🤝 Agentic AI Workflows

Stage 6: Multi-Agent Systems with Autonomous Execution

The cutting edge of AI — systems that can plan, use tools, collaborate, and execute actions autonomously. This guide covers everything from concepts to production deployment.

🎯 What Are Agentic AI Workflows?

TL;DR
Agentic AI moves from "suggest solutions" to "fix problems." Instead of a single LLM generating text, you have multiple specialized agents working together, each with tool access, that can plan multi-step workflows and execute actions autonomously.

The Evolution: From Text Generation to Action

❌ Previous Approaches (Stages 1-5)

- **Rule-Based:** "Here's a canned response"
- **ML Classifier:** "This is category X"
- **GenAI Prompting:** "Here's what to do"
- **RAG:** "According to docs, do X"
- **Fine-Tuning:** "Consistently says to do X"

👤 Human Still Does the Work

✅ Agentic AI (Stage 6)

- **Analyzes:** "Let me diagnose the issue"
- **Retrieves:** "I'll check the runbook"
- **Plans:** "Here's my 3-step approach"
- **Executes:** "Running schema sync..."
- **Verifies:** "Checking if fix worked..."
- **Adapts:** "That didn't work, trying plan B"

🤖 Agent Does the Work

Core Concepts

🤖 Agents

Specialized AI entities with specific roles, powered by LLMs, that can reason and make decisions.

Example: DiagnosticAgent, KnowledgeAgent, ResolutionAgent

🔧 Tools

Functions/APIs that agents can call to interact with the world (databases, APIs, scripts).

Example: query_logs(), restart_pipeline(), send_email()

🎯 Orchestration

Coordination logic that decides which agents run, in what order, and how they collaborate.

Patterns: Sequential, Parallel, Conditional, Hierarchical

🧠 Planning

Agents break down complex tasks into subtasks and determine execution strategy.

Technique: Chain-of-thought, ReAct (Reasoning + Acting)

🔄 Self-Correction

Agents verify results, detect failures, and retry with different approaches.

