



# AI Evolution: From Rules to Agents

A Complete Learning Guide for Data & ML Engineers

Understand the 40-year journey of AI systems through a single, practical use case:

**Building an L1 IT Support Agent for Banking AI & Data Platform**

## 6 Stages

Rule-Based → Agentic AI

## 1 Use Case

Throughout All Stages

## 4 Weeks

Structured Learning Plan



## Why This Guide Exists

### TL;DR

Your team needs to understand HOW and WHEN to use AI approaches. This guide shows the evolution from 1980s rule-based systems to 2024 multi-agent workflows using ONE consistent use case, so you can compare approaches directly and make informed decisions.

## The Problem We're Solving

Recent training failed because:

- **No depth:** Covered concepts superficially without real understanding
- **No connections:** Each technique presented in isolation, no clear evolution
- **No practical examples:** Theory without code or real-world context
- **No decision framework:** When to use what? Why did each approach emerge?

## What Makes This Different

### One Use Case Throughout

L1 IT Support Agent handles data pipeline failures, access requests, query optimization. You'll see how EACH stage tackles the same 500+ tickets/month problem.

### Real Metrics

Not just theory — see actual resolution rates, latency, costs, and development time for each approach.

### Working Code

Runnable Python examples using Google ADK and Gemini. Not pseudocode — actual implementations you can test.

### Clear Evolution

Understand WHY each innovation was needed and what problems it solved from the previous stage.



## The Use Case: L1 IT Support Agent

**Context:** Banking company with AI & Data Platform supporting 1000+ users

## Current Situation

**500+**

Tickets/Month

**4 hours**

Avg Resolution Time

**15**

Support Categories

**24/7**

Coverage Needed

## Target Goals

- **70% automation rate** — Reduce human intervention
- **<30 min resolution** — Faster ticket handling
- **40% auto-resolution** — Agent executes fixes, not just suggests
- **99.9% accuracy** — Critical for banking compliance

## Ticket Types We Handle

### Data Pipeline Issues

- Pipeline failures (30% of tickets)
- Job timeout errors
- Schema validation failures
- Dependency conflicts

### Data Quality Issues

- Null values in critical fields
- Duplicate records
- Format mismatches
- Data freshness alerts

### Access & Permissions

- Database access requests
- Role-based permissions
- API key generation
- Service account setup

## Query Optimization

- Slow query analysis
- Index recommendations
- Resource usage issues
- Cost optimization

# The 6-Stage Evolution Timeline



1980s-2010s

### Stage 1: Rule-Based Systems

Decision trees and keyword matching. Rigid but predictable.



2010s

### Stage 2: Traditional ML

Classification models learned patterns from historical data.



2022+

### Stage 3: GenAI + Prompting

Large Language Models with few-shot learning brought flexibility.



2023+

### Stage 4: RAG (Retrieval-Augmented Generation)

Grounded generation with vector search eliminated hallucinations.



2023+

## Stage 5: Fine-Tuning

Custom models trained on domain data for consistency.



2024+

## Stage 6: Agentic AI Workflows

Multi-agent systems with tools that can plan and execute actions.



## Stage 1: Rule-Based Systems (1980s-2010s)

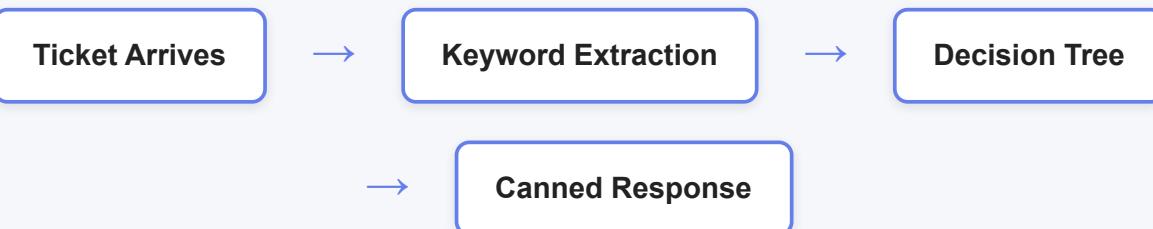
**Bottom Line:** If-then logic that matches keywords to predefined responses. Simple, predictable, but breaks easily with variations.

