# OpenAI Agents SDK Complete Guide - Beginner se Advanced (2025)

## Table of Contents

## OpenAI Agents SDK Kya Hai

OpenAI Agents SDK ek **latest** aur **production-ready** framework hai jo March 2025 mein release hui hai. Ye SDK aap ko help karti hai **intelligent AI agents** banane mein jo:

### 🚀 Key Features:

- **Lightweight**: Sirf zaroori components, no complexity
- **Production Ready**: Real applications ke liye designed
- **Multi-Agent Support**: Multiple agents ek saath kaam kar sakte hain
- **Built-in Tracing**: Debugging aur monitoring built-in hai
- **Provider Agnostic**: OpenAI ke alawa 100+ other LLMs support karta hai
- **Python-First**: Python ke natural features use karta hai

### 🔧 Core Primitives (Building Blocks):

1. **Agents**: LLMs with instructions aur tools
2. **Handoffs**: Agents ke beech control transfer
3. **Guardrails**: Input/output validation aur safety checks
4. **Tools**: Python functions ko AI tools banane ka tareeqa

### 🎯 Previous Swarm vs New Agents SDK:

| Feature | Swarm (Old) | Agents SDK (New) |
|---|---|---|
| Status | Experimental | Production Ready |
| Tracing | Basic | Advanced Built-in |
| Guardrails | No | Yes |
| Voice Support | No | Yes |
| Documentation | Limited | Complete |

## Setup aur Installation

### Prerequisites Check Karein:

bash

```bash
# Python version check (3.8+ required)
python --version

# Virtual environment banayein (recommended)
python -m venv agents_env

# Windows mein activate:
agents_env\Scripts\activate

# macOS/Linux mein activate:
source agents_env/bin/activate
```

### Installation Process:

bash

```bash
# Basic installation
pip install openai-agents

# Voice support ke saath (advanced features)
pip install 'openai-agents[voice]'

# Development dependencies (agar contribute karna hai)
pip install 'openai-agents[dev]'
```

### Environment Setup:

python

```python
# .env file banayein
OPENAI_API_KEY=your_openai_api_key_here

# Optional: Other providers ke liye
ANTHROPIC_API_KEY=your_anthropic_key
GOOGLE_API_KEY=your_google_key
```

### Verification Test:

python

```python
# test_installation.py
from agents import Agent, Runner
import os

# API key check
if not os.getenv('OPENAI_API_KEY'):
    print("❌ OPENAI_API_KEY environment variable set nahi hai!")
    exit(1)

try:
    # Simple test agent
    agent = Agent(
        name="Test Agent",
        instructions="You are a helpful test assistant."
    )

    result = Runner.run_sync(agent, "Say hello in one word")
    print(f"✅ Installation successful! Response: {result.final_output}")

except Exception as e:
    print(f"❌ Installation issue: {e}")
```

## Core Concepts Deep Dive

### 1. Agent Anatomy (Agent ki Structure)

python

```python
from agents import Agent, Runner

# Basic Agent Structure
agent = Agent(
    name="MyAgent",                    # Agent ka naam
    instructions="System prompt",      # Agent ko kya karna hai
    model="gpt-4o",                    # Konsa model use karna hai
    tools=[],                          # Kya tools available hain
    handoffs=[],                       # Kahan control transfer kar sakta hai
    guardrails=[],                     # Safety checks
    output_type=None,                  # Expected output format
    max_turns=50,                      # Maximum conversation turns
    temperature=0.7,                   # Creativity level (0-1)
    max_completion_tokens=1000,        # Response length limit
)
```

## 2. Runner Ki Working (Execution Engine)

python

```python
# Synchronous execution
result = Runner.run_sync(agent, "Your message")

# Asynchronous execution (recommended for production)
import asyncio

async def main():
    result = await Runner.run(agent, "Your message")
    print(result.final_output)

asyncio.run(main())
```

