

Open in app ↗

Medium

 Search

The perfect gift for readers and writers.

[Give the gift of Medium](#)

★ Member-only story

# A Comprehensive Reference Guide to Building Agentic AI Systems



Gaurav Nigam · Following

Published in aingineer

10 min read · 2 days ago



Listen



Share



More

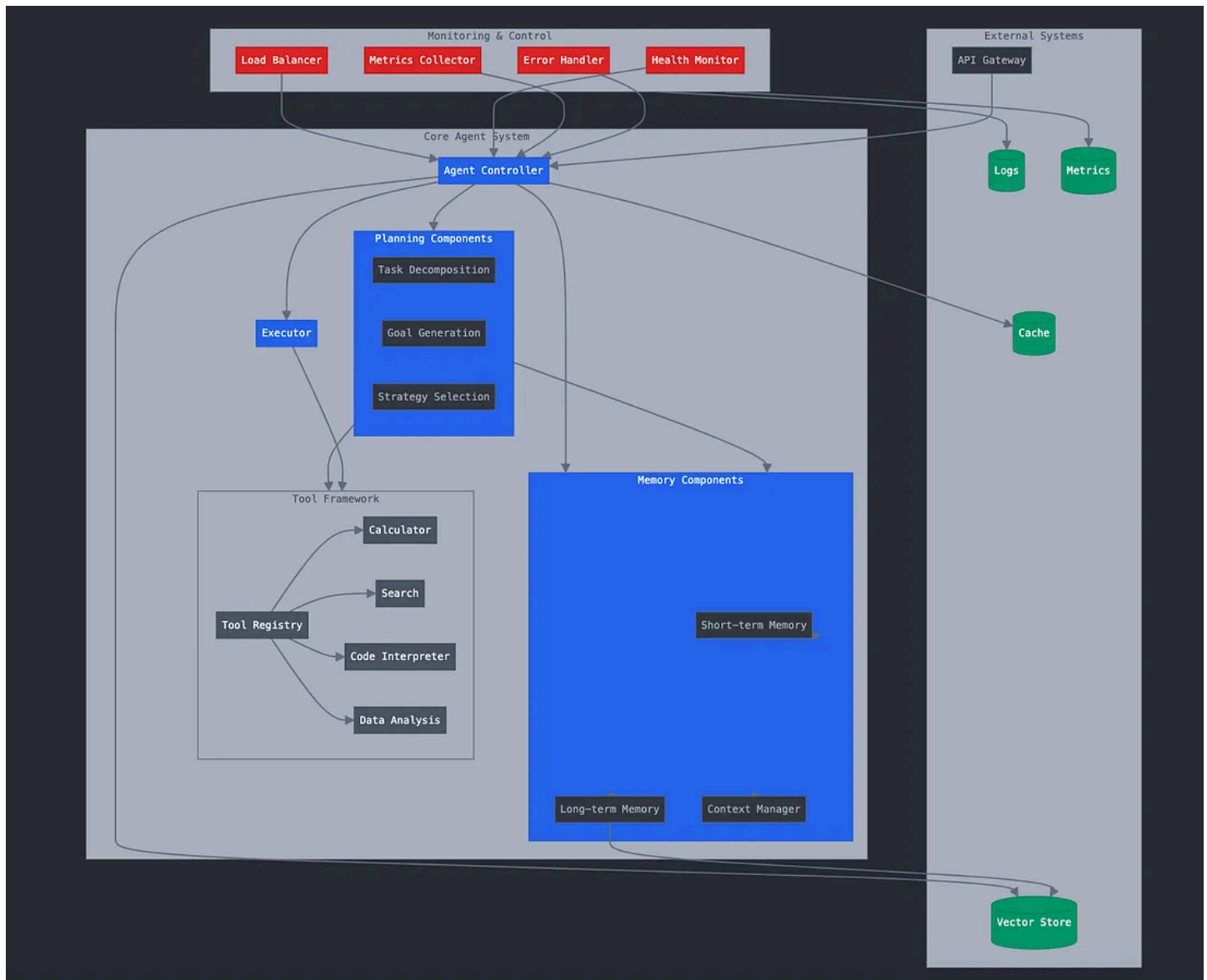
As artificial intelligence (AI) continues to evolve, the focus is shifting from passive models to **agentic AI** — systems capable of autonomous decision-making, planning, and execution. These systems leverage a combination of memory, tools, and sophisticated planning mechanisms to solve complex problems dynamically. For senior engineers and architects venturing into AI agent development, understanding the foundational components of agentic AI is essential.

## Why Agentic AI?

Traditional AI models, while powerful, often operate within predefined boundaries, generating outputs reactively based on input prompts. Agentic AI, however, moves beyond this by embedding planning, reflection, and goal-oriented action. This paradigm opens the door to building AI that can:

- Break down complex tasks into subgoals.
- Adapt and iterate based on feedback loops.
- Utilize external tools (e.g., calculators, code interpreters) to enhance functionality.
- Retain and apply knowledge through short-term and long-term memory.

# Architecture Component of Agentic System



Core Component Architecture

## Core Components of Agentic Systems

The architecture of agentic AI revolves around several key components, as illustrated in IBM's conceptual diagram:

### 1. Memory

Memory forms the backbone of agentic AI, enabling systems to recall past interactions and experiences. This memory is categorized into:

- **Short-term memory:** Facilitates immediate context retention.
- **Long-term memory:** Stores persistent knowledge for extended problem-solving capabilities.

Memory not only enhances continuity across sessions but also allows agents to learn and evolve over time.

## 2. Tools

Agentic AI leverages external tools to extend its capabilities. Common tools include:

- **Calculators** for numerical tasks.
- **Code interpreters** for executing and debugging scripts.
- **Search engines** to fetch real-time information.
- **Calendars** to manage scheduling tasks.

With tools, agents can perform actions beyond their internal logic, increasing flexibility and accuracy.

