# Complete Module Code

**File: Symposium/.gitignore**

*# Python virtual environment*

symposium\_env/

*# API keys*

.env

*# Python cache files*

\_\_pycache\_\_/

\*.pyc

**File: Symposium/publish.bat**

@**echo** off

*REM =================================================================*

*REM == Publishes The Symposium to GitHub and Google Drive ==*

*REM =================================================================*

*REM Change the current directory to the script's directory.*

**cd** /d "%~dp0"

*REM Prompt for a commit message*

**echo**.

**set** **/p** commitMessage="Enter a commit message for this update: "

*REM =================================================================*

**echo**.

**echo** [STEP 1] Pushing changes to GitHub...

*REM Add all changes to the staging area*

git add .

*REM Commit the changes with the provided message*

git commit -m "%commitMessage%"

*REM Push the commit to the 'main' branch on the remote 'origin'*

git push origin main

**echo**.

**echo** GitHub push complete.

**echo** =================================================================

**echo**.

**echo** [STEP 2] Syncing project to Google Drive...

**echo** Excluding .env, .git, \_\_pycache\_\_, and symposium\_env...

rclone sync . "gdrive:Symposium" --exclude "/symposium\_env/\*\*" --exclude "/.git/\*\*" --exclude "/\_\_pycache\_\_/\*\*" --exclude ".env" --progress

**echo**.

**echo** Google Drive sync complete.

**echo** =================================================================

**echo**.

**echo** PUBLISH COMPLETE - CONTEXT IS NOW SHARED

**echo** =================================================================

**pause**

**File: Symposium/run.bat**

@**echo** off

*REM =================================================================*

*REM == Runner for The Multi-LLM Symposium ==*

*REM =================================================================*

*REM Change the current directory to the script's directory.*

**cd** /d "%~dp0"

**echo** [STEP 1] Activating virtual environment...

**call** "symposium\_env\Scripts\activate.bat"

**echo**.

**echo** [STEP 2] Executing the Python script...

python src/symposium\_manager\_v2.py

**echo**.

**echo** =================================================================

**echo** == SCRIPT FINISHED ==

**echo** =================================================================

**pause**

**File: Symposium/src/models/agent.py**

*"""*

*Core Agent model with LLM provider integration.*

*Contains Agent class, provider enums, and model factory logic.*

*"""*

**import** **os**

**from** **dataclasses** **import** dataclass, field

**from** **typing** **import** List, Optional, Any

**from** **enum** **import** Enum

**from** **dotenv** **import** load\_dotenv

*# LangChain LLM Provider imports*

**from** **langchain\_anthropic** **import** ChatAnthropic

**from** **langchain\_google\_genai** **import** ChatGoogleGenerativeAI

**from** **langchain\_groq** **import** ChatGroq

**from** **langchain\_mistralai** **import** ChatMistralAI

**from** **langchain\_cerebras** **import** ChatCerebras

**from** **langchain\_core.tools** **import** Tool

*# Load environment variables*

load\_dotenv()

**class** **ModelProvider**(Enum):

*"""Enumeration of supported LLM providers."""*

GEMINI = "gemini"

CLAUDE = "claude"

GROQ = "groq"

MISTRAL = "mistral"

CEREBRAS = "cerebras"

**class** **AgentTier**(Enum):

*"""Classification of agent capability and cost."""*

SENIOR\_FELLOW = "senior\_fellow" *# Premium models for complex reasoning*

RESEARCH\_ASSISTANT = "research\_assistant" *# Cheaper models for tool use*

@dataclass

**class** **Agent**:

*"""Represents a configured LLM agent with specific capabilities."""*

name: str

provider: ModelProvider

model\_name: str

tier: AgentTier

persona\_prompt: str

tools: List[Tool] = field(default\_factory=list)

temperature: float = 0.0

model\_instance: Optional[Any] = field(default=**None**, init=**False**)

**def** \_\_post\_init\_\_(self):

*"""Initialize the actual LLM instance after dataclass creation."""*

self.model\_instance = self.\_create\_model\_instance()

**def** \_create\_model\_instance(self) -> Any:

*"""Factory method to create the appropriate LLM instance."""*

**try**:

**if** self.provider == ModelProvider.GEMINI:

**return** ChatGoogleGenerativeAI(

model=self.model\_name,

temperature=self.temperature

)

**elif** self.provider == ModelProvider.CLAUDE:

**return** ChatAnthropic(

model=self.model\_name,

temperature=self.temperature

)