## 3. Agent Loop Ki Deep Understanding

python

```python
"""
Agent Loop Kaise Kaam Karta Hai:

1. User input liya jata hai
2. Agent ko message bheja jata hai
3. LLM response generate karta hai
4. Agar tool calls hain, tools execute hote hain
5. Agar handoff hai, control dusre agent ko transfer hota hai
6. Final output mile ya max_turns reach ho jane tak loop chalta hai
"""

# Loop ki visualization
class CustomRunner:
    def visualize_loop(self, agent, user_input):
        turn = 0
        current_agent = agent
        messages = [{"role": "user", "content": user_input}]

        while turn < agent.max_turns:
            print(f"🔄 Turn {turn + 1}: Agent '{current_agent.name}' processing...")

            # LLM call simulation
            response = self.call_llm(current_agent, messages)

            if response.has_tool_calls:
                print("🔧 Executing tools...")
                # Tool execution logic

            elif response.has_handoff:
                print(f"🤝 Handing off to {response.handoff_target}")
                # Handoff logic

            elif response.is_final:
                print("✅ Final output received!")
                return response.content

            turn += 1

        return "Max turns reached"
```

## Beginner Level - Basic Projects

### Project 1: Hello World Agent (Detailed)

python

```python
# hello_world_detailed.py
from agents import Agent, Runner
import os
from dotenv import load_dotenv

# Environment variables load karein
load_dotenv()

class HelloWorldAgent:
    def __init__(self):
        """Agent initialize karte hain detailed configuration ke saath"""
        self.agent = Agent(
            name="Hello World Assistant",
            instructions="""
            You are a friendly hello world assistant.
            Guidelines:
            - Always be polite and helpful
            - Keep responses concise but informative
            - Use emojis when appropriate
            - If asked about yourself, explain you're a demo agent
            """,
            model="gpt-4o-mini",  # Cost-effective model for beginners
            temperature=0.7,      # Balanced creativity
            max_completion_tokens=200,  # Short responses
        )

    def single_interaction(self, message):
        """Single message process karta hai"""
        try:
            print(f"📤 Sending: {message}")
            result = Runner.run_sync(self.agent, message)
            print(f"📥 Response: {result.final_output}")
            return result.final_output
        except Exception as e:
            print(f"❌ Error: {e}")
            return None

    def interactive_chat(self):
        """Interactive conversation mode"""
        print("🤖 Hello World Agent started!")
        print("💡 Type 'quit' to exit\n")

        while True:
            user_input = input("You: ")

            if user_input.lower() in ['quit', 'exit', 'bye']:
                print("👋 Goodbye!")
                break

            if not user_input.strip():
                print("⚠️ Please enter a message!")
                continue

            response = self.single_interaction(user_input)
            print()  # Empty line for readability

# Usage examples
if __name__ == "__main__":
    agent = HelloWorldAgent()

    # Single interactions
    agent.single_interaction("Hello!")
    agent.single_interaction("What can you do?")
    agent.single_interaction("Tell me a joke")

    print("\n" + "="*50 + "\n")

    # Interactive mode
    agent.interactive_chat()
```

