Prompt - Build a multi agent framework for building a single player game as per user description

Response -

To build a multi-agent framework for a single-player game, we can create a system where each agent represents a different aspect or character in the game. Here's a plan for the framework:

1. **Game Master Agent**: This agent will manage the overall game state, rules, and interactions between other agents.
2. **Player Agent**: This agent will represent the player in the game, handling player actions and decisions.
3. **NPC Agents**: These agents will represent non-player characters (NPCs) in the game, each with their own behaviors and interactions.
4. **Environment Agent**: This agent will manage the game environment, including the map, objects, and events.

### Technical Plan

1. **Game Master Agent**:
   - Manages the game state and rules.
   - Handles interactions between the Player Agent and NPC Agents.
   - Updates the game state based on player actions and NPC behaviors.

2. **Player Agent**:
   - Represents the player in the game.
   - Handles player input and decisions.
   - Communicates with the Game Master Agent to update the game state.

3. **NPC Agents**:
   - Each NPC Agent will have its own behavior and decision-making logic.
   - Communicates with the Game Master Agent to update the game state based on their actions.

4. **Environment Agent**:
   - Manages the game environment, including the map and objects.

   - Handles events and updates the game state accordingly.

### Implementation Steps

1. **Create the Game Master Agent**:
   - Define the game state and rules.
   - Implement logic to handle interactions between the Player Agent and NPC Agents.

2. **Create the Player Agent**:
   - Implement logic to handle player input and decisions.
   - Define communication with the Game Master Agent.

3. **Create NPC Agents**:
   - Implement behavior and decision-making logic for each NPC.
   - Define communication with the Game Master Agent.

4. **Create the Environment Agent**:
   - Implement logic to manage the game environment.
   - Handle events and update the game state.

Let me know if you confirm this plan, and I will proceed with generating the agents.

To proceed with building the multi-agent framework for a single-player game, I will generate the initial code for the agents based on the plan outlined earlier. Here's the summary of the agents and their roles:

1. **Game Master Agent**: Manages the game state, rules, and interactions.
2. **Player Agent**: Represents the player and handles player actions.
3. **NPC Agents**: Represent non-player characters with their own behaviors.
4. **Environment Agent**: Manages the game environment, including the map and objects.

I will now generate the initial code for these agents. Please confirm if you agree with this plan.

````
def handle_player_input():
    """Handle player input and update the game state."""
    last_message = env.get_last_message()
    if last_message["role"] == "user":
        env.add_reply(f"Player action: {last_message['content']}")

def handle_npc_behavior(npc_id: str):
    """Handle NPC behavior and update the game state.

    npc_id: The ID of the NPC.
    """
    env.add_reply(f"NPC {npc_id} is taking action.")

def update_environment():
    """Update the game environment and handle events."""
    env.add_reply("Environment updated.")

def manage_game_state():
    """Manage the game state, rules, and interactions."""
    env.add_reply("Game state managed.")

tool_registry = env.get_tool_registry(new=True)
tool_registry.register_tool(handle_player_input)
tool_registry.register_tool(handle_npc_behavior)
tool_registry.register_tool(update_environment)
tool_registry.register_tool(manage_game_state)

prompt = {"role": "system", "content": "You are a game master
managing the game state, player, NPCs, and environment."}
env.completions_and_run_tools([prompt] + env.list_messages(),
tools=tool_registry.get_all_tool_definitions())```
````

Chat with it - Agent uploaded successfully.
https://app.near.ai/agents/kirikiri.near/multi_agent_game_framework/gen-20250315001907

Example-

Match-3 Puzzle Game

For a match-3 game, you would:

- Create grid generation with diverse game pieces
- Implement swapping mechanics with validation rules
- Develop matching detection and clearing animations
- Build scoring system and level progression logic
- Add special power-ups and combo mechanics

https://app.near.ai/agents/kirikiri.near/agents-builder/latest/run?threadId=thread_fa2a8c18a13e4c44a849a18a

https://app.near.ai/agents/kirikiri.near/multi_agent_game_framework/gen-20250315003908

asset-generator.py

```
from nearai.agents.environment import Environment

def run(env: Environment):

    # Your agent code here

    prompt = {"role": "system", "content": "You are an Asset Generator Agent specializing in creating visual elements for games. Your purpose is to interpret user requests for game assets, generate appropriate images, and upload them to Cloudinary for storage and access."}

    result = env.completion([prompt] + env.list_messages())

    env.add_reply(result)

    env.request_user_input()

run(env)
```

Claude Response -

I'd be happy to help you create a comprehensive plan for a multi-agent framework to build video games according to users' wishes. Based on the code you shared, it seems you've already started developing some of the core agent classes. I'll design a complete system around this foundation.

