Mixture of Expert Agents

This YAML is a little rough as it's made from multiple copy/paste of different pseudocodes. I'd like you to work through it to streamline a process for building a Life Simulation that can train AutoGen Agents in Simulated Environments.

Mixture of Experts, Function Callers & AI Agents

The document you've provided appears to be a detailed framework for a Life Simulation focusing on AI Agents. It outlines various components of the simulation, including the setup of the conversation, descriptions of life stage agents, interactions, capabilities, and parameters for simulation. Here's a streamlined process to build a Life Simulation for training AutoGen Agents in simulated environments, based on your document:

Initial Setup and User Interaction:

- Begin with a greeting and ask the user for key details: Age, Time Available, Learning Hours, Daily Routines Percentage, Life Experiences Percentage, Efficiency Factor, Complexity Multiplier, Model Throughput, Simulation Duration, and Simulation Focus.
- Collect and confirm these details from the user.

Agent Definitions:

- Define agents for each life stage (Baby, Toddler, Child, Teenager, Adult) with specific attributes and behaviors (e.g., sensory exploration for babies, language acquisition for toddlers).
- Detail the capabilities of each agent at different life stages, such as cognitive functions, social rules learning, and risk assessment.

Simulation Parameters and Framework:

- Set up the simulation parameters including token allocation, efficiency factor, complexity multiplier, and overall throughput.
- Develop a simulation output formula that calculates cumulative token usage and interactions over a specified time period.
- Implement the life model with core variables like age, time, learning, daily routines, and life experiences.

Dynamic Simulation Components:

- Include time allocation functions for different age groups.
- Outline daily activities, developmental stages, and external influences that affect the agents.
- Integrate cumulative experiences over time and consider cyclical and seasonal adjustments.

Base and Expanded Formulas:

- Use base formulas to calculate the total number of thoughts and words generated in a minute.
- Expand these formulas by applying multipliers for different states (casual, normal, urgent, etc.).

Life Stage Activities and External Influences:

- Categorize activities (Cognitive, Physical, Social Interactions, etc.) relevant to each life stage.
- Adjust for external influences like cultural factors, family environment, and global events.

Sub-models and Overall Simulation:

- Develop sub-models for cognitive development, social interaction, and physical activity.
- Combine these sub-models, applying weightings based on the life stage for an overall simulation.

- Ensure the model accounts for dynamic human development, empirical data, and individual variability.

Simulation Implementation:

- Use a suitable programming language or simulation software to implement the model.
- Run simulations based on user inputs, adjusting dynamically as per the model's structure and parameters.

Feedback and Iteration:

- After running simulations, collect feedback on agent behavior and outcomes.
- Iterate on the model to enhance realism, accuracy, and agent learning capabilities.

This process provides a comprehensive and adaptable framework for a life simulation that can effectively train AutoGen Agents in various simulated environments, reflecting different life stages and scenarios.

AU

You

Great! Now please write a comprehensive pseudocode for generating a formula for a lifetime so that we can train experts in every field of civilian lifestyle, as well as academic and corporate excellence. For this response, just focus on Section 1: Initial Setup and User Interaction.

Mixture of Experts, Function Callers & AI Agents

Certainly! To create a pseudocode for generating a formula for a lifetime, focusing on the Initial Setup and User Interaction (Section 1), we'll need to establish a structured approach. This section will involve collecting initial data from the user, which will be used to shape the subsequent stages of the life simulation. Here's the pseudocode:

// Pseudocode for Section 1: Initial Setup and User Interaction

```
// Life Simulation for Training Experts in Various Fields


Procedure InitializeLifeSimulation():

    Display "Welcome to the Life Simulation!"


    // Collect user inputs

    userAge <- GetUserInput("Enter your age: ")

    timeAvailable <- GetUserInput("Enter the time available for simulation (in hours): ")

    learningHours <- GetUserInput("Enter the number of hours you can dedicate to learning daily: ")

    dailyRoutinesPercentage <- GetUserInput("Enter the percentage of time spent on daily routines: ")

    lifeExperiencesPercentage <- GetUserInput("Enter the percentage of time spent on life experiences: ")
```

```
    efficiencyFactor <- GetUserInput("Enter your efficiency factor (1-10): ")
    complexityMultiplier <- GetUserInput("Enter the complexity multiplier for simulation scenarios: ")
    modelThroughput <- GetUserInput("Enter the desired model throughput for simulation: ")
    simulationDuration <- GetUserInput("Enter the desired duration of the simulation (in years): ")
    simulationFocus <- GetUserInput("Enter the focus of the simulation (e.g., academic, corporate, lifestyle): ")

    // Validate and confirm user inputs
    ConfirmUserInputs(userAge, timeAvailable, learningHours, dailyRoutinesPercentage,
                      lifeExperiencesPercentage, efficiencyFactor, complexityMultiplier,
                      modelThroughput, simulationDuration, simulationFocus)

    // Proceed to the next step or allow user to modify inputs
    If GetUserConfirmation("Would you like to proceed with these settings? (yes/no): ") = 'no' Then
        InitializeLifeSimulation() // Restart the process if user opts to change inputs
    EndIf

    // Store user inputs for further processing in subsequent sections
    StoreUserInputs(userAge, timeAvailable, learningHours, dailyRoutinesPercentage,
                    lifeExperiencesPercentage, efficiencyFactor, complexityMultiplier,
                    modelThroughput, simulationDuration, simulationFocus)

    // Indicate completion of initial setup
    Display "Initial setup complete. Proceeding to next stage of the simulation."

EndProcedure

// Helper function to display a message and get user input
Function GetUserInput(promptMessage: String) -> String:
```

```
    Display promptMessage

    userInput <- ReadUserInput()

    Return userInput

EndFunction


// Helper function to confirm and display user inputs

Procedure ConfirmUserInputs(*args):

    Display "Please confirm your inputs:"

    For each input in args:

        Display input

    EndFor

EndProcedure


// Helper function to get user confirmation

Function GetUserConfirmation(promptMessage: String) -> String:

    Display promptMessage

    userInput <- ReadUserInput()

    Return userInput

EndFunction


// Helper function to store user inputs for simulation

Procedure StoreUserInputs(*args):

    // Implementation to store the user inputs in the simulation's data structure

    // This can be a database, file, or in-memory data structure depending on the system design

EndProcedure


// Main entry point

Begin

    InitializeLifeSimulation()
```