Example: If fix fails → try alternative → escalate if all fail

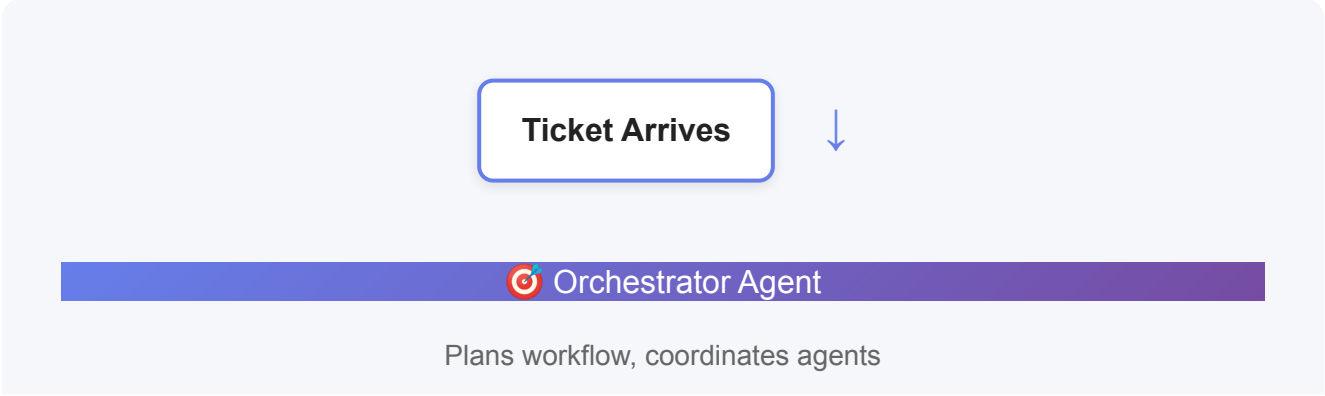
📝 Memory

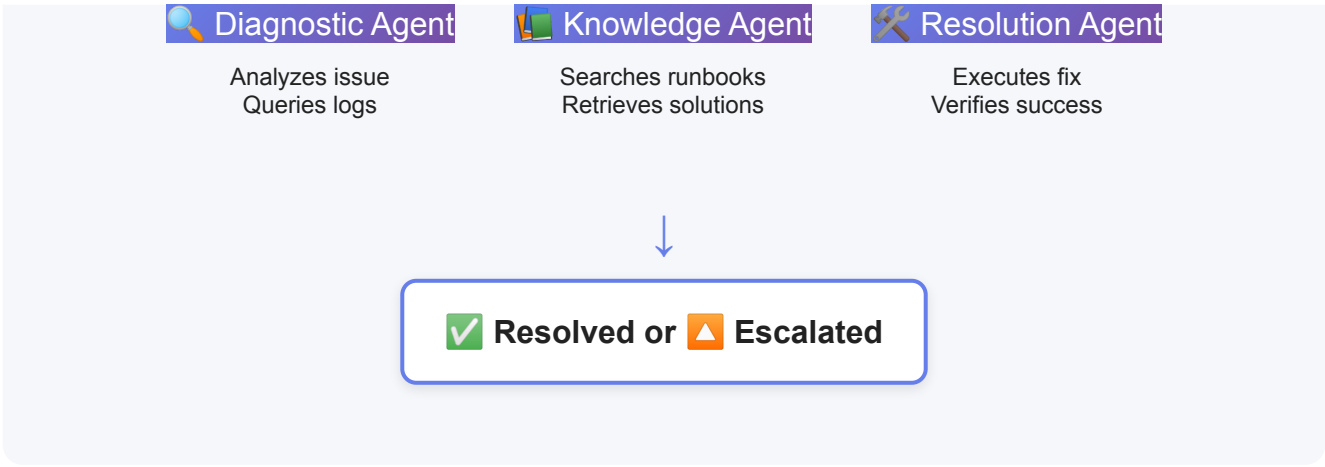
Agents maintain context across interactions (short-term) and learn from past experiences (long-term).

Types: Conversation history, task results, learned patterns

🏗️ Architecture for L1 IT Support

Multi-Agent System Design





Agent Workflow Example

1 Orchestrator Receives Ticket

Ticket: "Pipeline job-456 failed with schema mismatch error"

Decision: This requires diagnostic → knowledge → resolution flow

2 Diagnostic Agent Analyzes

Tool calls: query_logs(job_id="job-456"), get_pipeline_config()

Finding: "Column 'user_email' exists in source but not in target table 'users_prod'"

3 Knowledge Agent Retrieves Solution

Tool calls: vector_search(query="schema mismatch solution")

Found: Runbook SOP-451: "Run schema_sync_job for affected table, then restart pipeline"

4 Resolution Agent Executes Fix

Tool calls:

- run_schema_sync(table="users_prod") → Success ✓
- restart_pipeline(job_id="job-456") → Started ✓
- verify_job_status(job_id="job-456", timeout=300) → Success ✓

5 Orchestrator Reports Results

Status: ✅ Auto-Resolved
Time: 8 minutes
Actions Taken: Schema sync executed, pipeline restarted, verified successful
Next Steps: None - ticket closed

💻 Implementation with Google ADK

Setup and Installation

```
# Install ADK (Agentic Development Kit) pip install google-adk --break-system-packages # Install dependencies pip install google-generativeai chromadb --break-system-packages # Set up authentication export GOOGLE_API_KEY="your-gemini-api-key"
```