### Project 2: Smart Calculator Agent with Tools

python

```python
# calculator_agent.py
from agents import Agent, Runner, function_tool
import math
import asyncio

# Tool functions define karte hain
@function_tool
def add(a: float, b: float) -> float:
    """Add two numbers"""
    result = a + b
    print(f"🧮 Adding {a} + {b} = {result}")
    return result

@function_tool
def subtract(a: float, b: float) -> float:
    """Subtract second number from first"""
    result = a - b
    print(f"🧮 Subtracting {a} - {b} = {result}")
    return result

@function_tool
def multiply(a: float, b: float) -> float:
    """Multiply two numbers"""
    result = a * b
    print(f"🧮 Multiplying {a} × {b} = {result}")
    return result

@function_tool
def divide(a: float, b: float) -> float:
    """Divide first number by second"""
    if b == 0:
        raise ValueError("Cannot divide by zero!")
    result = a / b
    print(f"🧮 Dividing {a} ÷ {b} = {result}")
    return result

@function_tool
def power(base: float, exponent: float) -> float:
    """Calculate base raised to the power of exponent"""
    result = base ** exponent
    print(f"🧮 Power {base}^{exponent} = {result}")
    return result

@function_tool
def square_root(number: float) -> float:
    """Calculate square root of a number"""
    if number < 0:
        raise ValueError("Cannot calculate square root of negative number!")
    result = math.sqrt(number)
    print(f"🧮 Square root of {number} = {result}")
    return result

@function_tool
def factorial(n: int) -> int:
    """Calculate factorial of a number"""
    if n < 0:
        raise ValueError("Factorial is not defined for negative numbers!")
    if n > 20:  # Prevent very large calculations
        raise ValueError("Number too large for factorial calculation!")

    result = math.factorial(n)
    print(f"🧮 Factorial {n}! = {result}")
    return result

class SmartCalculator:
    def __init__(self):
        """Advanced calculator agent with multiple mathematical tools"""
        self.agent = Agent(
            name="Smart Calculator",
            instructions="""
            You are an advanced mathematical calculator assistant.

            Capabilities:
            - Basic arithmetic (add, subtract, multiply, divide)
            - Advanced operations (power, square root, factorial)
            - Step-by-step explanations
            - Error handling and validation

            Guidelines:
            - Always use the appropriate tool for calculations
            - Explain your work step by step
            - Handle errors gracefully
            - If user asks for explanation, provide mathematical context
            - Use emojis to make responses engaging
            """,
            tools=[
                add, subtract, multiply, divide,
                power, square_root, factorial
            ],
            model="gpt-4o-mini",
            temperature=0.1,  # Low temperature for accuracy
        )
    async def calculate(self, expression):
        """Single calculation perform karta hai"""
        try:
            print(f"📊 Calculating: {expression}")
            result = await Runner.run(self.agent, expression)
            print(f"✅ Final Result: {result.final_output}")
            return result.final_output
        except Exception as e:
            print(f"❌ Calculation Error: {e}")
            return f"Error: {e}"

    async def interactive_calculator(self):
        """Interactive calculator mode"""
        print("🧮 Smart Calculator Agent Started!")
        print("💡 Examples:")
        print("   - Add 15 and 25")
        print("   - What is 2 to the power of 8?")
        print("   - Calculate factorial of 5")
        print("   - Find square root of 144")
        print("📝 Type 'quit' to exit\n")

        while True:
            user_input = input("Math Problem: ")

            if user_input.lower() in ['quit', 'exit', 'bye']:
                print("👋 Calculator closing...")
                break

            if not user_input.strip():
                print("⚠ Please enter a math problem!")
                continue

            await self.calculate(user_input)
            print("-" * 40)

# Advanced usage examples
async def demo_calculations():
    """Different types of calculations demonstrate karta hai"""
    calc = SmartCalculator()

    test_cases = [
        "Add 123 and 456",
        "What is 12 squared?",
        "Calculate the factorial of 7",
        "Find the square root of 169",
        "Divide 100 by 7 and round to 2 decimal places",
        "What is 2 to the power of 10?",
        "Solve: (5 + 3) × 2 - 4",
    ]

    print("🧪 Running Demo Calculations:")
    print("=" * 50)

    for i, test in enumerate(test_cases, 1):
        print(f"\n📋 Test {i}: {test}")
        await calc.calculate(test)
        print("-" * 30)

# Main execution
if __name__ == "__main__":
    calc = SmartCalculator()

    # Choose mode
    mode = input("Select mode (1: Demo, 2: Interactive): ")

    if mode == "1":
        asyncio.run(demo_calculations())
    else:
        asyncio.run(calc.interactive_calculator())
```

