# Project Overview

You are tasked with building an agentic AI system in the form of a FastAPI microservice. This system will process free-form IT requests, determine which specialized agents are required, execute those agents to produce outputs, and return a comprehensive response that combines all results.

## Core Requirements

Your system must:

1. Plan which specialist agents are needed based on the input request
2. Execute those LLM-powered agents to produce concrete outputs
3. Support approval workflows when requested by the caller
4. Return a structured JSON or Markdown response with combined results

No private knowledge base or database is required—your agents can rely on the capabilities of the LLM and/or call public data sources that you choose to integrate.

# Technical Architecture

Your implementation must include the following components:

## API Layer

FastAPI service with these endpoints:

* **POST /api/v1/execute** - Process new requests
* **POST /api/v1/plans/{id}/approve** - Approve a pending plan
* **POST /api/v1/plans/{id}/reject** - Reject a pending plan
* **GET /api/v1/tasks/{id}** - Get task status and results

## Agent Layer

At minimum, you must implement these agents:

* **CoordinatorAgent**: Parses the request, creates execution plans, and merges outputs
* **DiagnosticAgent**: Performs root-cause analysis and provides ranked fixes
* **AutomationAgent**: Generates and syntax-checks PowerShell/Bash/Azure CLI scripts
* **WriterAgent**: Transforms results into structured content (email, SOP, summary) You're welcome to add additional agents if you think they would enhance your solution.

## Workflow Layer

You must use LangGraph for orchestrating the workflow:

* Implement a **CoordinatorGraph** for the overall execution flow
* Create at least one specialist graph with conditional edges (e.g., **DiagnosticGraph**)

## LLM Integration

* Use OpenAI's or Any LLM

## Advanced Techniques

Include at least one integration with DSPy, MCP such as:

* A DSPy router component
* An MCP context pruner
* Another appropriate integration of your choice

## Testing

Implement the 5 specific pytest cases outlined in section 6

## Documentation

* Complete README.md with setup and usage instructions
* Architecture diagram (using Mermaid, PNG,...)

# Required Test Cases

You must implement these specific pytest cases:

|  |  |
| --- | --- |
| **Test Name** | **Purpose** |
| test\_happy\_path | Verify that Example A returns "completed" status and includes non-empty diagnosis and script sections |
| test\_approval\_flow | Verify that Example B begins with "waiting\_approval" and transitions to "completed" after approval |

|  |  |
| --- | --- |
| test\_agent\_retry | Simulate a failure in the AutomationAgent and verify that the Coordinator either retries or handles the failure gracefully |
| test\_script\_compiles | Verify that generated PowerShell/Bash scripts pass syntax checks (using pwsh -Command {…} or bash -n) |

# API Contract & Examples

## POST /api/v1/execute

### Request Fields:

|  |  |  |
| --- | --- | --- |
| **Field** | **Type** | **Description** |
| request | string | Natural-language description of what the assistant should do |
| require\_approval | boolean | If true, pause after planning for explicit approval |

**Example A - Direct Execution (No Approval) Request:**

http

POST /api/v1/execute

Content-Type: application/json

{

"request": "Diagnose why Windows Server 2019 VM cpu01 hits 95%+

CPU, generate a PowerShell script to collect perfmon logs, and draft an email to management summarising findings.",

"require\_approval": false

}

### Internal Flow:

|  |  |
| --- | --- |
| **Phase** | **Action** |

|  |  |
| --- | --- |
| Plan | CoordinatorAgent detects three sub-tasks (diagnostic, script, email) |
| Execut e | DiagnosticAgent → root cause & fixes<br>AutomationAgent → PowerShell script  + lint pass<br>WriterAgent → management email |
| Merge | Coordinator combines results into single response |

**Expected Response:**

json

{

"task\_id": "123e4567-...", "status": "completed", "diagnosis": {

"root\_cause": "Wsappx runaway process",

"evidence": ["perfmon shows high kernel time ..."], "solutions": [

{ "title": "Disable Superfetch", "confidence": "high" },

{ "title": "Install KB500XYZ", "confidence": "medium" }

]

},

"script": {

"language": "powershell",

"code": "New-Item C:\\logs -Force; logman start ...", "lint\_passed": true

},

"email\_draft": "Hello team, ...", "duration\_seconds": 42

}

### Example B - Approval Flow Initial Request:

http

POST /api/v1/execute

Content-Type: application/json

{

"request": "Create Azure CLI commands to lock RDP (3389) on my

three production VMs to 10.0.0.0/24 and pause for approval before outputting the commands.",

"require\_approval": true

}

### Initial Response (waiting for approval):

json

{

"task\_id": "plan-456", "status": "waiting\_approval", "plan": {

"steps": [

"Generate NSG rules", "Generate rollback script"

],

"summary": "Will restrict 3389 inbound to 10.0.0.0/24 on vm-a, vm-b, vm-c"

}

}

### Approval Actions:

bash

*# approve and resume*

curl -X POST /api/v1/plans/plan-456/approve

*# reject*

curl -X POST /api/v1/plans/plan-456/reject

### Response After Approval:

json

{

"status": "completed", "commands": [

"az network nsg rule create ...",

"az network nsg rule delete ... # rollback"

]

}

## GET /api/v1/tasks/{id}

Returns the current state of a task with one of these status values:

* + - active
    - waiting\_approval
    - completed
    - failed
    - etc.

The response includes any partial results or error information.

# Deliverables

Your submission must include:

1. Git repository (or ZIP file) containing all code, tests, README, and architecture diagram
2. A screen capture demonstrating Example A and Example B flows
3. Postman or curl collection for testing the API