**elif** self.provider == ModelProvider.GROQ:

**return** ChatGroq(

model\_name=self.model\_name,

temperature=self.temperature

)

**elif** self.provider == ModelProvider.MISTRAL:

**return** ChatMistralAI(

model\_name=self.model\_name,

temperature=self.temperature

)

**elif** self.provider == ModelProvider.CEREBRAS:

**return** ChatCerebras(

model=self.model\_name,

temperature=self.temperature

)

**else**:

**raise** **ValueError**(f"Unsupported provider: **{**self.provider**}**")

**except** **Exception** **as** e:

print(f"Error creating model instance for **{**self.name**}**: **{**e**}**")

**return** **None**

**File: Symposium/src/models/symposium.py**

*"""*

*Agent registry and symposium management.*

*Handles agent creation, registration, and retrieval.*

*"""*

**from** **typing** **import** Dict, List, Optional

**from** **.agent** **import** Agent, ModelProvider, AgentTier

**class** **Symposium**:

*"""Main class for managing the agent registry and their relationships."""*

**def** \_\_init\_\_(self):

self.agents: Dict[str, Agent] = {}

self.\_initialize\_default\_agents()

**def** \_initialize\_default\_agents(self) -> **None**:

*"""Create the standard agent roster based on verified providers."""*

*# Senior Fellows (premium models)*

self.register\_agent(Agent(

name="Gemini",

provider=ModelProvider.GEMINI,

model\_name="gemini-2.5-pro",

tier=AgentTier.SENIOR\_FELLOW,

persona\_prompt="You are Gemini, a visionary Senior Fellow. Your role is strategic thinking, creative problem-solving, and comprehensive analysis."

))

self.register\_agent(Agent(

name="Claude",

provider=ModelProvider.CLAUDE,

model\_name="claude-sonnet-4-20250514",

tier=AgentTier.SENIOR\_FELLOW,

persona\_prompt="You are Claude, a analytical Senior Fellow. Your role is critical evaluation, logical reasoning, and identifying potential flaws."

))

*# Research Assistants (tool-using models)*

self.register\_agent(Agent(

name="Groq\_Assistant",

provider=ModelProvider.GROQ,

model\_name="llama-3.3-70b-versatile",

tier=AgentTier.RESEARCH\_ASSISTANT,

persona\_prompt="You are a Research Assistant. Your role is to execute specific tasks using tools and provide concise, factual results."

))

self.register\_agent(Agent(

name="Mistral\_Assistant",

provider=ModelProvider.MISTRAL,

model\_name="mistral-large-latest",

tier=AgentTier.RESEARCH\_ASSISTANT,

persona\_prompt="You are a Research Assistant specialized in analysis and synthesis of information from multiple sources."

))

self.register\_agent(Agent(

name="Cerebras\_Assistant",

provider=ModelProvider.CEREBRAS,

model\_name="llama-3.3-70b",

tier=AgentTier.RESEARCH\_ASSISTANT,

persona\_prompt="You are a fast Research Assistant optimized for quick responses and straightforward task execution."

))

**def** register\_agent(self, agent: Agent) -> **None**:

*"""Add an agent to the registry."""*

**if** agent.model\_instance **is** **None**:

print(f"Warning: Agent **{**agent.name**}** failed to initialize")

**return**

self.agents[agent.name] = agent

print(f"Registered agent: **{**agent.name**}** (**{**agent.provider.value**}**)")

**def** get\_agent\_registry(self) -> Dict[str, Agent]:

*"""Return the complete agent registry."""*

**return** self.agents.copy()

**def** get\_agent(self, name: str) -> Optional[Agent]:

*"""Retrieve a specific agent by name."""*

**return** self.agents.get(name)

**def** list\_agents\_by\_tier(self, tier: AgentTier) -> List[Agent]:

*"""Get all agents of a specific tier."""*

**return** [agent **for** agent **in** self.agents.values() **if** agent.tier == tier]

**File: Symposium/src/models/workflow.py**

*"""*

*LangGraph workflow management with provider-specific handling.*

*Contains workflow creation logic and execution state management.*

*"""*

**from** **dataclasses** **import** dataclass, field

**from** **typing** **import** Dict, List, Any, Optional

**from** **typing\_extensions** **import** Annotated

**from** **langgraph.graph.message** **import** add\_messages

**from** **langgraph.graph** **import** StateGraph, END

**from** **langchain\_core.messages** **import** BaseMessage, HumanMessage, AIMessage, ToolMessage

**from** **langchain\_core.tools** **import** Tool

**from** **.agent** **import** Agent, AgentTier

**from** **.tools.calculator** **import** create\_calculator\_tool

**from** **.tools.python\_executor** **import** create\_python\_executor\_tool

@dataclass

**class** **GraphState**:

*"""State object that flows through LangGraph workflows."""*

messages: Annotated[List[BaseMessage], add\_messages]

current\_agent: Optional[str] = **None**

task\_context: Optional[str] = **None**

metadata: Dict[str, Any] = field(default\_factory=dict)