## Project 3: Personal Assistant Agent

python

```python
# personal_assistant.py
from agents import Agent, Runner, function_tool
from datetime import datetime, timedelta
import json
import asyncio

# Personal Assistant Tools
@function_tool
def get_current_time() -> str:
    """Get current date and time"""
    now = datetime.now()
    return now.strftime("%Y-%m-%d %H:%M:%S")

@function_tool
def add_reminder(task: str, due_date: str = None) -> str:
    """Add a reminder/task to the list"""
    reminder = {
        "task": task,
        "created": datetime.now().isoformat(),
        "due_date": due_date,
        "completed": False
    }

    # Simple file-based storage (production mein database use karein)
    try:
        with open("reminders.json", "r") as f:
            reminders = json.load(f)
    except FileNotFoundError:
        reminders = []

    reminders.append(reminder)

    with open("reminders.json", "w") as f:
        json.dump(reminders, f, indent=2)

    return f"✅ Reminder added: {task}"

@function_tool
def get_reminders() -> str:
    """Get all active reminders"""
    try:
        with open("reminders.json", "r") as f:
            reminders = json.load(f)
    except FileNotFoundError:
        return "📝 No reminders found"
    active_reminders = [r for r in reminders if not r["completed"]]

    if not active_reminders:
        return "✅ No pending reminders!"

    result = "📋 Your Reminders:\n"
    for i, reminder in enumerate(active_reminders, 1):
        due_info = f" (Due: {reminder['due_date']})" if reminder['due_date'] else ""
        result += f"{i}. {reminder['task']}{due_info}\n"

    return result

@function_tool
def calculate_age(birth_year: int) -> str:
    """Calculate age from birth year"""
    current_year = datetime.now().year
    age = current_year - birth_year
    return f"You are {age} years old"

@function_tool
def weather_info(city: str) -> str:
    """Get weather information (mock function)"""
    # Real implementation mein weather API use karein
    mock_weather = {
        "karachi": "⛅ Partly cloudy, 28°C",
        "lahore": "☀️ Sunny, 32°C",
        "islamabad": "🌧️ Light rain, 22°C",
        "peshawar": "☀️ Clear, 35°C"
    }

    city_lower = city.lower()
    if city_lower in mock_weather:
        return f"Weather in {city}: {mock_weather[city_lower]}"
    else:
        return f"Weather data for {city} not available. Try: Karachi, Lahore, Islamabad, Peshawar"

@function_tool
def unit_converter(value: float, from_unit: str, to_unit: str) -> str:
    """Convert between different units"""
    conversions = {
        # Length
        ("meter", "feet"): 3.28084,
        ("feet", "meter"): 0.3048,
        ("km", "miles"): 0.621371,
        ("miles", "km"): 1.60934,

        # Weight
        ("kg", "pounds"): 2.20462,
        ("pounds", "kg"): 0.453592,

        # Temperature (special handling needed)
        ("celsius", "fahrenheit"): lambda c: (c * 9/5) + 32,
        ("fahrenheit", "celsius"): lambda f: (f - 32) * 5/9,
    }

    key = (from_unit.lower(), to_unit.lower())

    if key in conversions:
        converter = conversions[key]
        if callable(converter):
            result = converter(value)
        else:
            result = value * converter

        return f"{value} {from_unit} = {result:.2f} {to_unit}"
    else:
        return f"Conversion from {from_unit} to {to_unit} not supported"

class PersonalAssistant:
    def __init__(self):
        """Comprehensive personal assistant agent"""
        self.agent = Agent(
            name="Personal Assistant",
            instructions="""
            You are a helpful personal assistant named Alex.

            Your capabilities:
            📅 Time & Date: Current time, date calculations
            📝 Task Management: Add reminders, view tasks
            🌤 Weather: Get weather information for Pakistani cities
            🧮 Calculations: Age calculation, unit conversions
            💬 General Help: Answer questions, provide information

