श्रवण,मनन,निदिध्यासन

AgenticAI Overview

Vaibhav Zodge

7020616260

info.dnyanyog@gmail.com

https://www.dnyanyog.org

https://github.com/zodgevaibhav

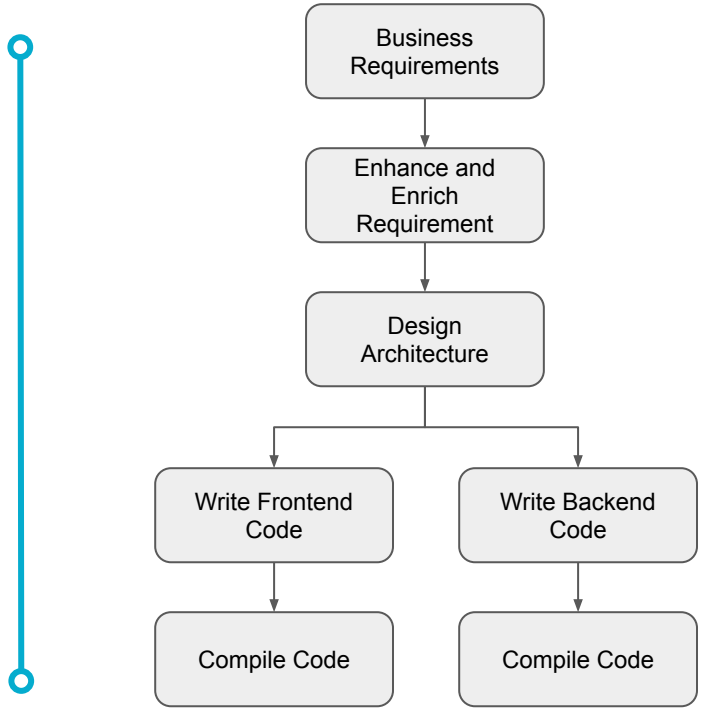
GenAI Program : Use Case

Gen AI is generation of Text, Image, Video or Audio using LLM

There are many real life usage of such system (GenAI)

Agentic SDLC
- Read RAW Requirement
- Groom and document requirement
- Design Architect
- Write Backend and Frontend Code
- Compile and Build Code
- Write Unit Test
- Run Tests
- Fix Bugs
- Deploy application
- Monitor and improve

Agentic Hospital Management
- Read the patient information
- Case History Summarization
- Help in diagnosis and prescription
- Appointment Schedule
- etc…



AI SDLC

```python
from openai import OpenAI
import os, shutil
from datetime import datetime

client = OpenAI()

BASE = "runs"
FEATURES = """Feature: Patient Management API
Scenario: CRUD operations
"""

DEV PROMPT = """
You are a Senior Full Stack Developer.

Generate a FULLY RUNNABLE project.

Backend:
- Spring Boot
- Maven
- REST CRUD
- Proper package structure
- Proper naming conventions
- Make sure to add CORS configuration to allow
frontend-backend communication at localhost:3000
port

Frontend:
- React
- npm
- Basic CRUD UI

STRICT RULES:
1. Use ONLY the format below
2. Every file MUST start with ===FILE===
3. Use full relative file paths
4. No explanations, no markdown
"""
```

```python
def clean workspace ():
    if os.path.exists(BASE):
        shutil.rmtree(BASE)
    os.makedirs(BASE)

def run agent (prompt, input_text):
    response =
client.chat.completions.create(
        model="gpt-4o-mini",
        messages=[
            {"role": "system", "content":
prompt},
            {"role": "user", "content":
input text}
        ],
        temperature=0.1
    )
    return
response.choices[0].message.content

def write code files (output):
    blocks = output.split("===FILE===")
    for block in blocks:
        block = block.strip()
        if not block:
            continue
        lines = block.splitlines()
        path = lines[0]
        content = "\n".join(lines[1:])
        os.makedirs(os.path.dirname(path),
exist ok=True)
        with open(path, "w") as f:
            f.write(content)
        print(f"✓ {path}")
```

```python
def main(requirement):
    clean_workspace()

    run dir = os.path.join(BASE,
f"run_{datetime.now().strftime('%Y%m%d_%H%M%S')}")
    os.makedirs(run dir)
    os.chdir(run_dir)

    # Requirement
    os.makedirs("requirements", exist_ok=True)
    with open("requirements/requirement.feature", "w") as f:
        f.write(requirement)

    # Architecture
    arch = run_agent(
        """You are a Software Architect.
        Produce architecture.md and PlantUML.""",
        requirement
    )
    os.makedirs("architecture", exist_ok=True)
    with open("architecture/architecture.md", "w") as f:
        f.write(arch)

    # Developer
    dev_output = run_agent(DEV_PROMPT, arch)
    materialize_files(dev_output)

    print("\n✅ FULL PROJECT GENERATED")

if __name__ == "__main__":
    main(FEATURES)
```