## 3. Planning

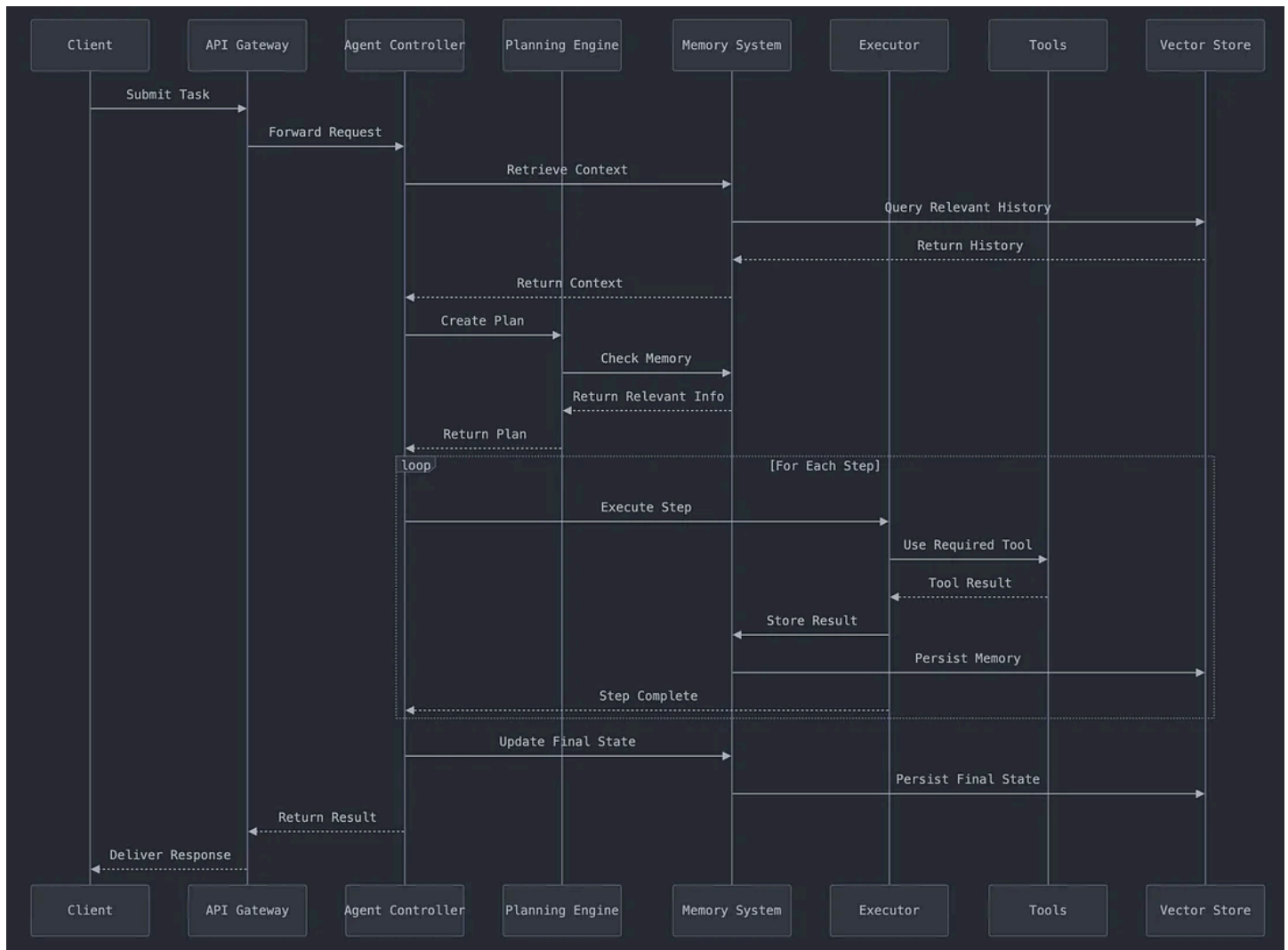
Planning enables agents to outline tasks, set goals, and break problems into manageable steps. Planning in agentic AI often incorporates:

- **Reflection:** Reviewing past decisions to refine approaches.
- **Self-criticism:** Allowing the agent to assess and correct errors.
- **Chain of thoughts:** Building a sequential thought process for complex reasoning.
- **Subgoal decomposition:** Dividing overarching goals into smaller, actionable steps.

## 4. Action

Once planning is complete, the agent proceeds to execute tasks, interacting with tools and refining its actions as necessary. The action phase may involve iterative loops, where the agent reviews outcomes and re-engages in planning.

## Execution Flow



Execution Flow

## Promising Architectures for Agentic AI

Several architectures stand out as promising frameworks for building agentic AI:

1. **ReAct (Reasoning + Acting) Framework:** This framework synergizes reasoning and acting steps, allowing agents to iteratively think and execute actions. It is particularly effective for tasks requiring multi-step problem-solving and tool use.
2. **LangGraph (Evolution of LangChain):** LangGraph facilitates the development of AI agents by structuring workflows into directed graphs, allowing branching decisions and complex task orchestration. It builds upon LangChain's foundation but introduces enhanced capabilities for agentic workflows.
3. **AutoGPT/AgentGPT Architectures:** These architectures focus on recursive task decomposition and automation. AutoGPT agents can autonomously set goals,

decompose tasks, and execute them iteratively.

## Implementation Details

### 1. Memory Systems

Memory in agentic AI requires careful implementation across multiple layers:

#### Short-term Memory (Working Memory)

```
from langchain.memory import ConversationBufferMemory, ConversationBufferWindowMemory

# For recent context retention
short_term_memory = ConversationBufferWindowMemory(
    k=5, # Keep last 5 interactions
    return_messages=True,
    memory_key="chat_history",
    input_key="input",
    output_key="output"
)
```

#### Long-term Memory (Persistent Storage)

```
from langchain.vectorstores import Chroma
from langchain.embeddings import OpenAIEmbeddings

# Initialize vector store
embeddings = OpenAIEmbeddings()
vectorstore = Chroma(
    collection_name="agent_memory",
    embedding_function=embeddings,
    persist_directory="./memory"
)

# Store and retrieve memories
def store_memory(text: str, metadata: dict):
    vectorstore.add_texts(
        texts=[text],
        metadatas=[metadata]
    )

def retrieve_relevant_memories(query: str, k: int = 3):
    return vectorstore.similarity_search(query, k=k)
```

## Memory Management Best Practices

- Implement periodic memory consolidation
- Use relevance scoring for memory retrieval
- Handle context window limitations
- Implement memory cleanup for outdated information

## 2. Tool Integration Framework

Tools should be implemented with proper error handling and validation:

```
from typing import Protocol, Dict, Any
from pydantic import BaseModel, Field

class ToolInterface(Protocol):
    def execute(self, inputs: Dict[str, Any]) -> Dict[str, Any]:
        """Execute tool functionality"""
        pass

class CalculatorTool(BaseModel):
    name: str = Field(default="calculator")
    description: str = Field(default="Performs mathematical calculations")

    def execute(self, inputs: Dict[str, Any]) -> Dict[str, Any]:
        try:
            expression = inputs.get("expression")
            if not expression:
                raise ValueError("No expression provided")

            # Safe eval implementation
            import ast
            import operator

            operators = {
                ast.Add: operator.add,
                ast.Sub: operator.sub,
                ast.Mult: operator.mul,
                ast.Div: operator.truediv
            }

            def eval_expr(node):
                if isinstance(node, ast.Num):
                    return node.n
                elif isinstance(node, ast.BinOp):
                    op = operators.get(type(node.op))
                    if not op:
                        raise ValueError("Unsupported operation")
                    return op(eval_expr(node.left), eval_expr(node.right))
```

```

        else:
            raise ValueError("Invalid expression")

        tree = ast.parse(expression, mode='eval')
        result = eval_expr(tree.body)

        return {"result": result}

    except Exception as e:
        return {"error": str(e)}

