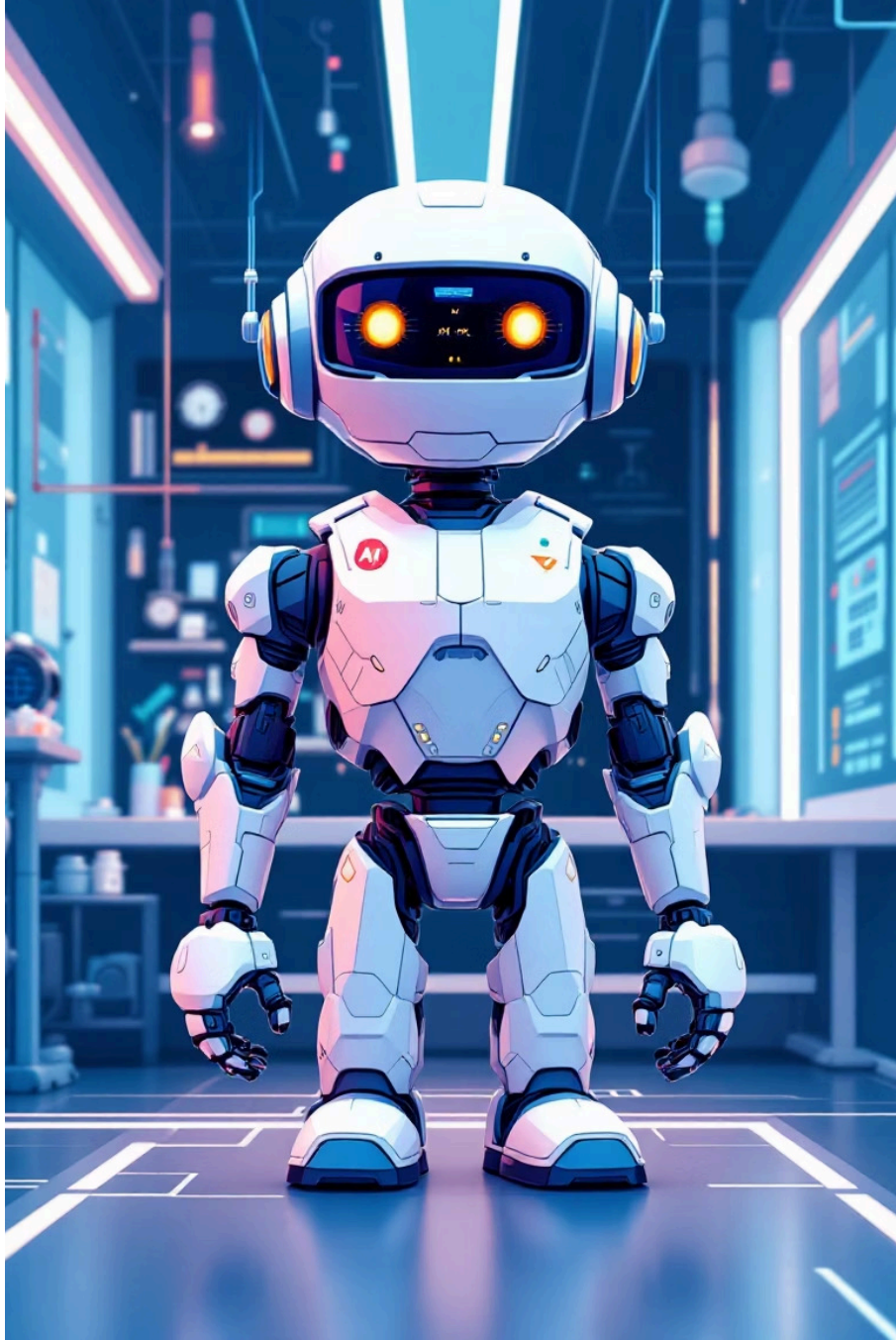
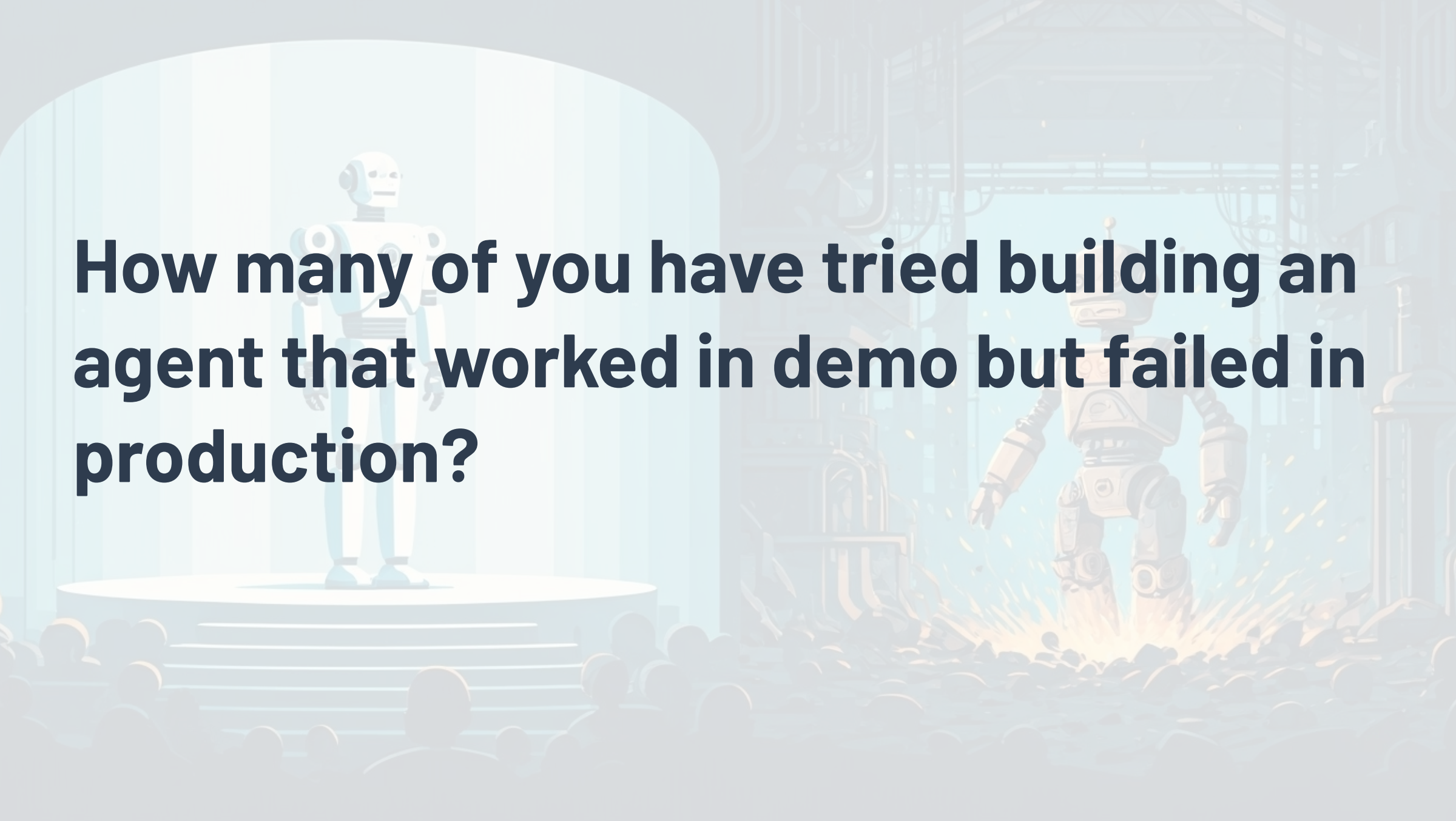
Designing and Building AI Agents: From Principles to Production