Let’s understand the Code

Step 1: Requirement Refinement

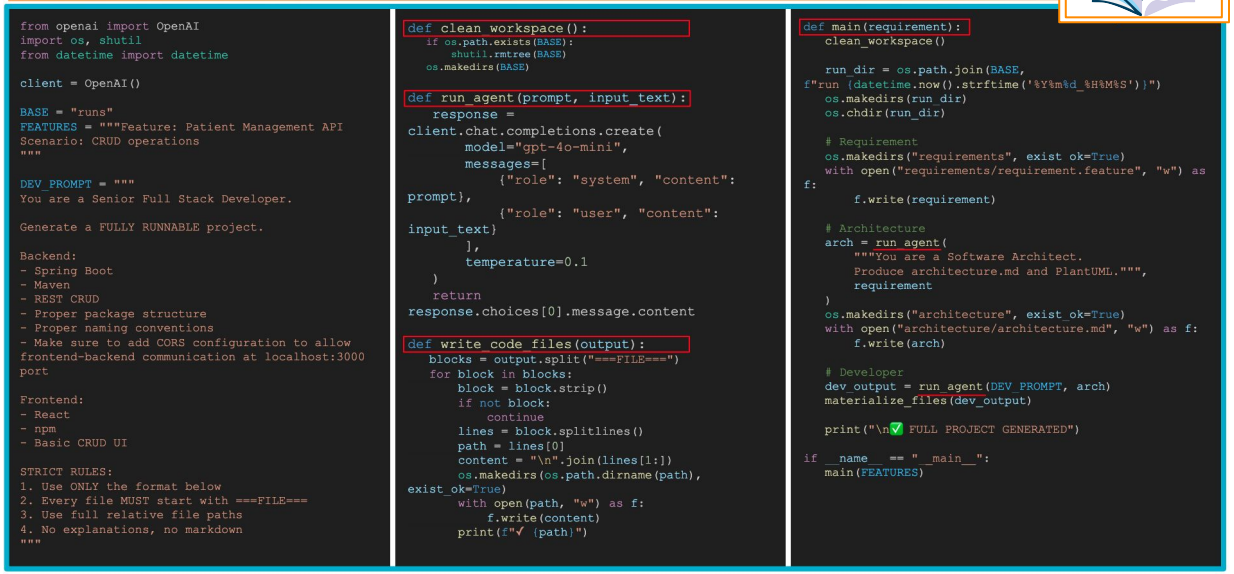
- Raw requirements are given as input to the model
- A LLM is used to refine the raw requirements
- A predefined prompt guides the refinement process
- Output: **Well-structured and refined requirements**

Step 2: Architecture Design

- Refined requirements are used as input
- An LLM helps decide the application architecture
- A specific architecture prompt is used
- Output: **Architecture design document**

Step 3: Code Generation

- Architecture document is given to the LLM
- LLM generates **Backend and Frontend code**
- Separate prompts are used for code generation
- Backend and Frontend generation functions run **in parallel**



```
from openai import OpenAI
import os, shutil
from datetime import datetime

client = OpenAI()

BASE = "runs"
FEATURES = """Feature: Patient Management API
Scenario: CRUD operations
"""

DEV_PROMPT = """
You are a Senior Full Stack Developer.

Generate a FULLY RUNNABLE project.

Backend:
- Spring Boot
- Maven
- REST CRUD
- Proper package structure
- Proper naming conventions
- Make sure to add CORS configuration to allow
frontend-backend communication at localhost:3000
port

Frontend:
- React
- npm
- Basic CRUD UI

STRICT RULES:
1. Use ONLY the format below
2. Every file MUST start with ===FILE===
3. Use full relative file paths
4. No explanations, no markdown
"""
```

```
def clean_workspace():
    if os.path.exists(BASE):
        shutil.rmtree(BASE)
    os.makedirs(BASE)

def run_agent(prompt, input_text):
    response =
client.chat.completions.create(
        model="gpt-4o-mini",
        messages=[
            {"role": "system", "content":
prompt},
            {"role": "user", "content":
input_text}
        ],
        temperature=0.1
    )
    return
response.choices[0].message.content

def write_code_files(output):
    blocks = output.split("===FILE===")
    for block in blocks:
        block = block.strip()
        if not block:
            continue
        lines = block.splitlines()
        path = lines[0]
        content = "\n".join(lines[1:])
        os.makedirs(os.path.dirname(path),
exist_ok=True)
        with open(path, "w") as f:
            f.write(content)
        print(f"✓ {path}")
```

```
def main(requirement):
    clean_workspace()

    run_dir = os.path.join(BASE,
f"run_{datetime.now().strftime('%Y%m%d_%H%M%S')}")
    os.makedirs(run_dir)
    os.chdir(run_dir)

    # Requirement
    os.makedirs("requirements", exist_ok=True)
    with open("requirements/requirement.feature", "w") as
f:
        f.write(requirement)

    # Architecture
    arch = run_agent(
        """You are a Software Architect.
        Produce architecture.md and PlantUML.""",
        requirement
    )
    os.makedirs("architecture", exist_ok=True)
    with open("architecture/architecture.md", "w") as f:
        f.write(arch)

    # Developer
    dev_output = run_agent(DEV_PROMPT, arch)
    materialize_files(dev_output)

    print("\n✅ FULL PROJECT GENERATED")

if __name__ == "__main__":
    main(FEATURES)
```