```

### 3. Planning and Execution Framework

Implementing a robust planning system using the ReAct framework:

```

from typing import List, Optional
from pydantic import BaseModel

class Action(BaseModel):
    tool: str
    input: Dict[str, Any]
    thought: str

class Plan(BaseModel):
    goal: str
    steps: List[Action]
    current_step: int = 0
    status: str = "pending"

class PlanningAgent:
    def __init__(self, tools: Dict[str, ToolInterface], llm):
        self.tools = tools
        self.llm = llm
        self.memory = ConversationBufferMemory()

    def create_plan(self, goal: str) -> Plan:
        # Generate plan using LLM
        prompt = self._create_planning_prompt(goal)
        response = self.llm.predict(prompt)

        # Parse response into structured plan
        actions = self._parse_actions(response)
        return Plan(goal=goal, steps=actions)

    def execute_plan(self, plan: Plan) -> Dict[str, Any]:
        results = []

        for step in plan.steps:
            try:

```

```

        # Execute tool
        tool = self.tools.get(step.tool)
        if not tool:
            raise ValueError(f"Tool {step.tool} not found")

        result = tool.execute(step.input)

        # Store result in memory
        self.memory.save_context(
            {"input": step.input},
            {"output": result}
        )

        results.append(result)

    except Exception as e:
        return {
            "error": str(e),
            "step": step.dict(),
            "results_so_far": results
        }

    return {"results": results}

```

## Real-world Implementation Considerations

### 1. Error Handling and Reliability

Implement comprehensive error handling:

```

class AgentError(Exception):
    """Base exception for agent-related errors"""
    pass

class ToolExecutionError(AgentError):
    """Raised when a tool execution fails"""
    pass

class PlanningError(AgentError):
    """Raised when plan creation or execution fails"""
    pass

def safe_execute_tool(tool: ToolInterface, inputs: Dict[str, Any]) -> Dict[str, Any]:
    try:
        return tool.execute(inputs)
    except Exception as e:
        raise ToolExecutionError(f"Tool execution failed: {str(e)}")

```



## 2. Performance Optimization

Implement caching and optimization strategies:

```
from functools import lru_cache
from typing import Optional
import time

class CachedTool:
    def __init__(self, tool: ToolInterface, cache_ttl: int = 3600):
        self.tool = tool
        self.cache_ttl = cache_ttl
        self._cache = {}

    @lru_cache(maxsize=1000)
    def execute(self, inputs: Dict[str, Any]) -> Dict[str, Any]:
        cache_key = str(inputs)
        current_time = time.time()

        # Check cache
        if cache_key in self._cache:
            result, timestamp = self._cache[cache_key]
            if current_time - timestamp < self.cache_ttl:
                return result

        # Execute and cache
        result = self.tool.execute(inputs)
        self._cache[cache_key] = (result, current_time)
        return result
```

## 3. Scaling Considerations

Implement rate limiting and concurrency control:

```
import asyncio
from typing import List
from async_timeout import timeout

class ScalableAgent:
    def __init__(self, max_concurrent: int = 5, timeout_seconds: int = 30):
        self.semaphore = asyncio.Semaphore(max_concurrent)
        self.timeout_seconds = timeout_seconds

    async def execute_concurrent_actions(self, actions: List[Action]) -> List[Dict[str, Any]]:
        async def execute_with_timeout(action: Action) -> Dict[str, Any]:
            async with self.semaphore:
                async with timeout(self.timeout_seconds):
                    return await action.execute()

        return await asyncio.gather(*[execute_with_timeout(action) for action in actions])
```

```

        return await self._execute_action(action)

    return await asyncio.gather(
        *[execute_with_timeout(action) for action in actions],
        return_exceptions=True
    )

```

## 4. Monitoring and Observability

Implement comprehensive logging and monitoring:

```

import logging
from datetime import datetime
from typing import Optional

class AgentMonitor:
    def __init__(self):
        self.logger = logging.getLogger("agent_monitor")
        self.metrics = {}

    def log_execution(
        self,
        action_type: str,
        start_time: datetime,
        end_time: datetime,
        status: str,
        error: Optional[Exception] = None
    ):
        duration = (end_time - start_time).total_seconds()

        self.logger.info({
            "action_type": action_type,
            "duration": duration,
            "status": status,
            "error": str(error) if error else None,
            "timestamp": datetime.utcnow().isoformat()
        })

        # Update metrics
        self.metrics[action_type] = {
            "count": self.metrics.get(action_type, {}).get("count", 0) + 1,
            "avg_duration": (
                self.metrics.get(action_type, {}).get("avg_duration", 0) *
                self.metrics.get(action_type, {}).get("count", 0) +
                duration
            )
        }

```

```
    ) / (self.metrics.get(action_type, {}).get("count", 0) + 1)
}
```

## Integration Examples

To bring this to life, let's consider building an AI-driven research assistant using LangGraph, leveraging the ReAct framework.

### 1. Building a Research Assistant Agent

```
class ResearchAssistant:
    def __init__(self):
        self.tools = {
            "search": SearchTool(),
            "summarize": SummarizationTool(),
            "extract": DataExtractionTool()
        }
        self.planner = PlanningAgent(self.tools, llm)
        self.monitor = AgentMonitor()

    async def research_topic(self, topic: str) -> Dict[str, Any]:
        try:
            # Create research plan
            plan = self.planner.create_plan(f"Research {topic}")

            # Execute plan
            start_time = datetime.utcnow()
            results = await self.planner.execute_plan(plan)
            end_time = datetime.utcnow()

            # Monitor execution
            self.monitor.log_execution(
                action_type="research",
                start_time=start_time,
                end_time=end_time,
                status="success"
            )

            return results

        except Exception as e:
            self.monitor.log_execution(
                action_type="research",
                start_time=start_time,
                end_time=datetime.utcnow(),
                status="error",
                error=e
            )
```

```
)  
raise
```