### How It Works for L1 Support

The system uses decision trees to match ticket text against known patterns:

```
def classify_ticket(ticket_text): # Simple keyword matching if "schema" in
    ticket_text.lower(): return { "category": "schema_validation", "solution": "Run
    schema sync job", "confidence": "high" } elif "timeout" in ticket_text.lower():
    return { "category": "pipeline_timeout", "solution": "Increase job timeout to
    2h", "confidence": "medium" } else: return { "category": "unknown", "solution":
    "Escalate to L2"}
```

### Architecture



## Performance Metrics

~20%

Tickets Handled

<100ms

Response Time

2 weeks

Development Time

\$0

Infrastructure Cost

### ✓ Strengths

- Extremely fast (<100ms)
- 100% predictable outputs
- No training data needed
- Easy to debug and explain
- Works offline
- Zero infrastructure costs

### ✗ Limitations

- Only handles ~20% of tickets
- Breaks with typos or variations
- Requires manual rule maintenance
- No learning from new patterns
- Cannot handle complex reasoning
- Rigid, no flexibility

## Why It Failed for Our Use Case

### Real Example:

- ✓ Matches: "Pipeline failed due to schema mismatch"
- ✗ Misses: "The job crashed because the schema was wrong"
- ✗ Misses: "Schema error in ETL process" (typo in "schema")

**Result:** Had to maintain 200+ rules, still only caught 20% of tickets.

## When to Still Use This Approach

- Ultra-low latency required (<50ms)
- Very limited, well-defined problem space (5-10 scenarios)
- 100% predictability is critical (safety systems, compliance)
- Offline operation needed

- Zero budget for ML infrastructure



## Stage 2: Traditional ML (2010s)

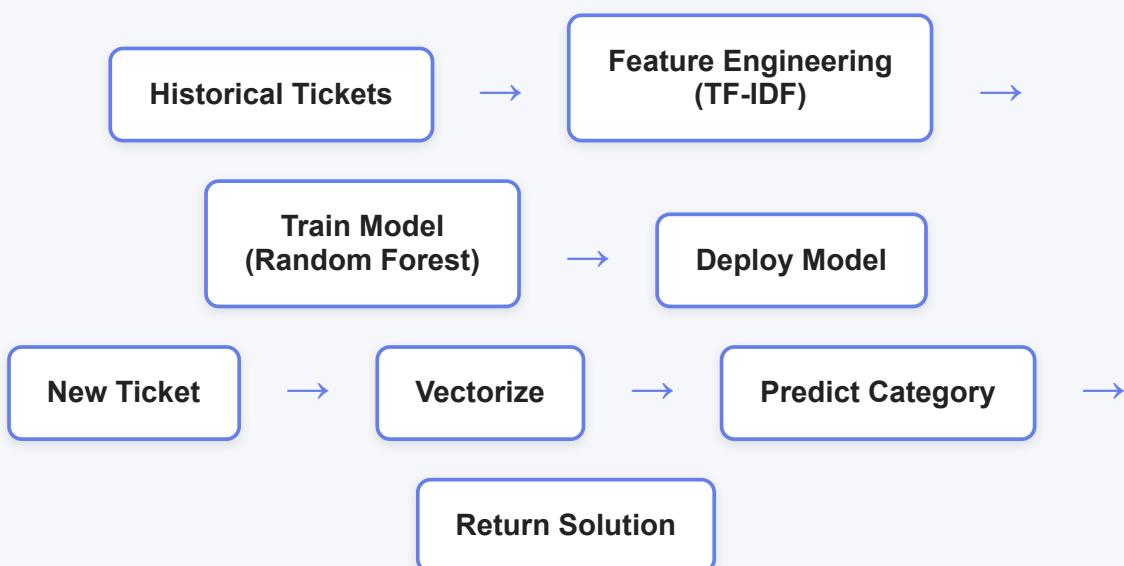
**Bottom Line:** Machine learning models trained on historical tickets to classify and predict. Better coverage than rules, but requires feature engineering and labeled data.