What we needed ?

- Different Prompts and execution functions
- State Management (Output of one function as input to other)
- Decide the flow of execution (Requirement > Architecture > Backend Code > Frontend Code)
- Parallel Execution

What More We want ?

- Retry mechanism (Iterative mechanism)
- Conditional LLM calls (routing & decision making)
- Observability & Debuggability
- …. & many more feature to make it work on production

LangGraph

Lang Graph is a framework built on top of LangChain

Supports Graph Based workflows than Linear workflow

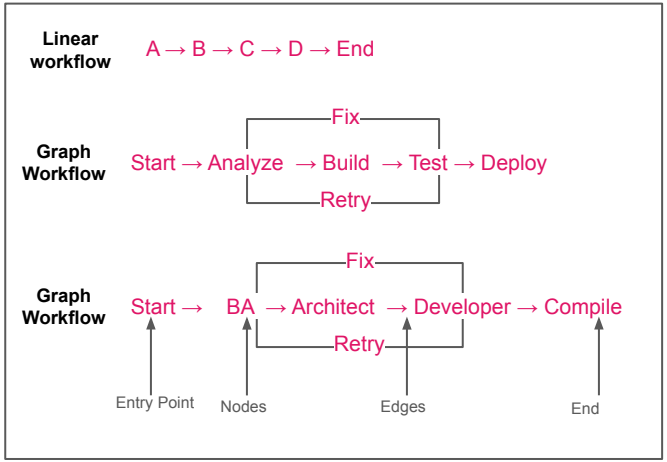
- Graph like structure
  - Nodes → Steps/Agent/Function
  - Edges → Path that decide what runs next
  - Entrypoint → Decide which Node to start
  - End → Decide where to stop
- Multiple Path
- Conditional decision of next step
- Loops possible

State Management between nodes/agents

Multiple agent management system

Conditional routing supported and inbuilt features for retries

Useful for multiple agent & decision based workflows, long running automation



Linear vs Graph Workflow

Edges Examples

Sequential execution (Linear)

```
graph.add_edge("architect", "developer")
```

Parallel execution (Fan-out)

```
graph.add_edge("developer", "backend_compile")
graph.add_edge("developer", "frontend_compile")
```

Join execution (Fan-in / Sync)

```
graph.add_edge("backend_compile", "compile_done")
graph.add_edge("frontend_compile", "compile_done")
```

Loop execution (Retry / Iterate)

```
graph.add_edge("test", "fix_code")
graph.add_edge("fix_code", "test")
```

Conditional execution (Decision routing)

```
graph.add_conditional_edges(
    "test",
    lambda s: "pass" if s["ok"] else "fail",
    {"pass": END, "fail": "fix_code"}
)
```

Early termination

```
graph.add_edge("deploy", END)
```

With **parallel execution** make sure shared keys are not mutated, else will get error
Either use specific keys return
ex.

Wrong

```
def backend_compiler_node(state):
    return state

def frontend_compiler_node(state):
    return state
```

Correct

```
def backend_compiler_node(state):
    return {"backend_build_ok": True}

def frontend_compiler_node(state):
    return {"frontend_build_ok": True}
```

Monitoring using LangSmith

Create LangSmith Account

URL : https://smith.langchain.com

Use free tier and generate the api key

Set below parameters in environmental variables

```
export LANGSMITH_API_KEY= <<KEY>>
export LANGSMITH_TRACING=true
export LANGSMITH_ENDPOINT=https://api.smith.langchain.com
```

Agentic Design Patterns

Refinement Loop Pattern

LangGraph primitives:
→ add_edge() with a loop-back edge + quality check node
How: Routes output back to the same node until acceptance criteria are met.

Retry with Feedback (Self-Healing) Pattern

LangGraph primitives:
→ add_conditional_edges() + retry counter in state
How: On failure, routes back to the generator with error context.

Guardrail / Gatekeeper Pattern

LangGraph primitives:
→ Dedicated validation node + conditional routing
How: Validator node decides PASS → continue / FAIL → block or fix.