## Cost and Resource Considerations

### 1. Token Usage Optimization

```
class TokenManager:  
    def __init__(self, max_tokens: int = 4096):  
        self.max_tokens = max_tokens  
        self.current_tokens = 0  
  
    def estimate_tokens(self, text: str) -> int:  
        # Rough estimation: 4 chars = 1 token  
        return len(text) // 4  
  
    def can_add_text(self, text: str) -> bool:  
        estimated_tokens = self.estimate_tokens(text)  
        return (self.current_tokens + estimated_tokens) <= self.max_tokens  
  
    def add_text(self, text: str) -> bool:  
        if self.can_add_text(text):  
            self.current_tokens += self.estimate_tokens(text)  
            return True  
        return False
```

### 2. Resource Monitoring

```
class ResourceMonitor:  
    def __init__(self, cost_per_token: float = 0.0001):  
        self.cost_per_token = cost_per_token  
        self.token_usage = 0  
        self.api_calls = 0  
  
    def track_usage(self, tokens_used: int):  
        self.token_usage += tokens_used  
        self.api_calls += 1  
  
    def get_cost_estimate(self) -> float:  
        return self.token_usage * self.cost_per_token  
  
    def get_usage_report(self) -> Dict[str, Any]:  
        return {  
            "total_tokens": self.token_usage,  
            "total_api_calls": self.api_calls,
```

```
        "estimated_cost": self.get_cost_estimate()  
    }
```

## Security Best Practices

### 1. Input Validation

```
from pydantic import BaseModel, validator  
from typing import List, Optional  
  
class UserInput(BaseModel):  
    query: str  
    max_tokens: Optional[int] = 1000  
  
    @validator('query')  
    def validate_query(cls, v):  
        if len(v.strip()) == 0:  
            raise ValueError("Query cannot be empty")  
        if len(v) > 1000:  
            raise ValueError("Query too long")  
        return v  
  
    @validator('max_tokens')  
    def validate_max_tokens(cls, v):  
        if v is not None and (v < 1 or v > 4096):  
            raise ValueError("max_tokens must be between 1 and 4096")  
        return v
```

### 2. Tool Access Control

```
from enum import Enum  
from typing import Set  
  
class ToolPermission(Enum):  
    READ = "read"  
    WRITE = "write"  
    EXECUTE = "execute"  
  
class SecureTool(ToolInterface):  
    def __init__(self, tool: ToolInterface, required_permissions: Set[ToolPermission]):  
        self.tool = tool  
        self.required_permissions = required_permissions  
  
    def execute(self, inputs: Dict[str, Any], user_permissions: Set[ToolPermission]):  
        if not self.required_permissions.issubset(user_permissions):
```

```
        raise PermissionError("Insufficient permissions to execute tool")
    return self.tool.execute(inputs)
```

## Testing Framework

### 1. Unit Testing

```
import pytest
from unittest.mock import Mock, patch

def test_planning_agent():
    # Mock dependencies
    mock_llm = Mock()
    mock_tool = Mock()

    # Setup agent
    agent = PlanningAgent({"test_tool": mock_tool}, mock_llm)

    # Test plan creation
    mock_llm.predict.return_value = "Test plan response"
    plan = agent.create_plan("Test goal")

    assert plan.goal == "Test goal"
    assert len(plan.steps) > 0

    # Test plan execution
    mock_tool.execute.return_value = {"result": "success"}
    result = agent.execute_plan(plan)

    assert result["results"][-1]["result"] == "success"
```

### 2. Integration Testing

```
@pytest.mark.asyncio
async def test_research_assistant():
    assistant = ResearchAssistant()

    # Test complete research flow
    result = await assistant.research_topic("test topic")

    assert "results" in result
    assert len(result["results"]) > 0

    # Test error handling
```

```
with pytest.raises(AgentError):  
    await assistant.research_topic("")
```

## Deployment Considerations

### 1. Environment Configuration

```
from pydantic import BaseSettings  
  
class AgentConfig(BaseSettings):  
    max_concurrent_actions: int = 5  
    memory_ttl: int = 3600  
    token_limit: int = 4096  
    api_timeout: int = 30  
    debug_mode: bool = False  
    vector_store_path: str = "./vector_store"  
    log_level: str = "INFO"  
  
class Config:  
    env_prefix = "AGENT_"
```

#### ### 2. Container Configuration

```
```dockerfile  
FROM python:3.9-slim  
  
# Set working directory  
WORKDIR /app  
  
# Copy requirements  
COPY requirements.txt .  
  
# Install dependencies  
RUN pip install --no-cache-dir -r requirements.txt  
  
# Copy application code  
COPY . .  
  
# Set environment variables  
ENV AGENT_MAX_CONCURRENT_ACTIONS=5  
ENV AGENT_MEMORY_TTL=3600  
ENV AGENT_TOKEN_LIMIT=4096  
  
# Run the application  
CMD ["python", "main.py"]
```

### 2. Kubernetes Deployment

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: ai-agent
  namespace: ai-agents
spec:
  replicas: 3
  selector:
    matchLabels:
      app: ai-agent
  template:
    metadata:
      labels:
        app: ai-agent
    spec:
      containers:
        - name: ai-agent
          image: ai-agent:latest
          resources:
            requests:
              memory: "1Gi"
              cpu: "500m"
            limits:
              memory: "2Gi"
              cpu: "1000m"
          env:
            - name: AGENT_MAX_CONCURRENT_ACTIONS
              value: "5"
            - name: AGENT_MEMORY_TTL
              value: "3600"
          volumeMounts:
            - name: vector-store
              mountPath: /app/vector_store
      volumes:
        - name: vector-store
          persistentVolumeClaim:
            claimName: vector-store-pvc

---
apiVersion: v1
kind: Service
metadata:
  name: ai-agent-service
spec:
  selector:
    app: ai-agent
  ports:
    - port: 80
      targetPort: 8080
  type: LoadBalancer
```



# Performance Tuning

## 1. Memory Optimization

```
from typing import Optional, Dict, Any
import gc
import psutil

class MemoryOptimizer:
    def __init__(self, threshold_mb: int = 1000):
        self.threshold_mb = threshold_mb
        self.last_cleanup = time.time()

    def check_memory_usage(self) -> Dict[str, Any]:
        process = psutil.Process()
        memory_info = process.memory_info()

        return {
            "rss": memory_info.rss / 1024 / 1024, # MB
            "vms": memory_info.vms / 1024 / 1024, # MB
            "percent": process.memory_percent()
        }

    async def optimize_if_needed(self):
        memory_usage = self.check_memory_usage()

        if memory_usage["rss"] > self.threshold_mb:
            await self.perform_optimization()

    async def perform_optimization(self):
        # Clear Python's garbage collector
        gc.collect()

        # Clear tool caches
        for tool in self.tools.values():
            if hasattr(tool, 'clear_cache'):
                tool.clear_cache()

        # Compact memory if possible
        if hasattr(gc, 'collect'):
            gc.collect()
```