### How It Works for L1 Support

Train classification models on historical ticket data to predict categories and suggest solutions:

```
from sklearn.ensemble import RandomForestClassifier from
sklearn.feature_extraction.text import TfidfVectorizer # Feature engineering:
Convert text to numbers vectorizer = TfidfVectorizer(max_features=1000) X_train =
vectorizer.fit_transform(historical_tickets) y_train = ticket_categories # Labels: "schema_error", "timeout", etc. # Train classifier model =
RandomForestClassifier(n_estimators=100) model.fit(X_train, y_train) # Predict
for new ticket def classify_ticket_ml(new_ticket): X_new =
vectorizer.transform([new_ticket]) category = model.predict(X_new)[0] probability
= model.predict_proba(X_new).max() return category, probability
```

### Architecture



## Performance Metrics

~50%

Tickets Handled

~200ms

Response Time

6 weeks

Development Time

~\$500/mo

Infrastructure Cost

### ✓ Strengths

- Learns patterns from data
- Handles typos and variations better
- Scales to 50% ticket coverage
- Probabilistic confidence scores
- Can retrain as new data arrives
- Still relatively fast (~200ms)

### ✗ Limitations

- Requires 1000s of labeled examples
- Needs ML expertise (feature engineering)
- Cannot generate explanations
- Struggles with new ticket types
- Brittle to data distribution shifts
- Maintenance overhead (retraining)

## Why We Needed Better (Evolution Trigger)

### The Problems:

- **Cannot generate natural language explanations** — Users want "why" not just categories
- **Poor zero-shot performance** — New ticket types require retraining
- **Feature engineering bottleneck** — Domain experts needed to craft features
- **No reasoning ability** — Can't chain multiple steps or provide troubleshooting

**What we needed:** A system that could understand context, generate natural language, and handle novel situations without retraining.

## When to Still Use This Approach

- Well-defined classification problem with plenty of labeled data
- Low latency requirements (50-200ms)
- Interpretability through feature importance

- Budget constraints (cheaper than LLM APIs)
- Data stays within controlled environment (no external APIs)



## Stage 3: GenAI + Prompting (2022+)

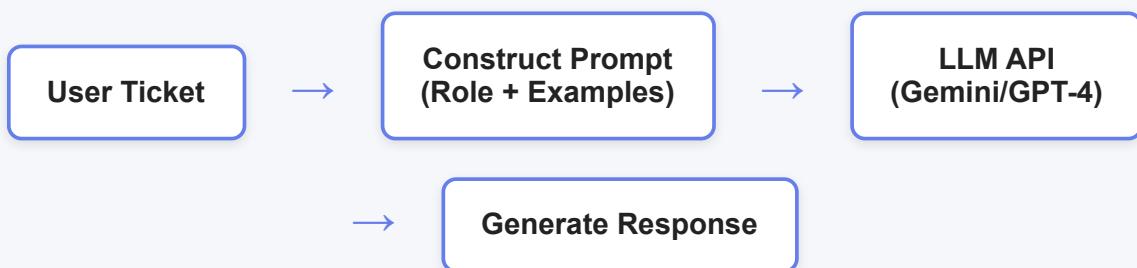
**Bottom Line:** Large Language Models with few-shot learning bring natural language understanding and generation. Flexible but can hallucinate without grounding.

### How It Works for L1 Support

Use LLMs (Gemini, GPT-4) with carefully crafted prompts and examples:

```
import google.generativeai as genai # Configure Gemini
genai.configure(api_key="YOUR_API_KEY") model = genai.GenerativeModel('gemini-1.5-pro') # Craft prompt with role and examples
prompt = """You are an L1 IT Support Agent for a banking data platform. EXAMPLES: Ticket: "Pipeline job-123 failed with schema mismatch error" Response: This is a schema validation issue. Solution: Run `schema_sync_job` for table ABC. Then restart pipeline job-123. Ticket: "Query timeout on dashboard_report" Response: Query exceeded 30s limit. Solution: Add indexes on date_column and user_id, or increase timeout to 60s in config. NEW TICKET: """ + user_ticket
response = model.generate_content(prompt)
print(response.text)
```

