
    - mypy
    - jupyter
    - amadeus
    - openai
```

1.3 Environment Variables Setup

Shell

```shell
# .env file
GOOGLE_APPLICATION_CREDENTIALS="path/to/your/credentials.json"
AMADEUS_API_KEY="your_amadeus_api_key"
AMADEUS_API_SECRET="your_amadeus_api_secret"
OPENWEATHER_API_KEY="your_openweather_api_key"
GEMINI_API_KEY="your_gemini_api_key"

# Development settings
DEBUG=True
LOG_LEVEL=INFO
```

1.4 Git Configuration

```shell
# .gitignore
__pycache__/
*.py[cod]
*$py.class
*.so
.Python
env/
venv/
ENV/
.env
.venv
pip-log.txt
pip-delete-this-directory.txt
.pytest_cache/
.coverage
htmlcov/
.idea/
.vscode/
*.log
credentials.json
config/secrets.yaml
```

1.2 Create Travel Concierge Agent

```python
# agents/travel_concierge.py
from adk import Agent, LlmAgent
from adk.tools import Tool

class TravelConciergeAgent(LlmAgent):
    def __init__(self):
        super().__init__(
            name="travel_concierge",
            description="Coordinates travel planning and rental
car booking",
            model="gemini-pro"
        )
        self.tools = [
            WeatherTool(),
            FlightSearchTool(),
            HotelSearchTool()
        ]

    def process_request(self, user_request):
        # Parse travel requirements
        # Coordinate with rental car agent if needed
        # Return comprehensive travel plan
        pass
```

1.3 Create Basic Weather Tool

```python
# tools/weather_tool.py
from adk.tools import Tool
import requests

class WeatherTool(Tool):
    def __init__(self):
        super().__init__(
            name="get_weather",
            description="Get weather forecast for a destination",
            schema={
                "type": "object",
                "properties": {
                    "location": {"type": "string"},
                    "dates": {"type": "array", "items": {"type":
"string"}}
                },
                "required": ["location", "dates"]
            }
        )

    def execute(self, location, dates):
        # Call weather API (OpenWeatherMap, etc.)
        # Return formatted weather data
        pass
```

Deliverable

- Basic agent that can respond to travel queries
- Working weather tool integration
- Simple test cases

Phase 2: Rental Car Agent (Week 2)

Goals

- Create specialized rental car agent
- Implement car search and availability tools
- Establish agent-to-agent communication

Tasks

2.1 Rental Car Agent

```python
# agents/rental_car_agent.py
from adk import LlmAgent
from adk.tools import Tool

class RentalCarAgent(LlmAgent):
    def __init__(self):
        super().__init__(
            name="rental_car_agent",
            description="Handles rental car search, pricing, and
booking",
            model="gemini-pro"
        )
        self.tools = [
            CarAvailabilityTool(),
            CarPricingTool(),
            CarBookingTool()
        ]

    def search_cars(self, pickup_location, return_location,
dates, preferences):
        # Search available rental cars
        # Compare prices across providers
        # Return ranked options
        pass
```

2.2 Car Availability Tool

```python
# tools/car_tools.py
class CarAvailabilityTool(Tool):
    def __init__(self):
        super().__init__(
            name="search_rental_cars",
            description="Search for available rental cars",
            schema={
                "type": "object",
                "properties": {
                    "pickup_location": {"type": "string"},
                    "return_location": {"type": "string"},
                    "pickup_date": {"type": "string"},
                    "return_date": {"type": "string"},
                    "car_type": {"type": "string", "enum":
["economy", "compact", "midsize", "fullsize", "luxury", "suv"]}
                },
                "required": ["pickup_location",
"return_location", "pickup_date", "return_date"]
            }
        )

    def execute(self, **kwargs):
        # Mock data for development
        return {
            "cars": [
                {
                    "provider": "Hertz",
                    "model": "Toyota Corolla",
                    "type": "economy",
                    "daily_rate": 45.99,
                    "total_cost": 183.96,
                    "availability": True
                },
                # Add more mock cars
            ]
        }
```

2.3 Agent Communication