## 2. Load Balancing

```
class LoadBalancer:
    def __init__(self, max_workers: int = 3):
        self.workers = []
```

```
self.current_index = 0
self.lock = asyncio.Lock()

for _ in range(max_workers):
    self.workers.append(AgentWorker())

async def get_next_worker(self) -> 'AgentWorker':
    async with self.lock:
        worker = self.workers[self.current_index]
        self.current_index = (self.current_index + 1) % len(self.workers)
        return worker

async def process_request(self, request: Dict[str, Any]) -> Dict[str, Any]:
    worker = await self.get_next_worker()
    return await worker.process(request)
```

## Advanced Features

### 1. Multi-Agent Collaboration

```
class AgentTeam:
    def __init__(self, agents: Dict[str, 'Agent']):
        self.agents = agents
        self.coordinator = TeamCoordinator()

    async def collaborate(self, task: Dict[str, Any]) -> Dict[str, Any]:
        # Decompose task into subtasks
        subtasks = self.coordinator.decompose_task(task)

        # Assign subtasks to agents
        assignments = self.coordinator.assign_subtasks(subtasks, self.agents)

        # Execute subtasks in parallel
        results = await asyncio.gather(*[
            agent.execute_task(subtask)
            for agent, subtask in assignments.items()
        ])

        # Combine results
        return self.coordinator.combine_results(results)

class TeamCoordinator:
    def decompose_task(self, task: Dict[str, Any]) -> List[Dict[str, Any]]:
        # Implementation of task decomposition logic
        pass

    def assign_subtasks(
        self,
```

```

        subtasks: List[Dict[str, Any]],
        agents: Dict[str, 'Agent']
    ) -> Dict['Agent', Dict[str, Any]]:
        # Implementation of task assignment logic
        pass

    def combine_results(self, results: List[Dict[str, Any]]) -> Dict[str, Any]:
        # Implementation of result combination logic
        pass

```

## 2. Learning and Adaptation

```

class AdaptiveAgent:
    def __init__(self):
        self.performance_history = []
        self.strategy_weights = defaultdict(float)
        self.learning_rate = 0.1

    def update_strategy_weights(self, strategy: str, performance: float):
        self.strategy_weights[strategy] = (
            (1 - self.learning_rate) * self.strategy_weights[strategy] +
            self.learning_rate * performance
        )

    def select_strategy(self) -> str:
        if random.random() < 0.1: # Exploration
            return random.choice(list(self.strategy_weights.keys()))
        else: # Exploitation
            return max(
                self.strategy_weights.items(),
                key=lambda x: x[1]
            )[0]

    async def execute_with_adaptation(self, task: Dict[str, Any]) -> Dict[str, Any]:
        strategy = self.select_strategy()
        start_time = time.time()

        try:
            result = await self.execute_strategy(strategy, task)
            execution_time = time.time() - start_time

            # Update strategy weights based on performance
            performance = self.calculate_performance(result, execution_time)
            self.update_strategy_weights(strategy, performance)

        except Exception as e:
            pass

        return result

```

```
self.update_strategy_weights(strategy, 0.0) # Penalize failed strategy  
raise
```

## Maintenance and Monitoring

### 1. Health Checks

```
class HealthMonitor:  
    def __init__(self):  
        self.last_check = time.time()  
        self.health_metrics = {}  
  
    async def check_health(self) -> Dict[str, Any]:  
        current_time = time.time()  
  
        # Check system resources  
        memory_usage = psutil.virtual_memory()  
        cpu_usage = psutil.cpu_percent(interval=1)  
  
        # Check component health  
        component_health = await self.check_component_health()  
  
        # Update health metrics  
        self.health_metrics.update({  
            "last_check": current_time,  
            "uptime": current_time - self.last_check,  
            "memory_usage": memory_usage.percent,  
            "cpu_usage": cpu_usage,  
            "components": component_health  
        })  
  
        return self.health_metrics  
  
    async def check_component_health(self) -> Dict[str, str]:  
        health = {}  
  
        # Check each component  
        for component in self.components:  
            try:  
                await component.health_check()  
                health[component.name] = "healthy"  
            except Exception as e:  
                health[component.name] = f"unhealthy: {str(e)}"  
  
        return health
```

## 2. Metrics Collection

```
class MetricsCollector:
    def __init__(self):
        self.metrics = defaultdict(list)

    def record_metric(
        self,
        metric_name: str,
        value: float,
        tags: Optional[Dict[str, str]] = None
    ):
        self.metrics[metric_name].append({
            "timestamp": time.time(),
            "value": value,
            "tags": tags or {}
        })

    def get_metrics_summary(self) -> Dict[str, Any]:
        summary = {}

        for metric_name, values in self.metrics.items():
            summary[metric_name] = {
                "count": len(values),
                "mean": statistics.mean(v["value"] for v in values),
                "max": max(v["value"] for v in values),
                "min": min(v["value"] for v in values)
            }

        return summary
```

## Iteration and Refinement

Post-execution, the agent reflects on the accuracy of its findings, refines data extraction methods, and improves based on feedback.

## Final Thoughts

Agentic AI will transform artificial intelligence, enabling systems that can reason, plan, and act independently. This guide provides a foundation for implementing such systems, but remember that each implementation may require specific adjustments based on use case requirements and constraints.

AI

Agents

Ai Agent

Genai

LLm



Follow

## Published in aingineer

4 Followers · Last published 2 days ago

AI Engineer by [nigamg.ai](#)



Following

## Written by Gaurav Nigam

31 Followers · 30 Following

Leader | Learner | All about Tech

## No responses yet



What are your thoughts?