### Architecture



### Performance Metrics

**~50%**

Tickets Handled

**2-5 sec**

Response Time

**1-2 weeks**

Development Time

**\$2K-5K/mo**

API Costs (500 tix)

### ✓ Strengths

- Natural language understanding
- Zero-shot and few-shot learning
- Generates helpful explanations
- Handles novel situations
- Fast to prototype (days not weeks)
- No training data required

### ✗ Limitations

- Can hallucinate solutions
- No access to live data/runbooks
- High API costs at scale
- Inconsistent outputs
- Slower latency (2-5s)
- Compliance concerns (data to 3rd party)

## Why We Needed Better (Evolution Trigger)

### Critical Incident:

User: "How do I fix null values in customer\_transactions table?"

LLM Response: "Run UPDATE customer\_transactions SET amount = 0 WHERE amount IS NULL"

**Problem:** This could corrupt financial data! The LLM hallucinated a solution without checking company-specific runbooks that explicitly say "Never modify financial tables without approval."

**What we needed:** A way to ground LLM responses in our actual documentation and data.

## When to Still Use This Approach

- Rapid prototyping and POC validation
- Low-volume use cases (<1000 requests/day)
- Non-critical applications where occasional errors are acceptable
- When you don't have existing knowledge base to retrieve from

- Creative or open-ended tasks

## Stage 4: RAG - Retrieval-Augmented Generation (2023+)

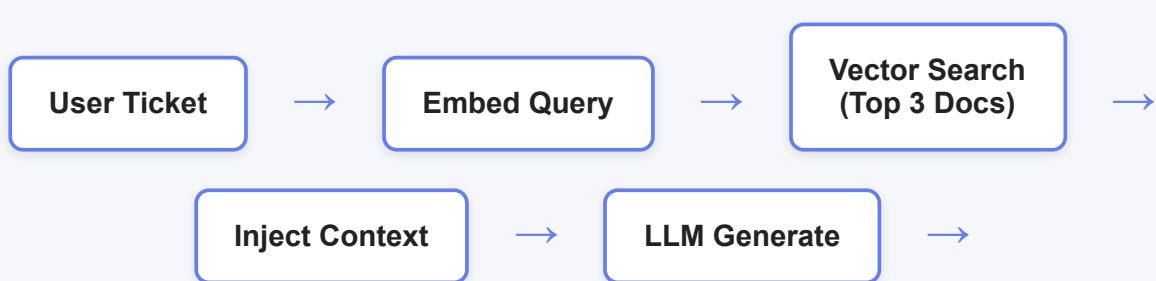
**Bottom Line:** Combine vector search with LLM generation to ground responses in real documentation. Eliminates hallucinations by providing source context.

### How It Works for L1 Support

Index runbooks, documentation, and past solutions in a vector database. Retrieve relevant context before generating responses:

```
import chromadb from sentence_transformers import SentenceTransformer # 1. Index
knowledge base client = chromadb.Client() collection =
client.create_collection("support_docs") # Add runbooks to vector DB embedder =
SentenceTransformer('all-MiniLM-L6-v2') for doc in runbooks: embedding =
embedder.encode(doc['text']) collection.add( embeddings=[embedding], documents=
[doc['text']], ids=[doc['id']] ) # 2. Retrieve relevant docs for new ticket def
answer_with_rag(ticket): # Search vector DB query_embedding =
embedder.encode(ticket) results = collection.query( query_embeddings=
[query_embedding], n_results=3 ) # Construct grounded prompt context =
"\n\n".join(results['documents'][0]) prompt = f"""Using ONLY the following
documentation, answer the ticket. DOCUMENTATION: {context} TICKET: {ticket}
ANSWER (with source references):""" response = model.generate_content(prompt)
return response.text, results['documents'][0] # Return with sources
```