```python
# Communication between agents
from adk.messaging import AgentMessage

class TravelConciergeAgent(LlmAgent):
    def coordinate_rental_car(self, travel_details):
        # Send request to rental car agent
        message = AgentMessage(
            to="rental_car_agent",
            action="search_cars",
            data=travel_details
        )
        response = self.send_message(message)
        return response
```

Deliverable

- Working rental car agent with mock data
- Agent-to-agent communication
- Car search and pricing functionality

Phase 3: Integration & Workflows (Week 3)

Goals

- Implement ADK workflow orchestration
- Create end-to-end travel planning
- Add parallel task execution

Tasks

3.1 Workflow Orchestration

```python
# workflows/travel_planning_workflow.py
from adk.workflows import SequentialWorkflow, ParallelWorkflow

class TravelPlanningWorkflow(SequentialWorkflow):
    def __init__(self):
        super().__init__(
            name="travel_planning",
            agents=[
                ("parse_request", "travel_concierge"),
                ("parallel_search", ParallelWorkflow([
                    ("search_flights", "travel_concierge"),
                    ("search_hotels", "travel_concierge"),
                    ("search_cars", "rental_car_agent")
                ])),
                ("compile_plan", "travel_concierge")
            ]
        )
```

3.2 Context Sharing

```python
# shared/context.py
```

```python
from adk.context import SharedContext

class TravelContext(SharedContext):
    def __init__(self):
        super().__init__()
        self.destination = None
        self.dates = None
        self.preferences = {}
        self.flight_options = []
        self.hotel_options = []
        self.car_options = []

    def update_context(self, key, value):
        setattr(self, key, value)
        self.notify_agents(key, value)
```

3.3 End-to-End Integration

```python
# main.py
from adk import AgentSystem
from agents.travel_concierge import TravelConciergeAgent
from agents.rental_car_agent import RentalCarAgent
from workflows.travel_planning_workflow import
TravelPlanningWorkflow

def main():
    # Initialize agent system
    system = AgentSystem()

    # Register agents
    system.register_agent(TravelConciergeAgent())
    system.register_agent(RentalCarAgent())

    # Register workflow
    system.register_workflow(TravelPlanningWorkflow())

    # Handle user request
    user_request = "Plan a 3-day trip to San Francisco from
December 15-18, need a rental car"
    result = system.execute_workflow("travel_planning",
user_request)

    print(result)

if __name__ == "__main__":
    main()
```

Deliverable

- Complete workflow orchestration
- Parallel task execution
- Shared context between agents

Phase 4: Real API Integration (Week 4)

Goals

- Replace mock data with real APIs
- Implement error handling and retries
- Add booking capabilities

Suggested APIs

Travel APIs

- **Amadeus API**: Flight and hotel search
- **Skyscanner API**: Flight comparison
- **Booking.com API**: Hotel availability
- **OpenWeatherMap**: Weather data

Rental Car APIs

- **Kayak API**: Car rental aggregation
- **Hertz API**: Direct integration
- **Enterprise API**: Fleet management

- **Priceline API**: Multi-provider search

4.1 Real API Integration

```python
# tools/amadeus_tools.py
import amadeus

class FlightSearchTool(Tool):
    def __init__(self):
        super().__init__(name="search_flights", ...)
        self.amadeus = amadeus.Client(
            client_id='YOUR_API_KEY',
            client_secret='YOUR_API_SECRET'
        )

    def execute(self, origin, destination, departure_date, 
return_date=None):
        try:
            response = 
self.amadeus.shopping.flight_offers_search.get(
                originLocationCode=origin,
                destinationLocationCode=destination,
                departureDate=departure_date,
                returnDate=return_date,
                adults=1
            )
            return self.format_flight_results(response.data)
        except Exception as e:
            return self.handle_error(e)
```

4.2 Error Handling & Retries

```python
# utils/error_handling.py
from adk.utils import RetryPolicy
import time
```

```
class APIErrorHandler:
    def __init__(self, max_retries=3, backoff_factor=2):
        self.max_retries = max_retries
        self.backoff_factor = backoff_factor

    def execute_with_retry(self, func, *args, **kwargs):
        for attempt in range(self.max_retries):
            try:
                return func(*args, **kwargs)
            except Exception as e:
                if attempt == self.max_retries - 1:
                    raise e
                wait_time = self.backoff_factor ** attempt
                time.sleep(wait_time)
```

Deliverable

- Real API integrations
- Robust error handling
- Production-ready code

Advanced Extensions

Option 1: MCP Integration

Add Model Context Protocol for secure tool access:

```python
# mcp/travel_mcp_server.py
from mcp import Server
from mcp.types import Tool

class TravelMCPServer(Server):
    def __init__(self):
        super().__init__("travel-tools")
        self.register_tools([
            self.create_booking_tool(),
            self.create_calendar_tool()
        ])

    def create_booking_tool(self):
        return Tool(
            name="book_travel",
            description="Book confirmed travel arrangements",
            input_schema={...}
        )
```

Option 2: Advanced Multi-Agent Patterns

Implement sophisticated coordination:

```python
# patterns/auction_pattern.py
class CarRentalAuction:
    def __init__(self):
        self.participants = []  # Multiple rental car agents

    def conduct_auction(self, requirements):
        bids = []
        for agent in self.participants:
            bid = agent.submit_bid(requirements)
            bids.append(bid)

        return self.select_winner(bids)
```

Option 3: Learning & Personalization

Add user preference learning:

Python
```python
# learning/preference_engine.py
class UserPreferenceEngine:
    def __init__(self):
        self.user_profiles = {}

    def learn_from_booking(self, user_id, booking_details):
        # Update user preferences based on bookings
        # Use for future recommendations
        pass
```

Evaluation & Testing

Unit Tests

Python
```python
# tests/test_rental_car_agent.py
import unittest
from agents.rental_car_agent import RentalCarAgent

class TestRentalCarAgent(unittest.TestCase):
    def setUp(self):
        self.agent = RentalCarAgent()

    def test_car_search(self):
        result = self.agent.search_cars(
            pickup_location="SFO",
            return_location="SFO",
            dates=["2024-12-15", "2024-12-18"],
            preferences={"type": "economy"}
        )
        self.assertIsNotNone(result)
        self.assertIn("cars", result)
```

Integration Tests

```python
# tests/test_workflow_integration.py
def test_end_to_end_travel_planning():
    system = AgentSystem()
    # Test complete workflow
    result = system.execute_workflow(
        "travel_planning",
        "Plan trip to NYC Dec 20-23"
    )
    assert "flights" in result
    assert "hotels" in result
    assert "rental_cars" in result
```

Performance Evaluation

- Response time metrics
- API call efficiency
- User satisfaction scores
- Agent coordination effectiveness

Deployment Options

Local Development

```shell
# Run locally
python main.py
```

Google Cloud Deployment

```
# cloudbuild.yaml
steps:
  - name: 'gcr.io/cloud-builders/docker'
    args: ['build', '-t', 'gcr.io/project/travel-agents', '.']
  - name: 'gcr.io/cloud-builders/docker'
    args: ['push', 'gcr.io/project/travel-agents']
```

Vertex AI Agent Engine

Python
```python
# deploy/vertex_ai.py
from google.cloud import aiplatform

def deploy_to_vertex():
    aiplatform.init(project="your-project")

    endpoint = aiplatform.Endpoint.create(
        display_name="travel-agents"
    )

    model = aiplatform.Model.upload(
        display_name="travel-concierge",
        artifact_uri="gs://bucket/model"
    )
```

Success Metrics

Technical Metrics

- **Response Time**: < 10 seconds for complete travel plan
- **API Success Rate**: > 95% for all external API calls
- **Agent Coordination**: Successful task delegation in > 90% of cases

User Experience Metrics

- **Plan Completeness**: All requested components included
- **Relevance Score**: User satisfaction with recommendations
- **Booking Success**: Percentage of plans that lead to actual bookings

Learning Outcomes

- Understanding multi-agent coordination
- Experience with real API integrations
- Knowledge of workflow orchestration
- Familiarity with ADK production patterns

Next Steps

1. **Start with Phase 1** - Get basic agents working
2. **Iterate Quickly** - Add one feature at a time
3. **Test Thoroughly** - Both unit and integration tests
4. **Document Everything** - Keep track of lessons learned
5. **Extend Gradually** - Add advanced features once basics work

This project provides hands-on experience with all key ADK concepts while building something practical and extensible. The phased approach ensures steady progress and learning milestones.