Respond

## More from Gaurav Nigam and aingineer



In aingineer by Gaurav Nigam

## Enterprise GraphRAG: Building Production-Grade LLM Applications with Knowledge Graphs

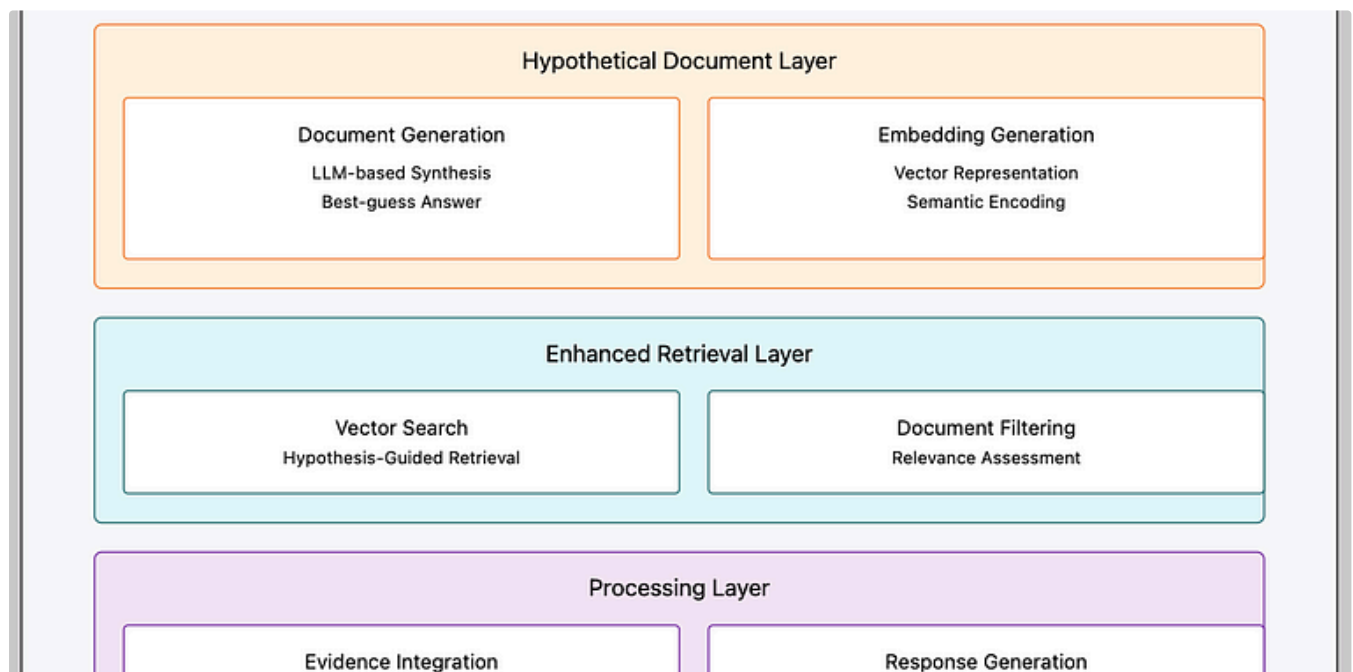
An In-Depth Guide for Engineering Leaders and Architects



Nov 23



4

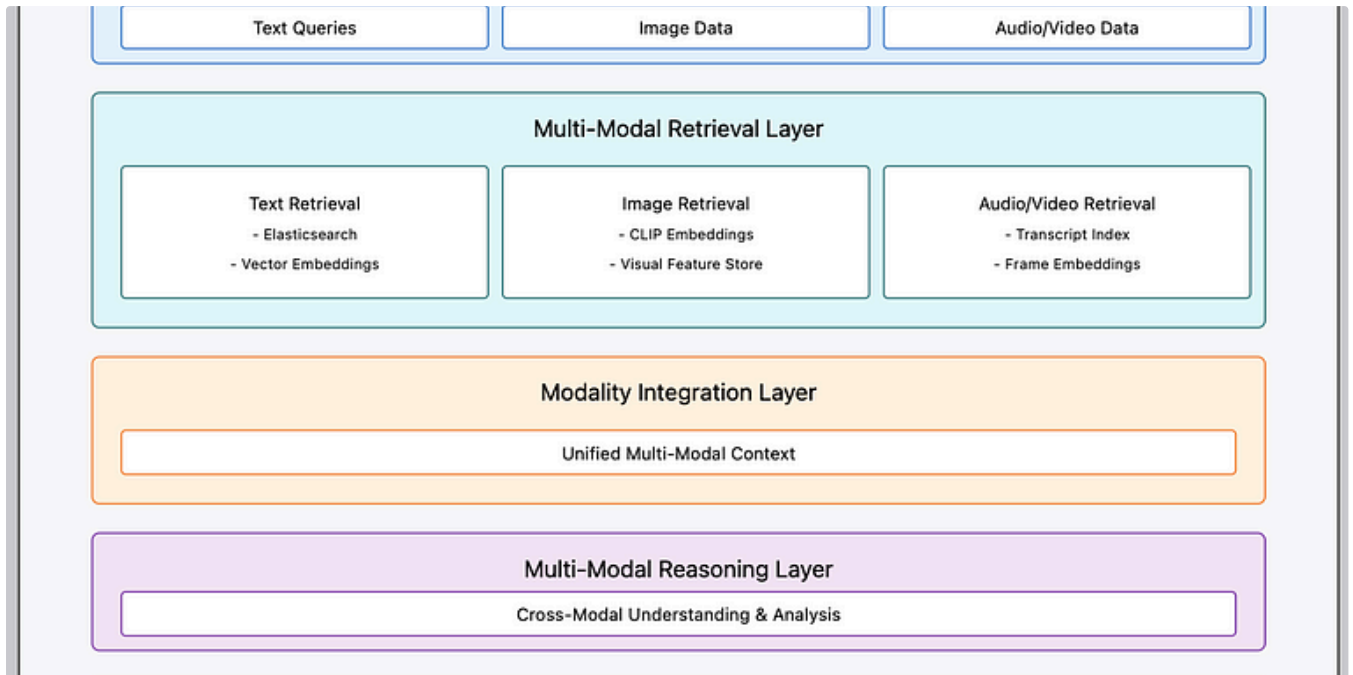



In aingineer by Gaurav Nigam

## A Complete Guide to Implementing HyDE RAG

Retrieval-Augmented Generation systems depend heavily on the quality of retrieved documents to provide accurate, context-rich answers. HyDE...

★ Dec 17




 In aingineer by Gaurav Nigam

## A Complete Guide to Implementing Multi-Modal RAG

As enterprises expand their AI capabilities, the need to handle and reason over diverse data types—such as text, images, audio, and...