**class** **WorkflowManager**:

*"""Manages LangGraph workflow creation and execution."""*

**def** create\_tool\_agent\_graph(self, agent: Agent, tools: Optional[List[Tool]] = **None**) -> StateGraph:

*"""Create a LangGraph workflow for a tool-using agent."""*

**if** tools **is** **None** **and** agent.tier == AgentTier.RESEARCH\_ASSISTANT:

tools = [create\_calculator\_tool(), create\_python\_executor\_tool()]

**elif** tools **is** **None**:

tools = []

*# Bind tools to the agent's model instance*

**if** tools:

model\_with\_tools = agent.model\_instance.bind\_tools(tools)

**else**:

model\_with\_tools = agent.model\_instance

**def** agent\_node(state: GraphState) -> Dict[str, Any]:

*"""Node function for the tool-using agent."""*

*# Construct system message with persona*

system\_message = HumanMessage(content=agent.persona\_prompt)

*# Get all messages including system context*

messages = [system\_message] + state.messages

print(f"--- Calling **{**agent.name**}** ---")

response = model\_with\_tools.invoke(messages)

print(f"--- Response from **{**agent.name**}**: **{**response.content[:100]**}**... ---")

**return** {

"messages": [response],

"current\_agent": agent.name

}

**def** should\_continue(state: GraphState) -> str:

*"""Determine if the agent should use tools or finish."""*

last\_message = state.messages[-1]

*# Check if the last message has tool calls*

**if** hasattr(last\_message, 'tool\_calls') **and** last\_message.tool\_calls:

**return** "tools"

**else**:

**return** "end"

**def** tool\_node(state: GraphState) -> Dict[str, Any]:

*"""Execute tool calls and return results."""*

last\_message = state.messages[-1]

tool\_calls = last\_message.tool\_calls

tool\_messages = []

current\_agent\_name = state.current\_agent

**for** tool\_call **in** tool\_calls:

tool\_name = tool\_call["name"]

tool\_call\_id = tool\_call["id"]

*# Handle different parameter key formats across providers*

parameter\_container = tool\_call.get("parameters") **or** tool\_call.get("args")

**if** **not** parameter\_container:

result = f"Error: No parameters found in tool call"

**else**:

tool\_to\_use = **None**

**for** tool **in** tools:

**if** tool.name == tool\_name:

tool\_to\_use = tool

**break**

**if** tool\_to\_use:

**try**:

**if** "\_\_arg1" **in** parameter\_container:

result = tool\_to\_use.func(parameter\_container["\_\_arg1"])

**else**:

result = tool\_to\_use.func(\*\*parameter\_container)

**except** **Exception** **as** e:

result = f"Error: **{**str(e)**}**"

**else**:

result = f"Error: Tool '**{**tool\_name**}**' not found"

*# Apply Cerebras-specific workaround*

**if** current\_agent\_name == "Cerebras\_Assistant":

*# Cerebras doesn't handle ToolMessage objects properly*

tool\_messages.append(

HumanMessage(content=f"Tool '**{**tool\_name**}**' returned the result: **{**result**}**", name=tool\_name)

)

**else**:

*# Standard ToolMessage format for other providers*

tool\_messages.append(

ToolMessage(content=str(result), tool\_call\_id=tool\_call\_id)

)

**return** {"messages": tool\_messages}

*# Build the graph*

workflow = StateGraph(GraphState)

workflow.add\_node("agent", agent\_node)

workflow.add\_node("tools", tool\_node)

workflow.set\_entry\_point("agent")

workflow.add\_conditional\_edges(

"agent",

should\_continue,

{"tools": "tools", "end": END}

)

*# Cerebras requires different workflow termination*

**if** agent.name == "Cerebras\_Assistant":

workflow.add\_edge("tools", END)

**else**:

workflow.add\_edge("tools", "agent")