### Architecture



## Performance Metrics

~70%

Tickets Handled

3-6 sec

Response Time

3-4 weeks

Development Time

\$3K-6K/mo

Total Costs

### ✓ Strengths

- Grounded in actual documentation
- Provides source citations
- Reduced hallucinations (~5% vs 30%)
- Knowledge stays up-to-date
- Explainable and auditable
- Handles 70% of tickets

### ✗ Limitations

- Quality depends on chunking strategy
- Cannot learn new behavior patterns
- Still some inconsistency in tone
- Additional latency from retrieval
- Requires vector DB infrastructure
- Doesn't execute actions (just suggests)

## Why We Needed Better (Evolution Trigger)

### Limitations in Production:

- **Inconsistent communication style** — Sometimes formal, sometimes casual, not aligned with bank's tone
- **Cannot learn company-specific patterns** — Every response starts fresh, no memory of what works
- **High cost at scale** — Processing 500 tickets/day with long context = \$5K+/month
- **Still suggests, doesn't execute** — Requires human to run the fix

**What we needed:** Options for (1) custom behavior/tone AND (2) autonomous execution.

## When to Still Use This Approach

- Best choice for most production Q&A systems
- Knowledge base changes frequently (daily/weekly updates)
- Need explainability and source citations
- Don't need custom tone/behavior (standard LLM voice is fine)
- Moderate volume (100-10K requests/day)
- Compliance requires data traceability

## Stage 5: Fine-Tuning (2023+)

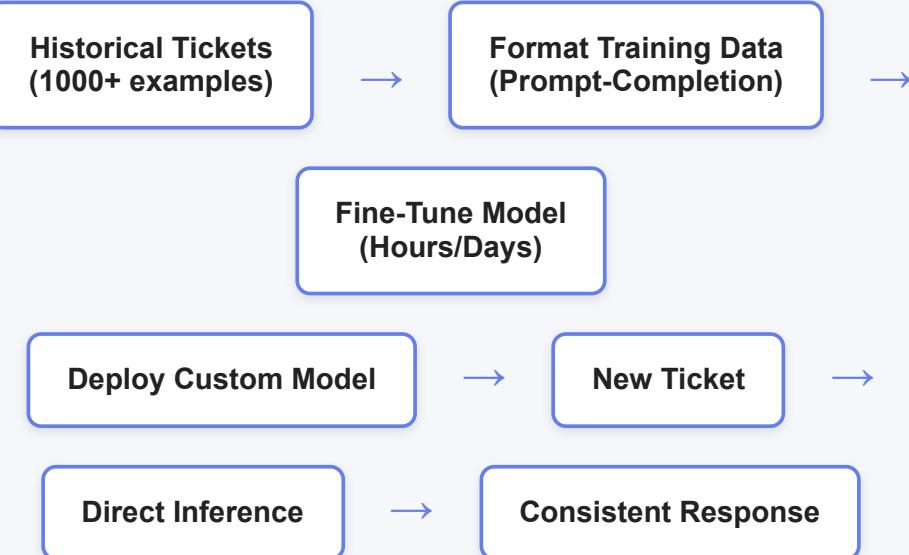
**Bottom Line:** Train a custom model on your domain data to learn consistent behavior, tone, and reasoning patterns. Best for high-volume, style-critical applications.

## How It Works for L1 Support

Create training dataset from resolved tickets and fine-tune Gemini to match your support team's behavior:

```
# 1. Prepare training data (JSONL format) training_data = [ { "messages": [ {"role": "user", "content": "Pipeline ABC failed: schema mismatch"}, {"role": "assistant", "content": "I've identified this as a schema validation issue. Per standard procedure SOP-451, please run the schema sync job for table customers_prod, then restart pipeline ABC via the control panel. ETA: 5 minutes. -Banking IT Support"} ] }, # ... 1000+ more examples ... ] # 2. Fine-tune via API from google.ai import generativelanguage client = generativelanguage.GenerativeServiceClient() operation = client.create_tuned_model( source_model="models/gemini-1.5-flash", training_data=training_data, hyperparameters={ "learning_rate": 0.001, "epochs": 3 } ) # 3. Use fine-tuned model tuned_model = genai.GenerativeModel('tunedModels/my-support-agent-xyz') response = tuned_model.generate_content(new_ticket)
```