1. Define Tools (Agent Actions)

```
from adk import Tool from typing import Dict, Any # Tool 1: Query logs to diagnose issues def query_logs(job_id: str, last_n_lines: int = 100) -> str: """ Query logs for a specific job to identify errors. Args: job_id: The pipeline job ID last_n_lines: Number of recent log lines to retrieve Returns: Log content as string """ # In production, this would connect to actual logging system # For demo, return mock data return f""" [ERROR] Job {job_id} failed at 10:45:23 [ERROR] Schema validation failed [ERROR] Column 'user_email' not found in target table 'users_prod' [INFO] Source table has columns: id, user_email, created_at [INFO] Target table has columns: id, email, created_at """ # Tool 2: Get pipeline configuration def get_pipeline_config(job_id: str) -> Dict[str, Any]: """Retrieve pipeline configuration details""" return { "job_id": job_id, "source_table": "staging.users", "target_table": "prod.users_prod", "schema_validation": "strict", "timeout": 1800 } # Tool 3: Vector search for solutions def vector_search(query: str, top_k: int = 3) -> List[Dict]: """Search knowledge base for relevant solutions""" # In production, use
```

```
ChromaDB or similar return [ { "doc_id": "SOP-451", "title": "Schema Mismatch Resolution", "content": "Run schema_sync_job for affected table, then restart pipeline", "confidence": 0.95 } ] # Tool 4: Execute schema sync def run_schema_sync(table: str) -> Dict[str, Any]: """ Run schema synchronization job for a table. Args: table: Table name (e.g., 'users_prod') Returns: Status dict with success/failure """ # In production, call actual schema sync API print(f"🔧 Running schema sync for {table}...") return {"status": "success", "columns_synced": 1, "time_taken": "3.2s"} # Tool 5: Restart pipeline def restart_pipeline(job_id: str) -> Dict[str, Any]: """Restart a failed pipeline job""" print(f"🔄 Restarting pipeline {job_id}...") return {"status": "started", "new_run_id": f"{job_id}-retry-1"} # Tool 6: Verify job success def verify_job_status(job_id: str, timeout: int = 300) -> Dict[str, Any]: """Poll job status until completion or timeout""" print(f"⏳ Verifying job {job_id}...") # In production, poll actual job status return {"status": "success", "rows_processed": 15420, "duration": "4.5min"} # Register tools with ADK tools = [ Tool("query_logs", query_logs), Tool("get_pipeline_config", get_pipeline_config), Tool("vector_search", vector_search), Tool("run_schema_sync", run_schema_sync), Tool("restart_pipeline", restart_pipeline), Tool("verify_job_status", verify_job_status) ]
```

2. Create Specialized Agents

```
from adk import Agent # Agent 1: Diagnostic - Analyzes issues diagnostic_agent = Agent( name="DiagnosticAgent", role="""You are a diagnostic specialist for a banking data platform. Your job is to analyze support tickets and determine root causes. Use available tools to: 1. Query logs to find error messages 2. Get pipeline configurations 3. Identify the specific issue (schema, timeout, data quality, etc.) Output a structured diagnosis with: - Issue category - Root cause - Affected components - Severity level """, model="gemini-1.5-pro", tools= ["query_logs", "get_pipeline_config"], temperature=0.1 # Low temp for factual analysis ) # Agent 2: Knowledge - Retrieves solutions knowledge_agent = Agent( name="KnowledgeAgent", role="""You are a knowledge retrieval specialist. Your job is to search runbooks and documentation for solutions. Given a diagnosis, use vector search to find: 1. Standard Operating Procedures (SOPs) 2. Previous similar incidents 3. Step-by-step resolution guides Output: - Relevant SOP reference - Recommended solution steps - Estimated resolution time """, model="gemini-1.5-flash", # Faster model for retrieval tools=["vector_search"], temperature=0.0 # Zero temp for retrieval ) # Agent 3: Resolution - Executes fixes resolution_agent = Agent( name="ResolutionAgent",
```

```
role="""You are a resolution execution specialist for banking systems.
Your job is to execute fixes and verify they worked. CRITICAL SAFETY
RULES: 1. Only execute pre-approved actions (schema sync, restart, config
changes) 2. Never delete or modify data without explicit approval 3.
Verify each step succeeded before proceeding 4. If any step fails, STOP
and escalate 5. Log all actions taken Given a solution plan: 1. Execute
steps in order 2. Verify each step completed successfully 3. Perform final
validation 4. Report results with evidence """, model="gemini-1.5-pro",
tools=["run_schema_sync", "restart_pipeline", "verify_job_status"],
temperature=0.1, safety_settings={"require_human_approval": ["delete",
"modify_data"]} # Extra safety )
```

3. Orchestrate Multi-Agent Workflow

```
from adk import Orchestrator, WorkflowType # Create orchestrator with
sequential workflow orchestrator = Orchestrator(
name="L1SupportOrchestrator", agents=[diagnostic_agent, knowledge_agent,
resolution_agent], workflow=WorkflowType.SEQUENTIAL, # Run agents in order
max_iterations=5, # Allow retry loops enable_memory=True # Remember past
interactions ) # Process a ticket async def
process_ticket(ticket_description: str, ticket_id: str): """ Main entry
point for ticket processing. This orchestrates the full workflow: 1.
Diagnostic agent analyzes 2. Knowledge agent finds solution 3. Resolution
agent executes fix """ print(f"\n Processing Ticket {ticket_id}\n")
print(f"Description: {ticket_description}\n") # Run orchestrated workflow
result = await orchestrator.run( task=ticket_description, context={
"ticket_id": ticket_id, "timestamp": datetime.now().isoformat(),
"priority": "high" } ) # Extract results print(f"\n{'='*80}")
print(f"RESULTS FOR TICKET {ticket_id}") print(f"{'='*80}\n")
print(f"Status: {result.status}") # 'resolved' or 'escalated' print(f"Time
Taken: {result.duration_seconds}s") print(f"Actions Performed:
{len(result.actions)}") for agent_result in result.agent_results:
print(f"\n{agent_result.agent_name}:") print(f" {agent_result.output}")
return result # Example usage import asyncio ticket = "Pipeline job-456
failed with schema mismatch error. Column 'user_email' not found in target
table." result = asyncio.run(process_ticket(ticket, "TKT-001"))
```