★ Dec 16





 In aingineer by Gaurav Nigam



# Agentic RAG with ReAct: A Practical Guide to Building Autonomous Agents

An In-Depth Guide for Engineering Teams

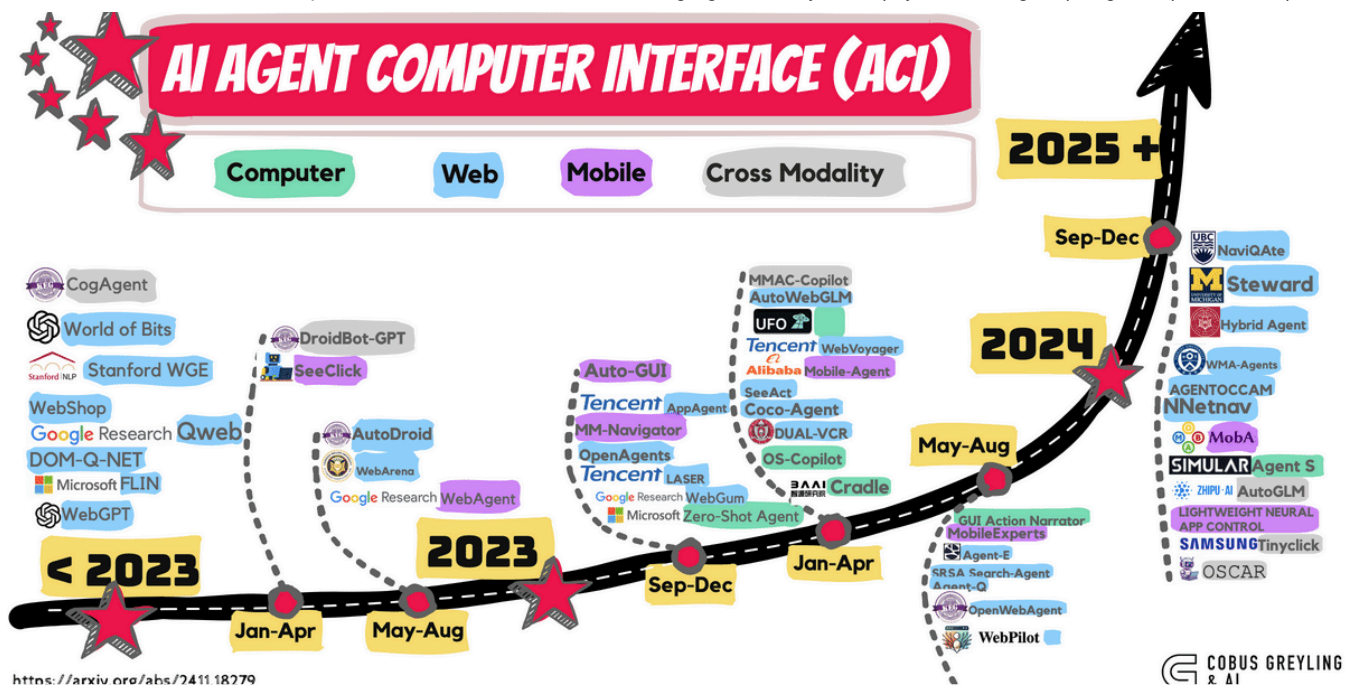
 Nov 24  1

See all from Gaurav Nigam

See all from aingineer

## Recommended from Medium

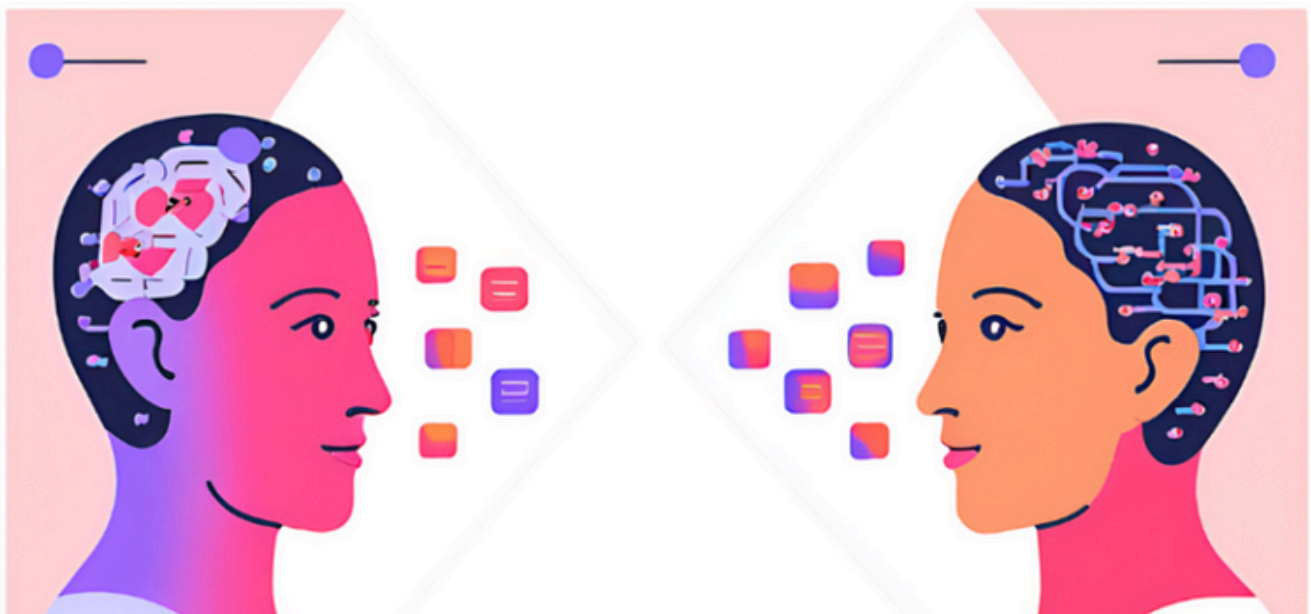


Cobus Greyling

## AI Agent Computer Interface (ACI)

Revolutionising User Interactions & How AI Agents Are Moving Beyond Models to Frameworks, Redefining the Future of Computer Interfaces

5d ago 11



MAA1

## What is Agentic AI?

In the current craze about AI, it's sometimes easy to forget that AI can be used for more than creating chatbots or generating images...