by: Tebian Abdelbagi

20 September 2025





How many of you have tried building an agent that worked in demo but failed in production?

The Hook

Infinite Loops & Unpredictable Actions

They get stuck in endless cycles or take erratic actions that break your systems.

Catastrophic Hallucinations

LLM hallucinations lead to illogical decisions that can damage business operations.

Context Loss in Complex Workflows

They lose state and context, becoming useless when handling sophisticated processes.

Demo-to-Production Gap

The chasm between a cool demo and a robust, reliable production system is vast.

🚀 Today, we'll bridge that gap.

Today's Journey

01

🧠 Core Concepts: What is an Agent?

02

🏗️ LangChain Architecture: The Blueprint

03

📊 The Agent Spectrum: Principles, Pitfalls & Production Tips

04

⚙️ Production Best Practices: Making it safe, monitored, and reliable

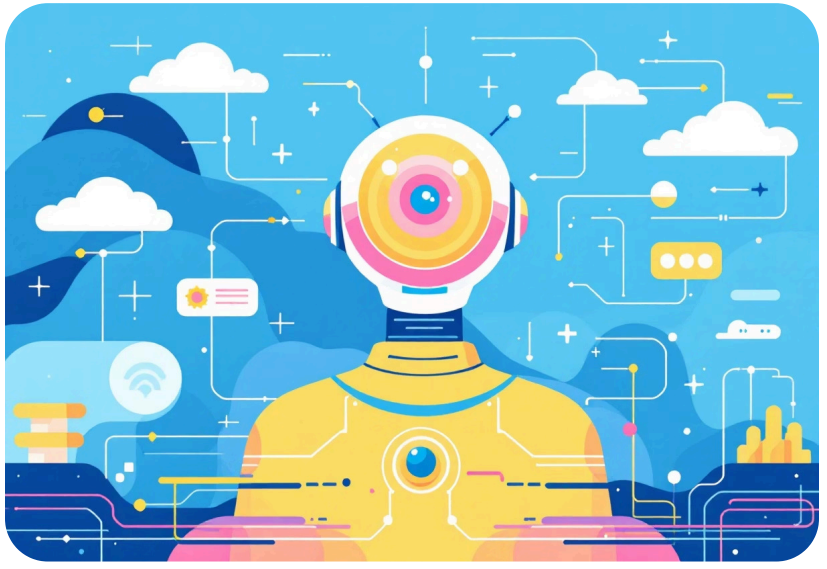
05

🚀 Case Study: Building a Data Pipeline QA Agent

AI Agent Architecture with LangChain

What is AI Agent?

An autonomous system that perceives its environment, reasons about it, and takes actions to achieve specific goals.



The LangChain Blueprint

- **Perception:** LangChain Connectors & Retrievers
- **Reasoning:** LangChain Chains & Agents (ReAct, Plan-and-Execute)
- **Memory:** ConversationBufferMemory, VectorStoreRetrieverMemory
- **Action:** Tools & APIs for real-world interaction

Single-Agent Systems

Independent LangChain agents handling specific workflows autonomously.

Multi-Agent Systems

Orchestrated LangChain agents collaborating on complex, distributed tasks.

Key Distinction: Bots are scripted · Assistants are reactive · LangChain Agents are autonomous with memory, tools, and planning capabilities.

What Makes an AI Agent Effective?

Successful AI agents combine six fundamental principles that enable them to operate reliably in real-world environments whilst maintaining user trust.

Autonomy

Operates independently with minimal human intervention. Robot vacuums map and clean rooms autonomously, adapting to furniture layouts.

Goal-Oriented Behaviour

Each action drives toward defined objectives. GPS systems suggest optimal routes to reach destinations efficiently, considering traffic patterns.

Adaptability & Learning

Improves performance through experience. Security systems learn to differentiate between pets and potential intruders over time.

Transparency & Trust

Actions remain explainable and predictable. Medical AI systems provide clear reasoning behind diagnostic recommendations.

Robustness & Error Handling

Recovers gracefully from unexpected issues. Chatbots escalate unknown queries to human agents rather than providing incorrect responses.

Simplicity & Modularity

Complex tasks broken into manageable components. Language models separate summarisation from response generation for better maintainability.

From Simple Reactions to Self-Learning Systems

AI agents exist across a spectrum of complexity, each suited to different challenges and operational requirements. Understanding this taxonomy helps choose the right architecture for your use case.