            Personality:
            - Friendly and professional
            - Proactive in offering help
            - Use appropriate emojis
            - Ask clarifying questions when needed
            - Remember context within conversation

            Guidelines:
            - Always greet users warmly
            - Use tools when appropriate
            - Provide helpful suggestions
```

```python
            - Be concise but thorough
            - Handle errors gracefully
            """,
            tools=[
                get_current_time,
                add_reminder,
                get_reminders,
                calculate_age,
                weather_info,
                unit_converter
            ],
            model="gpt-4o",
            temperature=0.7,
        )

    async def process_request(self, user_input):
        """User request process karta hai"""
        try:
            result = await Runner.run(self.agent, user_input)
            return result.final_output
        except Exception as e:
            return f"❌ Sorry, I encountered an error: {e}"

    async def start_assistant(self):
        """Interactive personal assistant start karta hai"""
        print("🤖 Personal Assistant Alex is ready!")
        print("\n💡 I can help you with:")
        print("  📅 Time and date queries")
        print("  📝 Managing reminders and tasks")
        print("  🌤️ Weather information")
        print("  🧮 Calculations and conversions")
        print("  💬 General questions and assistance")
        print("\n🗣️ Try saying:")
        print("  - 'What time is it?'")
        print("  - 'Add reminder to call mom tomorrow'")
        print("  - 'What's the weather in Karachi?'")
        print("  - 'Convert 100 km to miles'")
        print("\n💬 Type 'quit' to exit\n")

        conversation_history = []

        while True:
            user_input = input("You: ")

            if user_input.lower() in ['quit', 'exit', 'bye', 'goodbye']:
                farewell_messages = [
                    "👋 Goodbye! Have a great day!",
                    "🌟 Take care! I'm here whenever you need help.",
                    "👋 See you later! Stay awesome!"
                ]
                import random
                print(random.choice(farewell_messages))
                break

            if not user_input.strip():
                print("🤔 I'm listening... what can I help you with?")
                continue

            print("🤖 Alex: ", end="", flush=True)
            response = await self.process_request(user_input)
            print(response)
            print()  # Empty line for readability

            # Conversation history maintain karein (optional)
            conversation_history.append({
                "user": user_input,
                "assistant": response,
                "timestamp": datetime.now().isoformat()
            })

# Demo scenarios
async def demo_scenarios():
    """Different use cases demonstrate karta hai"""
    assistant = PersonalAssistant()

    scenarios = [
        "What time is it right now?",
        "Add reminder to submit project report by Friday",
        "Show me my reminders",
        "What's the weather like in Lahore?",
        "Convert 75 kg to pounds",
        "Calculate my age if I was born in 1995",
        "Convert 100 degrees Fahrenheit to Celsius"
    ]

    print("🎭 Demo Scenarios:")
    print("=" * 50)

    for i, scenario in enumerate(scenarios, 1):
        print(f"\n📋 Scenario {i}: {scenario}")
        response = await assistant.process_request(scenario)
        print(f"🤖 Alex: {response}")
        print("-" * 40)
```

```python
# Main execution
if __name__ == "__main__":
    assistant = PersonalAssistant()

    print("🚀 Personal Assistant Demo")
    mode = input("Select mode (1: Demo Scenarios, 2: Interactive): ")

    if mode == "1":
        asyncio.run(demo_scenarios())
    else:
        asyncio.run(assistant.start_assistant())
```