## Architecture



## Performance Metrics

**~70%**

Tickets Handled

**1-2 sec**

Response Time

**6-8 weeks**

Development Time

**\$1K-2K/mo**

Inference Costs

### ✓ Strengths

- Consistent tone and formatting
- Learns company-specific patterns
- Lower cost per query at scale
- Faster inference (no retrieval)
- Can learn complex reasoning
- Better at structured outputs (JSON)

### ✗ Limitations

- Requires 1000+ quality examples
- Slow to update (retrain needed)
- High upfront training cost
- ML expertise needed
- Can bake in outdated info
- Still doesn't execute actions

## Combining RAG + Fine-Tuning (Best Practice)

### The Winning Strategy:

- **Fine-tune for:** Consistent tone, formatting, reasoning patterns, company terminology
- **RAG for:** Up-to-date factual information, policies, runbooks

**Example:** Fine-tuned model speaks like your team ("Per SOP-451..."), RAG ensures it cites current procedures (not outdated ones).

**Result:** Best of both worlds — consistency + freshness.

## When to Use This Approach

- High volume (10K+ requests/day) — cost savings matter
- Consistent tone/brand voice is critical
- Need structured outputs (JSON, XML) reliably
- Have 1000+ high-quality examples
- Domain-specific jargon and patterns
- Long-term deployment (ROI from training investment)

## Stage 6: Agentic AI Workflows (2024+)

**Bottom Line:** Multi-agent systems that can plan, use tools, and execute actions autonomously. The cutting edge — moves from "suggest solutions" to "fix problems."

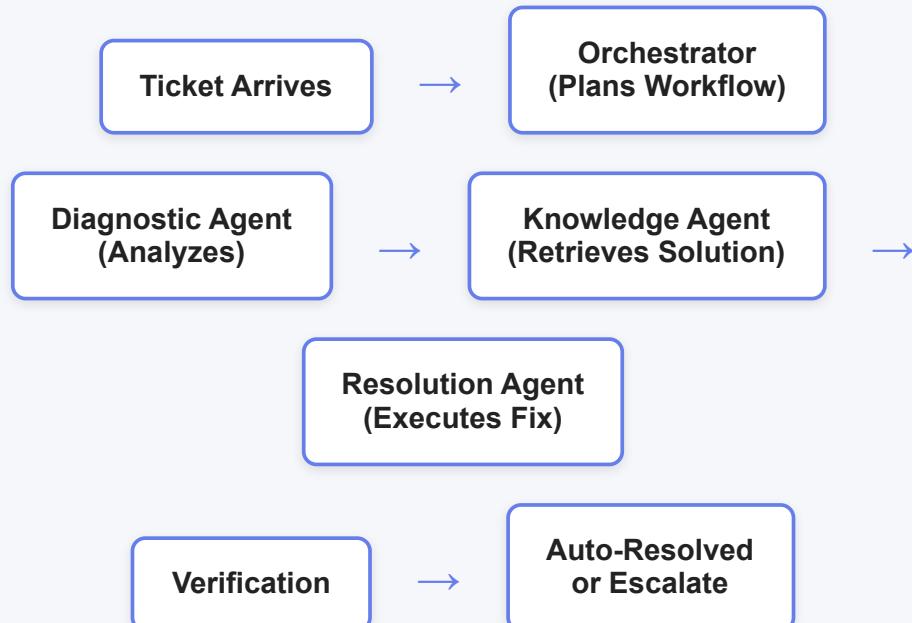
## How It Works for L1 Support

Orchestrate specialized agents with tool access using Google ADK (Agentic Development Kit):

```
from adk import Agent, Tool, Orchestrator # Define specialized agents
diagnostic_agent = Agent( name="DiagnosticAgent", role="Analyze ticket and
determine root cause", model="gemini-1.5-pro", tools=["query_logs",
"check_pipeline_status"] ) knowledge_agent = Agent( name="KnowledgeAgent",
role="Search runbooks for solutions", model="gemini-1.5-flash", tools=
["vector_search", "retrieve_docs"] ) resolution_agent = Agent(
name="ResolutionAgent", role="Execute fix and verify resolution", model="gemini-
1.5-pro", tools=["run_schema_sync", "restart_pipeline", "verify_job"] ) #
Orchestrate workflow orchestrator = Orchestrator( agents=[diagnostic_agent,
knowledge_agent, resolution_agent], workflow="sequential" # Can also be
```

```
"parallel" or "conditional" ) # Process ticket autonomously result =  
orchestrator.run(ticket=user_ticket)
```