Decompose–Solve–Compose Pattern

LangGraph primitives:
→ Parallel nodes + merge/composer node
How: Subtasks write to different state keys and are combined later.

Dual-Agent Verification Pattern

LangGraph primitives:
→ Sequential nodes (Generator → Reviewer)
How: Reviewer updates approval or feedback in state.

Conditional Routing Pattern

LangGraph primitives:
→ add_conditional_edges()
How: Routing function chooses next node based on state values.

Human-in-the-Loop Pattern

LangGraph primitives:
→ Pause/wait node + external signal + resume edge
How: Workflow halts until human approval is received.

Memory-Backed Decision Pattern

LangGraph primitives:
→ External memory store + decision node
How: Node consults past outcomes to influence routing or prompts.

Spec-Driven Generation Pattern

LangGraph primitives:
→ Typed state (Pydantic / TypedDict) + schema validation node
How: Generator emits structured spec enforced by validator.

Timeout & Escalation Pattern

LangGraph primitives:
→ Retry counter in state + conditional exit edge
How: Exceeds threshold → escalate, fallback, or terminate safely.

Sandbox Execution Pattern

LangGraph primitives:
→ Execution node + error capture in state
How: Runs code/scripts in isolation and records logs/errors.

Fallback Model / Tool Pattern

LangGraph primitives:
→ Conditional routing + alternate agent/tool nodes
How: Failure routes to backup model or tool.

Validation-First Pattern

LangGraph primitives:
→ Entry validation node + early termination
How: Inputs are validated before costly generation steps.

Checkpoint & Replay Pattern

LangGraph primitives:
→ State checkpointer + resumable graph
How: Enables replay, debugging, and auditability.

Progressive Disclosure Pattern

LangGraph primitives:
→ Sequential nodes with incremental context enrichment
How: Each node adds detail before passing state forward.

“ Production agent systems succeed by structure, validation, and control — not by prompt size. ”

Agentic AI : Note

Agentic systems are **state machines with intelligence**, not autonomous magic.

Reliability comes from **explicit state transitions**, not from clever prompts.

Every agent must have **one responsibility** and a clear success criterion.

If an agent cannot be **validated**, it should not be automated.

Good agent architectures **fail fast, fail safe, and recover automatically**.

Most “intelligence” in production agents lives in **routing logic**, not in LLMs.

Agent workflows should be **observable first, intelligent second**.

Parallelism increases speed but **multiplies state-management complexity**.

Determinism is achieved through **constraints, schemas, and retries**, not temperature.

A workflow without guardrails is **a liability, not an AI system**.

Agent loops must always have **explicit exit conditions**.

Memory is useful only when it **changes future decisions**.

Agents should produce **machine-consumable outputs**, not human prose.

Validation is cheaper than regeneration — **validate early, validate often**.

Production agents are **composed systems**, not single “smart” agents.

Human-in-the-loop is a **design choice**, not a failure mode.

Agent failures should **teach the system**, not just retry blindly.

Code execution is an agent too — **treat tools as first-class nodes**.

Good agent graphs are **boring, predictable, and repeatable**.

If a workflow cannot be diagrammed, it is probably **not production-ready**.

Agentic AI : Note

1. Entry & Control Layer

Accepts raw input (user / system / event)
Performs **early validation & sanitization**
Initializes **shared workflow state**

Fail fast before intelligence is applied

2. Orchestration Layer (Graph Engine)

Deterministic **state machine**
Nodes = agents / tools
Edges = control flow & decisions
Conditional routing & loops

Control flow is the backbone of agentic systems

3. Agent Layer (Single Responsibility)

Requirement Refiner
Planner / Architect
Generator (Code / Content / Spec)
Reviewer / Critic
Executor (Build / Test / Run)

Agents think locally, architecture thinks globally

4. State Layer (Source of Truth)

Typed, explicit, versioned state
Immutable updates
Retry counters, flags, errors

State is the memory, context, and contract

5. Validation & Guardrails

Schema validation
Business rules
Security & policy checks
Output gating

Nothing proceeds without validation

6. Execution & Tooling

Sandboxed code execution
CLI, Docker, APIs, scripts
Logs & errors captured into state

Tools are first-class citizens, not side effects

7. Memory & Learning

Past failures & decisions
Conventions & preferences
Optional long-term storage

Memory exists only to change future behavior

8. Recovery & Escalation

Retry with feedback
Fallback models/tools
Human-in-the-loop
Safe termination

Self-healing beats perfect generation

9. Observability & Persistence

State checkpoints
Agent outputs
Replay & audit

If you can't replay it, you can't trust it

Agentic AI systems succeed by structure, validation, and control — not by prompt size.

LangGraph patterns emerge from how you combine nodes, state, and conditional edges — not from special features.