## Intermediate Level - Multi-Agent Systems

### Project 4: Customer Support System

python

```python
# customer_support_system.py
from agents import Agent, Runner, function_tool
import asyncio
import json
from datetime import datetime
from typing import List, Dict

# Customer Support Tools
@function_tool
def search_knowledge_base(query: str) -> str:
    """Search the knowledge base for information"""
    # Mock knowledge base (production mein real database use karein)
    knowledge_base = {
        "password reset": "To reset your password: 1) Go to login page 2) Click 'Forgot Password' 3) Enter your email 4) Ch
        "billing": "For billing inquiries, you can view your invoices in Account Settings > Billing section. Contact billing@cor
        "technical issue": "Please try these steps: 1) Clear browser cache 2) Disable browser extensions 3) Try incognito mo
        "account deletion": "To delete your account: 1) Go to Settings > Account 2) Click 'Delete Account' 3) Confirm via er
        "refund": "Refund requests can be made within 30 days of purchase. Go to Account > Orders > Request Refund or
    }

    query_lower = query.lower()
    for key, value in knowledge_base.items():
        if key in query_lower:
            return f"📚 Found relevant information: {value}"

    return "❓ No specific information found. Please provide more details or contact a human agent."

@function_tool
def create_ticket(issue_type: str, description: str, priority: str = "medium") -> str:
    """Create a support ticket"""
    ticket_id = f"TICK-{datetime.now().strftime('%Y%m%d-%H%M%S')}"

    ticket = {
        "id": ticket_id,
        "type": issue_type,
        "description": description,
        "priority": priority,
        "status": "open",
        "created": datetime.now().isoformat(),
        "assigned_to": None
    }

    # Save ticket (production mein database use karein)
    try:
        with open("tickets.json", "r") as f:
            tickets = json.load(f)
    except FileNotFoundError:
        tickets = []

    tickets.append(ticket)

    with open("tickets.json", "w") as f:
        json.dump(tickets, f, indent=2)

    return f"🎫 Support ticket created: {ticket_id}. You'll receive updates via email."

@function_tool
def escalate_to_human(reason: str) -> str:
    """Escalate the conversation to a human agent"""
    return f"🔄 Escalating to human agent. Reason: {reason}. Average wait time: 5-10 minutes. Please stay connected."

class CustomerSupportSystem:
    def __init__(self):
        """Multi-agent customer support system"""

        # Triage Agent - First point of contact
        self.triage_agent = Agent(
            name="Support Triage",
            instructions="""
            You are the first-line customer support triage agent.

            Your role:
            🎯 Understand customer issues quickly
            🔍 Search knowledge base for solutions
            📋 Categorize issues (technical, billing, account, general)
            🎫 Create tickets for complex issues
            🔄 Escalate to specialized agents when needed

            Guidelines:
            - Be empathetic and professional
            - Ask clarifying questions if issue is unclear
            - Try to resolve simple issues immediately
            - Use appropriate handoffs for specialized problems
            - Always acknowledge customer frustration

            Handoff Criteria:
            - Technical Agent: Complex technical issues, bugs, integrations
            - Billing Agent: Payment disputes, refund requests, subscription issues
            - Human Agent: Angry customers, policy exceptions, legal issues
            """,
            tools=[search_knowledge_base, create_ticket, escalate_to_human],
            handoffs=[],  # Will be set after creating other agents
            model="gpt-4o",
    temperature=0.6,
)

# Technical Support Agent
self.technical_agent = Agent(
    name="Technical Support",
    instructions="""
    You are a specialized technical support agent.

    Expertise:
    🔧 API integrations and troubleshooting
    🐛 Bug investigation and workarounds
    ⚙️ System configuration and setup
    📊 Performance optimization
    🔐 Security and authentication issues

    Approach:
    - Ask for specific technical details (error codes, browser, OS)
    - Provide step-by-step troubleshooting guides
    - Offer workarounds for known issues
    - Create detailed tickets for bugs
    - Know when to escalate to development team

    Communication style:
    - Technical but accessible language
    - Include screenshots/logs requests when needed
    - Provide relevant documentation links
    """,
    tools=[search_knowledge_base, create_ticket],
    model="gpt-4o",
    temperature=0.3,  # Lower temperature for technical accuracy
)

# Billing Support Agent
self.billing_agent = Agent(
    name="Billing Support",
    instructions="""
    You are a specialized billing and payments support agent.

    Expertise:
    💳 Payment processing issues
    📄 Invoice and billing inquiries
    💰 Refund and credit processing
    📅 Subscription management
    🔄 Plan upgrades and downgrades

    Guidelines:
            - Handle sensitive financial information carefully
            - Explain billing policies clearly
            - Process refunds within policy guidelines
            - Escalate policy exceptions to supervisors
            - Provide detailed invoice breakdowns

            Security:
            - Never ask for full credit card numbers in chat
            - Verify customer identity before accessing billing info
            - Follow PCI compliance guidelines
            """,
            tools=[search_knowledge_base, create_ticket],
            model="gpt-4o",
            temperature=0.4,
        )

        # Set up handoffs after all agents are created
        self.triage_agent.handoffs = [
            self.technical_agent,
            self.billing_agent
        ]

    async def handle_support_request(self, customer_message: str, customer_context: Dict = None):
        """Main entry point for customer support requests"""

        # Add customer context to the message if available
        if customer_context:
            context_info = f"""
            Customer Information:
            - Name: {customer_context.get('name', 'N/A')}
            - Account Type: {customer_context.get('account_type', 'N/A')}
            - Previous Tickets: {customer_context.get('ticket_count', 0)}

            Customer Issue: {customer_message}
            """
        else:
            context_info = f"Customer Issue: {customer_message}"

        try:
            result = await Runner.run(self.triage_agent, context_info)
            return result.final_output
        except Exception as e:
            return f"❌ System error occurred. Please try again or contact support directly. Error: {e}"

    async def interactive_support_demo(self
```