**return** workflow.compile()

**File: Symposium/src/models/tools/calculator.py**

*"""*

*Basic calculator tool for fast arithmetic operations.*

*"""*

**from** **langchain\_core.tools** **import** Tool

**def** create\_calculator\_tool() -> Tool:

*"""Create a simple calculator tool for basic math."""*

**def** calculate(expression: str) -> str:

*"""Simple calculator that evaluates basic math expressions."""*

**try**:

*# Only allow basic math operations for security*

allowed\_chars = set('0123456789+-\*/.() ')

**if** **not** all(c **in** allowed\_chars **for** c **in** expression):

**return** "Error: Only basic math operations allowed"

result = eval(expression)

**return** f"Result: **{**result**}**"

**except** **Exception** **as** e:

**return** f"Calculation error: **{**str(e)**}**"

**return** Tool(

name="calculator",

description="Fast arithmetic for simple expressions (2+2, 127\*89). Use for basic math only.",

func=calculate

)

**File: Symposium/src/models/tools/python\_executor.py**

*"""*

*Whitelist-based Python execution tool.*

*Safe for basic calculations, falls back to Docker for complex operations.*

*"""*

**import** **ast**

**import** **sys**

**import** **io**

**from** **contextlib** **import** redirect\_stdout, redirect\_stderr

**from** **langchain\_core.tools** **import** Tool

*# Whitelist of safe operations*

SAFE\_IMPORTS = {

'math', 'decimal', 'statistics', 'random', 'json', 're', 'datetime'

}

SAFE\_BUILTINS = {

'print', 'len', 'range', 'enumerate', 'zip', 'sum', 'max', 'min',

'abs', 'round', 'sorted', 'reversed', 'any', 'all', 'int', 'float',

'str', 'list', 'dict', 'set', 'tuple', 'bool'

}

**def** \_analyze\_code\_safety(code: str) -> tuple[bool, str]:

*"""Analyze if code contains only whitelisted operations."""*

**try**:

tree = ast.parse(code)

**except** **SyntaxError** **as** e:

**return** **False**, f"Syntax error: **{**e**}**"

**for** node **in** ast.walk(tree):

**if** isinstance(node, ast.Import):

**for** alias **in** node.names:

**if** alias.name **not** **in** SAFE\_IMPORTS:

**return** **False**, f"Non-whitelisted import: **{**alias.name**}**"

**elif** isinstance(node, ast.ImportFrom):

**if** node.module **not** **in** SAFE\_IMPORTS:

**return** **False**, f"Non-whitelisted import: **{**node.module**}**"

**return** **True**, "Safe"

**def** create\_python\_executor\_tool() -> Tool:

*"""Create whitelist-based Python executor with Docker fallback option."""*

**def** execute\_python(code: str) -> str:

*"""Execute Python code with safety analysis."""*

*# Analyze code safety*

is\_safe, safety\_msg = \_analyze\_code\_safety(code)

**if** **not** is\_safe:

**return** f"Code rejected by whitelist: **{**safety\_msg**}\n**(Docker fallback not implemented yet)"

*# Execute safe code directly*

stdout\_capture = io.StringIO()

stderr\_capture = io.StringIO()

safe\_globals = {

'\_\_builtins\_\_': {name: getattr(\_\_builtins\_\_, name) **for** name **in** SAFE\_BUILTINS}

}

*# Add safe modules*

**try**:

**import** **math**, **decimal**, **statistics**, **random**, **json**, **re**, **datetime**

safe\_globals.update({

'math': math,

'decimal': decimal,

'Decimal': decimal.Decimal,

'statistics': statistics,

'random': random,

'json': json,

're': re,

'datetime': datetime,

})

**except** **ImportError**:

**pass**

**try**:

**with** redirect\_stdout(stdout\_capture), redirect\_stderr(stderr\_capture):

exec(code, safe\_globals)

stdout\_output = stdout\_capture.getvalue()

stderr\_output = stderr\_capture.getvalue()

**if** stderr\_output:

**return** f"Error: **{**stderr\_output.strip()**}**"

**elif** stdout\_output:

**return** stdout\_output.strip()

**else**:

**return** "Code executed successfully (no output)"

**except** **Exception** **as** e:

**return** f"Execution error: **{**str(e)**}**"

**return** Tool(

name="python\_executor",

description="Execute Python code with whitelist security. Supports math, decimal, statistics, datetime, json, re modules. Include print() statements to show results.",

func=execute\_python

)