## Architecture



## Performance Metrics

**70%**  
Tickets Handled

**40%**  
Auto-Resolved

**5-15 sec**  
End-to-End Time

**\$4K-8K/mo**  
Total Costs

## Key Capabilities

### Planning & Reasoning

Agents break down complex tickets into sub-tasks, determine execution order, and adapt based on results.

## Tool Use

Can call APIs, run scripts, query databases, restart services — not just suggest actions.

## Self-Correction

Verifies fix worked, retries with different approach if failed, escalates if unsure.

## Multi-Agent Collaboration

Specialized agents work together — diagnostic + knowledge + execution — better than one generalist.

### Strengths

- End-to-end automation (40%)
- Can execute actions, not just suggest
- Handles complex multi-step workflows
- Self-corrects and adapts
- Auditable decision trail
- Continuous improvement via feedback

### Limitations

- Complex to build and test
- Higher latency (multi-step)
- Requires robust safety guardrails
- Debugging can be challenging
- Cost increases with tool calls
- Still emerging (fewer best practices)

## Production Considerations for Banking

### Critical Requirements:

- **Safety Guardrails:** Agents can only execute pre-approved actions with defined parameters
- **Human-in-the-Loop:** Critical operations (data deletion, permission changes) require approval

- **Audit Logging:** Every agent decision and action logged for compliance
- **Rollback Capability:** All changes reversible via automated rollback procedures
- **Cost Monitoring:** Circuit breakers prevent runaway API costs from infinite loops

## When to Use This Approach

- You need **autonomous execution**, not just recommendations
- Complex workflows with multiple steps and decision points
- Have mature AI/ML team and infrastructure
- High-value use case justifies complexity (ROI > 3x)
- Can invest in safety, monitoring, and governance
- Want cutting-edge capabilities, accept more risk



## Complete Side-by-Side Comparison

Criteria	Rule-Based	Traditional ML	GenAI Prompting	RAG	Fine-Tuning	Agentic AI
Era	1980s-2010s	2010s	2022+	2023+	2023+	2024+
Ticket Coverage	~20%	~50%	~50%	~70%	~70%	~70% (40% auto-resolved)
Response Time	<100ms	~200ms	2-5s	3-6s	1-2s	5-15s
Dev Time	2 weeks	6 weeks	1-2 weeks	3-4 weeks	6-8 weeks	8-12 weeks
Monthly Cost (500 tickets)	\$0	~\$500	\$2K-5K	\$3K-6K	\$1K-2K (+ \$5K training)	\$4K-8K
ML Expertise	None	High	Low	Medium	High	Very High
Flexibility	Very Low	Low	Very High	Very High	Medium	Very High
Hallucination Risk	None	None	High (30%)	Low (5%)	Low (3%)	Medium (10%, in plans)

<b>Execution Capability</b>	No	No	No	No	No	<b>YES</b>
<b>Best For</b>	Ultra-simple, predictable	Stable classification	Rapid prototyping	Grounded Q&A	Consistent style at scale	Complex automation

## 🎓 Decision Framework: How to Choose

### Start Here

**Golden Rule:** Always start with the simplest approach that meets your requirements. Don't use agentic AI when rule-based would work. Complexity is a cost.