Simple Reflex Agents

Immediate reactions without memory. Smoke detectors trigger alarms when sensors detect particles.

Model-Based Reflex Agents

Uses memory for informed decisions. Smart lawn sprinklers consider weather history and soil conditions.

Goal-Based Agents

Plans actions to achieve objectives. Warehouse robots navigate efficiently to deliver items.

Utility-Based Agents

Evaluates trade-offs for optimal outcomes. Smart thermostats balance comfort and energy efficiency.

Learning Agents

Continuously improves from experience. Recommendation systems adapt to user preferences over time.

The Building Blocks of Safe Agent Design

Agent Type	Core Principle	Key LangChain Components
Simple Reflex	Stateless, deterministic IF-THEN operation	LLMChain, PromptTemplate
Model-Based Reflex	Maintains short-term context and state to disambiguate intent	ConversationBufferMemory, VectorStoreRetrieverMemory
Goal-Based	Plans and decomposes explicit objectives into sub-tasks	PlanAndExecuteAgentExecutor, ReAct Agent Types
Utility-Based	Uses a scoring function to choose the optimal action from many	Custom Agents, Toolkits, RunnableLambda for scoring
Learning	Adapts its policy over time based on feedback and new data	LangSmith (for feedback loops), Fine-tuning, RLHF

Real-World Analogy: Reflex (Thermostat) → Model-Based (GPS with traffic) → Goal-Based (Travel Agent) → Utility-Based (Stock Trading Bot) → Learning (Self-Driving Car)

The Traps That Break Your Agents

Agent Type	Common Pitfalls	Watch Out For...
Simple Reflex	• Brittle to unseen inputs • Repetitive/contradictory responses • Zero context awareness	Fragility. It will break on anything not explicitly in its rules.
Model-Based Reflex	• Memory drift (state becomes corrupted) • Inconsistent world state	Letting the LLM update memory directly without a strict schema.
Goal-Based	• Infinite loops from ambiguous goals • Hallucinations leading to incorrect tool use	Goals that are not clear, measurable, and achievable.
Utility-Based	• Reward hacking (gaming the utility function) • Computationally expensive	Overly complex utility functions that are hard to debug.
Learning	• Catastrophic forgetting (new learning overwrites old) • Privacy violations from feedback data	Deploying new policies without safe canary testing.

The Golden Rule: The more autonomous the agent, the more robust your safeguards must be.

How to Stay Out of the Traps

Agent Type	Production Implementation Tips
Simple Reflex	• Use fixed prompt templates in LLMChain. • Deploy behind a rate-limited API gateway. • Implement exhaustive logging.
Model-Based Reflex	• Store memory in Pinecone/Chroma with metadata. • Use VectorStoreRetrieverMemory for persistence. • Take periodic memory snapshots to S3/DB.
Goal-Based	• Enforce structured output (e.g., JSON schema). • Validate all tool inputs and outputs. • Implement closed-loop re-planning after each action.
Utility-Based	• Keep utility functions simple and interpretable. • Log all scores for debugging and optimization. • Use bandit algorithms for large action spaces.
Learning	• Isolate training from inference environments. • Use LangSmith to trace and manage feedback loops. • Deploy new policies in shadow/canary mode.

Universal Best Practice: Use LangSmith. Trace every call, monitor costs and latency, and debug based on real traces.

The Pillars of Production Resilience

Building agents that don't break requires more than code; it requires a mindset shift. These three principles are non-negotiable.

Predictability Over Brilliance

A predictable, simple agent is always better than a brilliant, unstable one.

- Guardrails, prompt templates, and limited tools ensure deterministic behavior.

Observability Above All

You cannot debug what you cannot see. Every decision must be traceable.

- This is why LangSmith is your most critical production dependency.

Embrace Iterative Evolution

Agents are not built; they are grown. Start simple and add complexity slowly.

- Feedback loops, canary deployments, and a culture of experimentation.

Your number one job is to reduce surprise.

🎯 Case Study: Data Pipeline QA Agent

Goal

Ensure end-to-end quality of ETL/data pipelines by automatically detecting, reporting, and correcting data issues.

The Problem

Data quality is reactive. Engineers fight fires instead of preventing them.