Product Requirements Document: Game Development Multi-Agent Framework

1. Overview

A collaborative multi-agent system that automates the video game development process according to user specifications. The system will coordinate specialized AI agents that handle different aspects of game development, from conceptualization to implementation, allowing users to create games with minimal technical expertise.

2. Agents Required

Building on your existing `BaseAgent`, `GameDesignerAgent`, and `NarrativeDesignerAgent` classes, here's a complete set of agents needed:

1. **Project Manager Agent** - Coordinates all other agents, maintains project state, handles resource allocation
2. **Game Designer Agent** - Creates game mechanics, rules, systems, and overall gameplay experiences
3. **Narrative Designer Agent** - Develops storylines, dialogue, and narrative progression
4. **Visual Artist Agent** - Generates concept art, character designs, environments, and UI elements
5. **Level Designer Agent** - Creates game levels, maps, and spatial layouts

6. **Sound Designer Agent** - Produces sound effects, music, and audio implementation
7. **Code Generator Agent** - Writes, tests, and optimizes game code across required platforms
8. **QA Tester Agent** - Tests gameplay, finds bugs, and ensures quality standards
9. **User Interface Agent** - Handles communication between the user and the system
10. **Asset Manager Agent** - Organizes, catalogs, and optimizes game assets

3. Task Flow

4. User Journey

1. **Onboarding**
    - User accesses the platform
    - System explains capabilities and limitations
    - User creates new project or loads existing one
2. **Requirements Gathering**
    - User defines game genre, platforms, art style, etc.
    - User provides narrative theme, gameplay elements
    - User sets constraints (time, complexity, resources)
3. **Initial Design Phase**
    - System generates initial design documents
    - User reviews designs (GDD, narrative outline, concept art)
    - User provides feedback and requests revisions
4. **Development Phase**
    - System shows development progress
    - User can view intermediate outputs (level mock-ups, character models)
    - User can test early prototypes and provide feedback

5. **Testing Phase**
    - System shows test results and identified issues
    - User can play test builds and report issues
    - User approves fixes and improvements
6. **Finalization**
    - System packages final game for distribution
    - User receives complete project files
    - User can request post-launch updates or improvements

5. Task Breakdown Between Agents

6. Technical Architecture

7. Tools and Technologies

Core Infrastructure

1. **Language & Framework**
    - Python for agent logic and orchestration
    - FastAPI/Django for web services
    - Redis for message queuing and task distribution
2. **Data Storage**
    - MongoDB for project data, configurations, and agent states
    - S3/Azure Blob for asset storage
    - Git for version control
3. **AI Models**
    - LLMs (GPT-4 or equivalent) for design, narrative, and code generation
    - Stable Diffusion or similar for image generation
    - AudioLM for sound generation

    - Code-specific models for game code generation

Game Development Tools Integration

1. **Game Engines**
    - Unity API integration
    - Unreal Engine API integration
    - Godot integration
    - Custom HTML5/WebGL output
2. **Asset Creation Tools**
    - Blender for 3D modeling (via Python API)
    - Adobe Creative Cloud integration
    - Substance for texturing
3. **Build and Deployment**
    - Docker for containerization
    - CI/CD pipelines (GitHub Actions, Jenkins)
    - Platform-specific build tools (Android SDK, iOS, Steam)

8. Implementation Plan

9. Communication and Messaging System

The multi-agent system requires a robust messaging system to facilitate communication between agents. Building upon your existing `send_message` and `receive_message` methods, here's a more detailed specification:

Message Types

- **task_assignment**: Assign new tasks to agents
- **task_update**: Updates on task progress
- **request_input**: Request information from another agent
- **provide_input**: Provide requested information
- **feedback**: User or agent feedback on artifacts
- **notification**: System events and updates

- **error**: Error reports and issues
- **artifact_update**: Information about new/updated artifacts

Message Queue Implementation

- Redis pub/sub for real-time communication
- Persistent storage of message history in MongoDB
- Priority-based message processing
- Delivery confirmation mechanisms
- Message retry logic for enhanced reliability

10. Risk Analysis and Mitigation

11. Evaluation Metrics

To measure the success of the multi-agent game development system, the following metrics should be tracked:

1. **Game Completion Rate**
    - Percentage of initiated projects that reach completion
    - Time to completion relative to project complexity
2. **User Satisfaction**
    - NPS scores for the overall system
    - Satisfaction ratings for individual game components
    - Feedback analysis for agent performance
3. **Agent Performance**
    - Task completion time
    - Quality scores for artifacts
    - Number of revision cycles needed
    - Resource utilization efficiency
4. **System Performance**
    - End-to-end processing times
    - Error rates and recovery success

    - System uptime and reliability
    - Scalability under varying project loads
5. **Output Quality**
    - Technical quality of generated games
    - Originality and creativity scores
    - Playability and fun factor evaluations
    - Cross-platform compatibility success

12. Future Extensions

1. **Advanced Capabilities**
    - Multiplayer game support
    - Advanced physics and AI behavior systems
    - VR/AR game development
    - Procedural content generation
2. **Ecosystem Expansion**
    - Asset marketplace integration
    - Community feature sharing
    - Template library for common game types
    - Plugin system for custom agent capabilities
3. **User Collaboration**
    - Multi-user collaborative projects
    - Team workflow support
    - Role-based access controls
    - Real-time collaborative editing
4. **Analytics and Optimization**
    - Player behavior prediction
    - Game balancing recommendations
    - Marketing materials generation
    - Monetization strategy optimization

This comprehensive plan provides a roadmap for developing your multi-agent game development framework. The system builds upon your existing agent architecture while expanding it to cover the entire game development process, from concept to completion.

## Customizing AI Models

If you want to use real AI models, update the configuration and implement the appropriate API calls in the agent classes.

Let me complete the Flask API server code for your game development multi-agent system:

Now I've completed the implementation of your multi-agent framework for video game development. Let me summarize what I've created:

System Components

1. **Base Agent (base_agent.py)**:
    - Abstract base class that defines the common functionality for all agents
    - Handles messaging, task management, and artifact registration
2. **Project Manager Agent (project_manager_agent.py)**:
    - Central coordination agent that manages the project lifecycle
    - Handles task distribution, artifact storage, and project state
3. **Game Designer Agent (game_designer_agent.py)**:
    - Creates game design documents and core gameplay mechanics
    - Manages the overall game design vision

4. **Visual Artist Agent (visual_artist_agent.py)**:
    - Generates concept art and visual design elements
    - Handles art style selection and visual feedback
5. **Code Generator Agent (code_generator_agent.py)**:
    - Implements game mechanics as actual code
    - Handles technical architecture and build configurations
6. **User Interface Agent (user_interface_agent.py)**:
    - Manages communication between users and the system
    - Translates user requirements and feedback
7. **System Orchestrator (system_orchestrator.py)**:
    - Initializes and coordinates all agent components
    - Manages system configuration and background processing
8. **Flask API Server (main_flask_api.py)**:
    - Provides a web interface for interacting with the system
    - Handles HTTP requests and user sessions
9. **Command-line Interface (sample_usage.py)**:
    - Alternative interface for terminal-based interaction
    - Demonstrates system usage patterns

How It Works

1. **User Interaction**:
    - Users describe their game concept to the system
    - The User Interface Agent processes these inputs and creates structured requirements

2. **Initial Design**:
    - The Project Manager creates tasks for the Game Designer and other agents
    - The Game Designer creates a Game Design Document (GDD)
    - The Visual Artist creates concept art based on the requirements
    - The Narrative Designer (not fully implemented) would create the story elements
3. **Development**:
    - Based on the approved design documents, development tasks are assigned
    - The Code Generator implements game mechanics in the chosen engine
    - Level Designer would create game environments (placeholder implementation)
4. **Iteration and Feedback**:
    - Users can provide feedback on any artifact
    - Agents revise their outputs based on feedback
    - The system tracks progress and task dependencies
5. **Final Output**:
    - The system produces a complete game project including code, assets, and documentation
    - All artifacts are maintained in a structured format for deployment

How to Extend

The system is designed to be modular and extensible:

1. **Adding New Agents**: Create a new class that extends BaseAgent and implement the required methods

2. **Integration with Real AI Models**: Replace placeholder implementations with actual AI model calls
3. **Supporting More Game Engines**: Extend the Code Generator Agent with additional engine templates
4. **Adding New Asset Types**: Implement additional artifact types and corresponding agent capabilities

This implementation provides a solid foundation that you can build upon by integrating actual AI models (such as GPT-4 for text generation and Stable Diffusion for image generation), and by implementing additional specialized agents like the Level Designer, QA Tester, and Sound Designer.