🎨 Advanced Patterns

Pattern 1: Conditional Orchestration

> **Use Case:** Different agents for different ticket types

```
# Define conditional workflow def ticket_router(ticket: str) ->
List[Agent]: """Route tickets to appropriate agents based on type""" if
"schema" in ticket.lower() or "pipeline" in ticket.lower(): # Technical
issue - full diagnostic workflow return [diagnostic_agent,
knowledge_agent, resolution_agent] elif "access" in ticket.lower() or
"permission" in ticket.lower(): # Access request - skip diagnostic, go
straight to resolution return [permission_agent] elif "slow" in
ticket.lower() or "performance" in ticket.lower(): # Performance issue -
need specialized optimization agent return [diagnostic_agent,
performance_agent, resolution_agent] else: # Unknown - start with triage
agent return [triage_agent, diagnostic_agent, knowledge_agent,
resolution_agent] # Use dynamic routing orchestrator = Orchestrator(
name="AdaptiveOrchestrator", agent_selector=ticket_router, # Dynamic
selection function workflow=WorkflowType.DYNAMIC )
```

Pattern 2: Parallel Execution

> **Use Case:** Run multiple agents simultaneously for speed

```
# Run diagnostic and knowledge retrieval in parallel parallel_orchestrator
= Orchestrator( name="ParallelOrchestrator", agents=[ [diagnostic_agent,
knowledge_agent], # These run in parallel resolution_agent # This runs
after both complete ], workflow=WorkflowType.PARALLEL_THEN_SEQUENTIAL ) #
Result: Saves ~3-5 seconds by running diagnostic + search concurrently
```

Pattern 3: Self-Correction Loop

```
# Agent that retries with different approaches class
SelfCorrectingAgent(Agent): def __init__(self, **kwargs):
super().__init__(**kwargs) self.max_retries = 3 async def
execute_with_retry(self, task): for attempt in range(self.max_retries):
result = await self.execute(task) # Verify result verification = await
self.verify(result) if verification.success: return result else: print(f"⚠
Attempt {attempt+1} failed: {verification.error}") print(f"🔄 Trying
alternative approach...") # Adjust strategy based on error task =
self.adapt_strategy(task, verification.error) # All retries failed -
escalate return {"status": "escalate", "reason": "All retry attempts
failed"}
```

Pattern 4: Human-in-the-Loop

```
# Agent that requests approval for critical actions class
SafeResolutionAgent(Agent): def __init__(self, **kwargs):
super().__init__(**kwargs) self.critical_actions = ['delete',
'modify_prod_data', 'change_permissions'] async def execute_action(self,
action_name, params): if action_name in self.critical_actions: # Request
human approval approval = await self.request_approval( action=action_name,
params=params, reason=f"Critical action requires approval per banking
policy" ) if not approval.granted: return {"status": "blocked", "reason":
approval.rejection_reason} # Execute the action return await
self.tool_manager.execute(action_name, params)
```

⚙️ Production Considerations

⚠️ **Critical for Banking:** Agentic AI systems require robust safety measures, monitoring, and governance before production deployment.

Safety & Guardrails

🛡️ Action Whitelisting

Only allow agents to execute pre-approved actions. Maintain explicit allow-list.

```
allowed_actions = [ "query_logs", "restart_pipeline", "run_schema_sync",
"send_notification" ] # Block everything else
```

🔐 Parameter Validation

Validate all parameters before execution. Prevent injection attacks.

```
def validate_params(action, params): if action == "restart_pipeline": assert
"job_id" in params assert re.match(r"^job-\d+$", params["job_id"])
```

📝 Audit Logging

Log every agent decision and action for compliance and debugging.

```
audit_log.record({ "timestamp": now(), "agent": agent.name, "action":
action_name, "params": params, "result": result })
```