### Decision Tree

```

START: What's your primary need? ┌ Ultra-low latency (<100ms) + simple patterns
| └→ RULE-BASED SYSTEMS ┌ Classification with labeled data, no LLM needed
| └→ TRADITIONAL ML ┌ Natural language, rapid prototyping, low volume
| └→ GenAI PROMPTING ┌ Need grounded, factual responses with sources
| ┌ Knowledge changes frequently?
| | └→ RAG ┌ Need consistent tone/style at high volume?
| └→ RAG + FINE-TUNING ┌ Require autonomous execution of actions
| ┌ Complex multi-step workflows?
| └→ AGENTIC AI ┌ HYBRID: Different approaches for different ticket types
| ┌ Example: Rules for simple (20%) + RAG for complex (50%) + Agents for execution (30%)
  
```

### Common Patterns in Production

#### Pattern 1: Progressive Fallback

**Try:** Rule-based → ML classifier → RAG → Human

**Why:** Fast and cheap first, expensive only when needed

#### Pattern 2: Hybrid Pipeline

**Use:** Rules for 20% + RAG for 50% + Agents for 30%

**Why:** Right tool for each ticket complexity

### Pattern 3: RAG + Fine-Tuning

**Combine:** Fine-tuned for tone, RAG for facts

**Why:** Consistency + freshness

### Pattern 4: Agent + Human-in-Loop

**Use:** Agent proposes, human approves critical actions

**Why:** Automation with safety



## 4-Week Learning Plan

### Structured Path to Mastery

#### Week 1: Foundations (Rule-Based + Traditional ML)

**Goal:** Understand why simple approaches exist and when they work

- **Day 1-2:** Read Stage 1-2 sections, run rule\_based\_agent.py
- **Day 3-4:** Build your own classifier (sklearn) on sample tickets
- **Day 5:** Compare rule-based vs ML: precision, recall, coverage

**Deliverable:** Working ML classifier + analysis doc

#### Week 2: GenAI Era (Prompting + RAG)

**Goal:** Master LLM-based approaches

- **Day 1-2:** Experiment with Gemini prompting (few-shot learning)
- **Day 3-4:** Build RAG system with ChromaDB + Gemini

- **Day 5:** Read RAG guide (04-rag.html), implement chunking strategies

**Deliverable:** Working RAG prototype with your team's runbooks

## Week 3: Advanced Techniques (Fine-Tuning)

**Goal:** Learn when and how to customize models

- **Day 1-2:** Prepare fine-tuning dataset (100+ examples)
- **Day 3-4:** Fine-tune Gemini on support tickets
- **Day 5:** A/B test: base model vs fine-tuned vs RAG

**Deliverable:** Fine-tuned model + comparison metrics

## Week 4: Cutting Edge (Agentic Workflows)

**Goal:** Understand multi-agent systems

- **Day 1-2:** Read Agentic guide (06-agentic-workflows.html) thoroughly
- **Day 3-4:** Build multi-agent system with ADK
- **Day 5:** Team presentation: decision framework for your use cases

**Deliverable:** Working agent + production readiness checklist

**Post-Training:** Each team member should be able to:

- ✓ Explain trade-offs between all 6 approaches
- ✓ Design an AI system for a new use case
- ✓ Estimate costs, timelines, and technical requirements
- ✓ Implement at least one working prototype

## Additional Resources

### Detailed Guides

- [Rule-Based Systems Deep Dive](#)
- [Traditional ML Guide](#)
- [GenAI Prompting Guide](#)
- [RAG Implementation Guide](#)
- [Fine-Tuning Guide](#)

- [Agentic Workflows Guide](#) ★

## Working Code Examples

- [Rule-Based Agent](#) (Python)
- [ML Classifier](#) (sklearn)
- [Prompting Examples](#) (Gemini)
- [RAG System](#) (ChromaDB)
- [Multi-Agent System](#) (ADK)

## Quick References

- [Quick Start Guide](#)
- [Overview & Setup](#)
- [Decision Tree \(Printable\)](#)
- [ROI Calculator](#)

## External Links

- [Google ADK Documentation](#)
- [Gemini API Docs](#)
- [LangChain](#)
- [ChromaDB](#)

Ready to dive deeper?

[Quick Start Guide](#)

[Agentic Workflows \(Priority\)](#)

[Run Code Examples](#)