**File: Symposium/src/controllers/orchestrator.py**

*"""*

*Main orchestration controller for agent selection and message routing.*

*Handles the core Director-Agent interaction logic.*

*"""*

**from** **typing** **import** Optional

**from** **langchain\_core.messages** **import** HumanMessage

**from** **models.symposium** **import** Symposium

**from** **models.agent** **import** Agent, AgentTier

**from** **models.workflow** **import** WorkflowManager, GraphState

**class** **Orchestrator**:

*"""Main controller for routing messages to appropriate agents."""*

**def** \_\_init\_\_(self):

self.symposium = Symposium()

self.workflow\_manager = WorkflowManager()

self.active\_workflows = {}

**def** quick\_ask(self, agent\_name: str, message: str, use\_tools: bool = **False**) -> str:

*"""Simple method to ask a single agent a question."""*

agent = self.symposium.get\_agent(agent\_name)

**if** **not** agent:

**return** f"Error: Agent **{**agent\_name**}** not found"

**try**:

**if** use\_tools **and** agent.tier == AgentTier.RESEARCH\_ASSISTANT:

*# Use the tool-enabled workflow*

graph = self.workflow\_manager.create\_tool\_agent\_graph(agent)

initial\_state = GraphState(

messages=[HumanMessage(content=message)],

current\_agent=agent\_name

)

final\_state = **None**

**for** event **in** graph.stream(initial\_state):

final\_state = event

**if** final\_state **and** "agent" **in** final\_state:

last\_message = final\_state["agent"]["messages"][-1]

**return** last\_message.content

**else**:

*# Handle case where graph terminates at tools node (Cerebras)*

last\_message = final\_state["tools"]["messages"][-1]

**return** last\_message.content

**else**:

*# Direct model call*

response = agent.model\_instance.invoke([HumanMessage(content=message)])

**return** response.content

**except** **Exception** **as** e:

**return** f"Error calling **{**agent\_name**}**: **{**str(e)**}**"

**def** get\_agent\_registry(self):

*"""Expose the agent registry for external access."""*

**return** self.symposium.get\_agent\_registry()

**def** get\_agent(self, name: str) -> Optional[Agent]:

*"""Get a specific agent."""*

**return** self.symposium.get\_agent(name)

**def** list\_agents\_by\_tier(self, tier: AgentTier):

*"""List agents by capability tier."""*

**return** self.symposium.list\_agents\_by\_tier(tier)

**File: Symposium/src/views/cli.py**

*"""*

*Command-line interface for testing and interacting with the symposium.*

*"""*

**from** **controllers.orchestrator** **import** Orchestrator

**from** **models.agent** **import** AgentTier

**class** **SymposiumCLI**:

*"""Simple CLI interface for testing symposium functionality."""*

**def** \_\_init\_\_(self):

self.orchestrator = Orchestrator()

**def** run\_demo(self):

*"""Run the standard demo sequence from the original manager."""*

print("--- Initializing Symposium ---")

print("**\n**--- Agent Registry ---")

**for** name, agent **in** self.orchestrator.get\_agent\_registry().items():

status = "✓" **if** agent.model\_instance **else** "✗"

print(f"**{**status**}** **{**name**}** (**{**agent.provider.value**}**) - **{**agent.tier.value**}**")

print("**\n**--- Testing Simple Agent Call ---")

response = self.orchestrator.quick\_ask("Claude", "What is 2+2? Answer in exactly 5 words.")

print(f"Response: **{**response**}**")

print("**\n**--- Testing Tool-Using Agents ---")

research\_assistants = ["Groq\_Assistant", "Mistral\_Assistant", "Cerebras\_Assistant"]

test\_calculation = "Calculate 127 \* 89 + 456"

**for** assistant\_name **in** research\_assistants:

print(f"**\n**--- Testing **{**assistant\_name**}** ---")

response = self.orchestrator.quick\_ask(assistant\_name, test\_calculation, use\_tools=**True**)

print(f"**{**assistant\_name**}**: **{**response**}**")

print("**\n**--- Symposium Ready ---")

**if** \_\_name\_\_ == "\_\_main\_\_":

cli = SymposiumCLI()

cli.run\_demo()

**File: Symposium/src/symposium\_manager\_v2.py**

*#!/usr/bin/env python3*

*"""*

*Multi-LLM Symposium Manager v2.0*

*Clean entry point using modular MVC architecture.*

*"""*

**from** **views.cli** **import** SymposiumCLI

**from** **controllers.orchestrator** **import** Orchestrator

**def** main():

*"""Main entry point for the symposium system."""*

*# For direct access to orchestrator*

*# orchestrator = Orchestrator()*

*# For CLI demo*

cli = SymposiumCLI()

cli.run\_demo()

**if** \_\_name\_\_ == "\_\_main\_\_":

main()