Oct 19

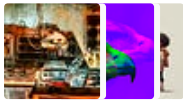


## Lists



### Generative AI Recommended Reading

52 stories · 1565 saves



### What is ChatGPT?

9 stories · 487 saves



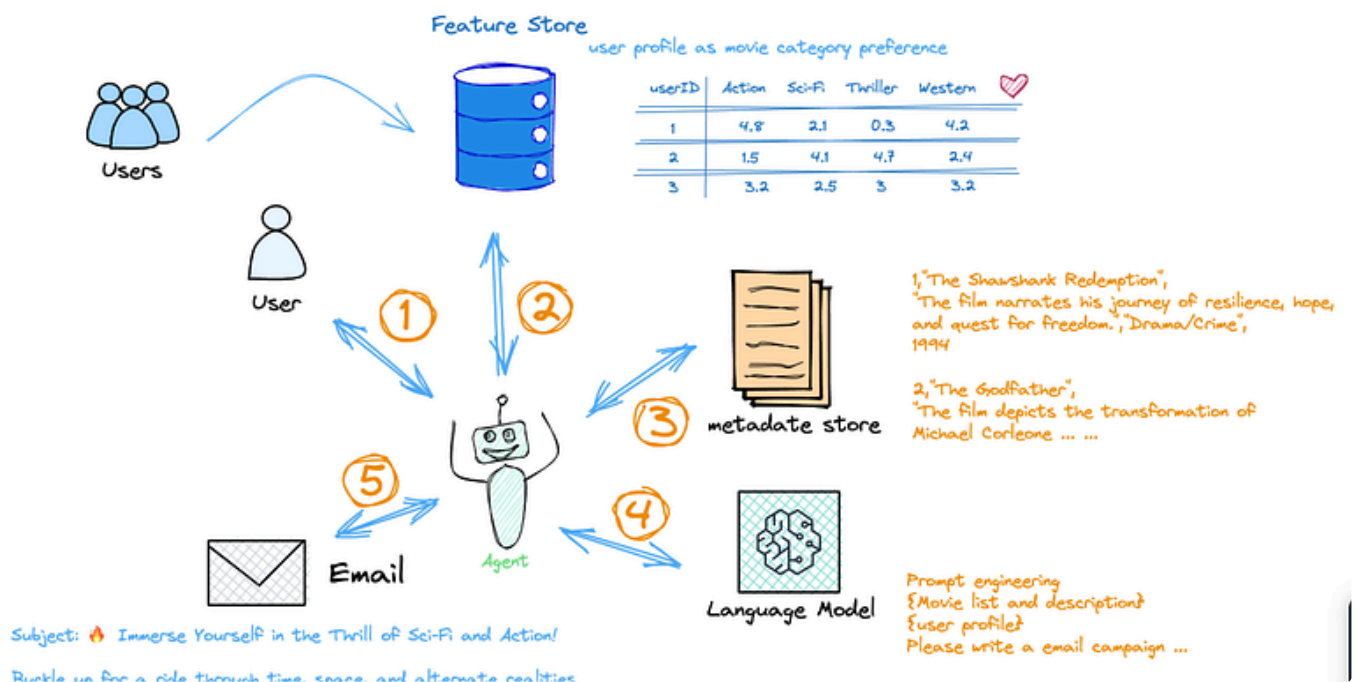
### Natural Language Processing

1874 stories · 1497 saves



### The New Chatbots: ChatGPT, Bard, and Beyond

12 stories · 530 saves



Lekha Priya

## Introduction to Agentic AI and Its Design Patterns

Artificial Intelligence has come a long way from rule-based systems to sophisticated autonomous agents capable of making decisions and...

Dec 9

84


1



MYSCALE

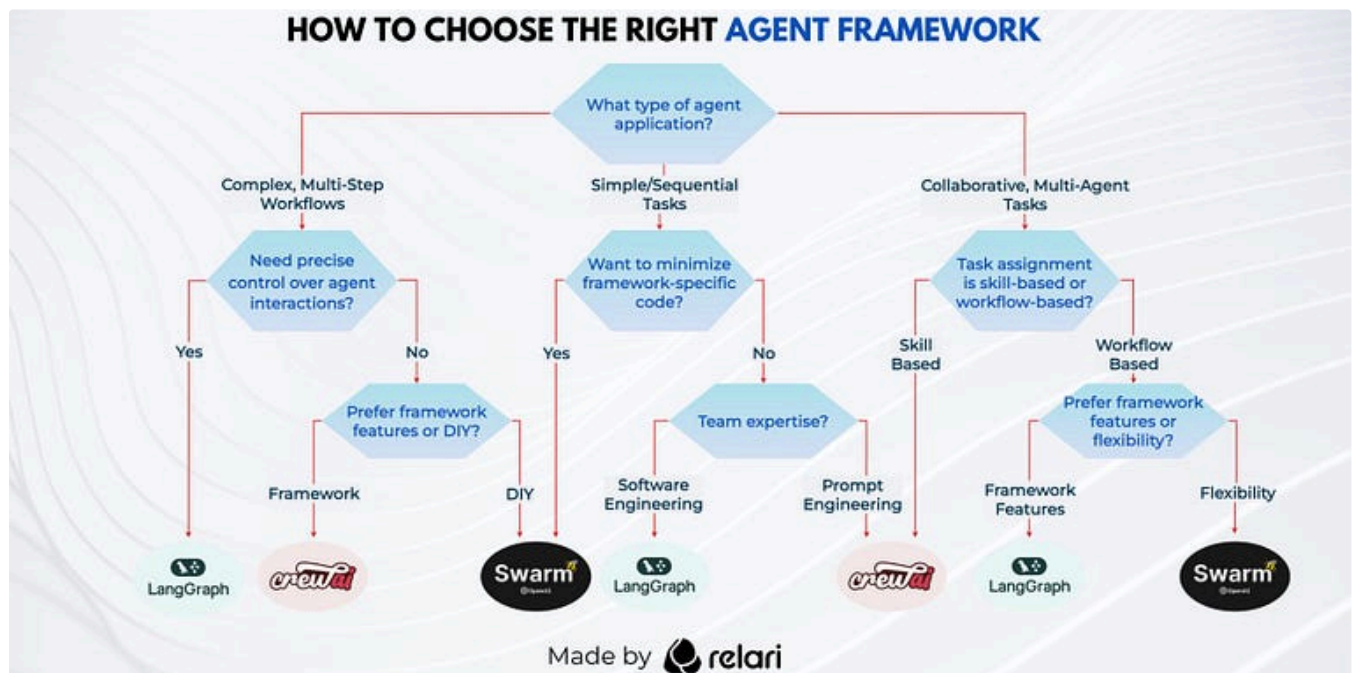
# Agentic AI vs Generative AI

## Understanding the Key Differences and Impacts

 MyScale

### Agentic AI vs Generative AI: Understanding the Key Differences and Impacts

Large Language Models (LLMs) like GPT can generate text, answer questions, and assist with many tasks. However, they are reactive, meaning...

Dec 4  123  4 In Relari Blog by Yi Zhang

### Choosing the Right AI Agent Framework: LangGraph vs CrewAI vs OpenAI Swarm

In-depth comparison of agent orchestration with the same Agentic Finance App built using 3 different frameworks.

Dec 3

👍 195

💬 3

 Data Journal

## How to Use User Agents for Web Scraping?

In this article, I'll explain what user agents are, why they matter for web scraping, and how you can use them to avoid getting blocked and...

Sep 3

[See more recommendations](#)