80%

Time Saved

Reduction in manual QA tasks

24/7

Monitoring

Continuous pipeline surveillance

1. Agent Design (Conceptual Layer)

We break down the agent's design using the Perception → Memory → Planning → Action cycle:

Perception

Input: Pipeline logs, Spark job metrics, database validation queries, schema mismatches.

Example: Monitor Spark stages for shuffle read errors or detect nulls in primary key columns.

Memory

Short-Term: Current pipeline execution context (errors in latest batch).

Long-Term: Historical validation results in Pinecone for trend analysis.

Example: Recall that customer_id frequently fails validation → run stricter checks.

Planning

Decides what QA rules to apply dynamically based on detected patterns.

Example: If schema drift detected → run schema-matching checks; if data skew detected → run distribution checks.

Action

- Raise alerts in Slack/Teams
- Auto-correct minor issues
- Generate QA reports for governance

Feedback Loop → LangSmith → Improves Agent

2. LangChain Implementing the Spectrum:

We can map the agent types into stages of increasing intelligence:

Reflex Agent

Immediately flags failed records (schema mismatch, nulls in key columns). Fast, rule-based, no history needed.

Model-Based Agent

Adds context by recalling past failures. "This error usually happens when source system X has missing lookups."

Goal-Based Agent

Orchestrates entire validation workflows. "If pipeline stage fails → retry → validate → escalate if still failing."

Utility-Based Agent

Prioritises checks by business impact. "Check revenue-related tables before less critical logs."

Learning Agent

Adapts QA rules from feedback. If analyst repeatedly ignores an alert, reduce its weight.

3. Production Pipeline (Step by Step)

01

Data Sources & ETL

Spark, Airflow, and database connectors feeding the pipeline.

02

Agent Perception Layer

Ingest logs, metrics, and row samples for analysis.

03

Memory Integration

Pinecone/Weaviate stores historical issues. PostgreSQL for structured QA metadata.

04

Planning Layer

Chain of thought: Detect anomaly → classify → pick best validation check. Tools: SQL executor, Spark validator, Slack notifier.

05

Action Layer

Auto-correct minor issues, generate daily QA reports (PDF/HTML), alert data engineers.

06

Monitoring & Feedback

Feedback stored in vector DB. Agent retrains and improves prioritisation logic.

4. Pitfalls & Mitigations

1

Infinite Loops

Risk: Agent gets stuck in retry cycles

Mitigation: Cap retries, add circuit breakers with exponential backoff

2

Hallucination / False Alerts

Risk: LLM generates incorrect validation rules

Mitigation: Use strict schema rules + deterministic checks for critical paths

3

Environment Drift

Risk: Training data doesn't match production reality

Mitigation: Separate training (historical QA) vs. inference (production pipeline) environments

4

Scalability Issues

Risk: LLM bottlenecks on large datasets

Mitigation: Use Spark for large-scale checks, not the LLM directly

5. Example Workflow

Data Ingestion

Spark job ingests CDRs → pipeline runs normally

Issue Detection

Agent detects nulls in customer_id column during validation

Context Recall

Memory recalls this happened in past 3 months during vendor outage

Intelligent Planning

Planner decides: run extra lookup validation + notify vendor team

Automated Action

Auto-impute with backup table, raise Slack alert, log correction

Feedback Loop

Feedback stored → system adapts (reduce false alarms next time)

👉 This makes the Data Pipeline QA Agent not just a "validator" but an autonomous QA co-pilot for the entire data engineering team.

Key Takeaways

1

Start Simple

Choose the simplest agent architecture that gets the job done effectively.

2

Architecture is Key

Clearly separate Perception, Memory, Planning, and Action for maintainable systems.

3

Production ≠ Demo

Build in guardrails, monitoring (LangSmith!), and feedback loops from day one.

4

LangChain Advantage

LangChain provides all the abstractions you need to build production-ready agent systems.

Thank You

Questions & Discussion

Connect with me:

Tebian Abdelbagi

Big data specalist

Resources:

- LangChain Documentation
- LangSmith Platform
- Production Agent Examples