🚨 Circuit Breakers

Stop execution if costs, errors, or latency exceed thresholds.

```
if cost_this_hour > BUDGET_LIMIT: raise CircuitBreakerError( "Cost limit
exceeded" )
```

Monitoring & Observability

Metric	What to Track	Alert Threshold

Success Rate	% of tickets auto-resolved	Alert if drops below 60%
Latency (P95)	Time to resolution	Alert if exceeds 30 seconds
Tool Call Rate	API calls per ticket	Alert if exceeds 20 calls
Error Rate	Failed actions / total actions	Alert if exceeds 5%
Cost per Ticket	LLM API costs	Alert if exceeds $0.50
Escalation Rate	Tickets sent to humans	Alert if exceeds 40%

Cost Management

Cost Breakdown for 500 Tickets/Month:

Scenario 1: Full Agentic Workflow (All tickets)
• Diagnostic Agent: 500 tickets × $0.01/call = $5
• Knowledge Agent: 500 tickets × $0.005/call = $2.50
• Resolution Agent: 200 auto-resolved × $0.02/call = $4
• Tool Calls: 500 tickets × 5 avg calls × $0.001 = $2.50
Total: ~$14/month (much cheaper than expected!)

Scenario 2: Hybrid Approach (Smart routing)
• Simple tickets (30%): Rule-based (free)
• Medium complexity (40%): RAG ($2)
• Complex (30%): Full agentic ($4.20)
Total: ~$6.20/month

Key Insight: Agentic AI is more affordable than expected if you use efficient models (Gemini Flash) and optimize tool calls.

✅ Production Readiness Checklist

🏗️ Architecture

- [ ] Agents are single-responsibility (one clear job each)
- [ ] Tools are atomic and well-tested

- [ ] Orchestration logic is documented and version-controlled
- [ ] Error handling covers all failure modes

🛡️ Safety

- [ ] Action whitelist defined and enforced
- [ ] Parameter validation on all tool calls
- [ ] Human-in-the-loop for critical actions
- [ ] Rollback capability for all state changes
- [ ] Rate limiting to prevent runaway costs

📊 Monitoring

- [ ] Success/failure metrics tracked
- [ ] Latency monitoring (P50, P95, P99)
- [ ] Cost tracking per ticket
- [ ] Audit logs for all agent actions
- [ ] Alerts configured for anomalies

🧪 Testing

- [ ] Unit tests for each tool
- [ ] Integration tests for agent workflows
- [ ] Chaos testing (what if tool fails?)
- [ ] Performance testing with realistic load
- [ ] Regression tests for edge cases

📚 Documentation

- [ ] Agent responsibilities documented
- [ ] Tool API contracts defined
- [ ] Runbook for common issues
- [ ] Incident response playbook
- [ ] Training materials for team

🔐 Compliance (Banking)

- [ ] Data retention policies followed
- [ ] PII handling compliant
- [ ] Audit trail meets regulatory requirements
- [ ] Access controls implemented
- [ ] Security review completed

🚀 Getting Started: 4-Week Plan

W1 Week 1: Learn Fundamentals

- Read this guide thoroughly
- Complete ADK tutorials
- Build simple single-agent system (no orchestration yet)
- Experiment with tool calling

Deliverable: Single agent that can query logs and generate reports

W2 Week 2: Build Multi-Agent System

- Create 3 specialized agents (diagnostic, knowledge, resolution)
- Implement simple sequential orchestration
- Add tool functions for your use case
- Test with 10-20 tickets

Deliverable: Working multi-agent prototype

W3 Week 3: Add Safety & Monitoring

- Implement action whitelisting
- Add parameter validation
- Set up audit logging
- Create monitoring dashboard
- Add human-in-the-loop for critical actions

Deliverable: Production-ready safety controls

W4 Week 4: Test & Deploy

- Run comprehensive testing (unit, integration, chaos)
- Pilot with 10% of tickets
- Monitor metrics and iterate
- Document learnings and best practices
- Present to team

Deliverable: Production deployment + team presentation

📚 Additional Resources

Documentation

- Google ADK Docs
- Gemini API Reference
- LangChain Agents

Code Examples

- Full Implementation
- Tool Definitions
- Orchestration Logic

Related Guides

- AI Evolution Overview
- RAG Implementation
- Fine-Tuning Guide

Community

- ADK GitHub
- Gemini Discord
- Internal: #ai-agents Slack channel