End

This pseudocode outlines the initial setup and user interaction process for a comprehensive life simulation system. It emphasizes user input collection, validation, confirmation, and storage, setting a solid foundation for the simulation's subsequent stages.

AU

You

Great! Now Section 2: Agent Definitions

Mixture of Experts, Function Callers & AI Agents

Certainly! For Section 2: Agent Definitions, the pseudocode will focus on defining various life stage agents (like Baby, Toddler, Child, Teenager, Adult), each with specific attributes, behaviors, and capabilities. Here's a comprehensive pseudocode structure for this section:

// Pseudocode for Section 2: Agent Definitions

```
// Life Simulation for Training Experts in Various Fields


Procedure DefineAgents():

    // Define agent characteristics for each life stage

    babyAgent <- DefineBabyAgent()

    toddlerAgent <- DefineToddlerAgent()

    childAgent <- DefineChildAgent()

    teenagerAgent <- DefineTeenagerAgent()

    adultAgent <- DefineAdultAgent()


    // Store agent definitions for further processing

    StoreAgentDefinitions(babyAgent, toddlerAgent, childAgent, teenagerAgent, adultAgent)


    // Indicate completion of agent definitions

    Display "Agent definitions for all life stages have been completed."
```

```
EndProcedure

// Define the characteristics of a Baby Agent
Function DefineBabyAgent() -> Agent:
    babyAgent <- CreateNewAgent()
    babyAgent.AddAttribute("Sensory Exploration", True)
    babyAgent.AddAttribute("Basic Communication Skills", False)
    babyAgent.AddAttribute("Physical Development", "Basic")
    babyAgent.AddBehavior("Exploring Surroundings", "High")
    babyAgent.AddBehavior("Forming Basic Bonds", "Moderate")
    babyAgent.SetStage("Baby")
    Return babyAgent
EndFunction

// Define the characteristics of a Toddler Agent
Function DefineToddlerAgent() -> Agent:
    toddlerAgent <- CreateNewAgent()
    toddlerAgent.AddAttribute("Language Acquisition", True)
    toddlerAgent.AddAttribute("Social Interaction", "Basic")
    toddlerAgent.AddAttribute("Cognitive Skills", "Developing")
    toddlerAgent.AddBehavior("Imitation Learning", "High")
    toddlerAgent.AddBehavior("Problem Solving", "Moderate")
    toddlerAgent.SetStage("Toddler")
    Return toddlerAgent
EndFunction

// Define the characteristics of a Child Agent
Function DefineChildAgent() -> Agent:
    childAgent <- CreateNewAgent()
```

```
    childAgent.AddAttribute("Advanced Cognitive Skills", True)
    childAgent.AddAttribute("Social Awareness", "Moderate")
    childAgent.AddAttribute("Physical Coordination", "Developed")
    childAgent.AddBehavior("Collaborative Play", "High")
    childAgent.AddBehavior("Academic Learning", "Moderate")
    childAgent.SetStage("Child")
    Return childAgent
EndFunction

// Define the characteristics of a Teenager Agent
Function DefineTeenagerAgent() -> Agent:
    teenagerAgent <- CreateNewAgent()
    teenagerAgent.AddAttribute("Critical Thinking", True)
    teenagerAgent.AddAttribute("Emotional Intelligence", "Developing")
    teenagerAgent.AddAttribute("Risk Assessment", "Moderate")
    teenagerAgent.AddBehavior("Peer Interaction", "High")
    teenagerAgent.AddBehavior("Self-Identity Exploration", "High")
    teenagerAgent.SetStage("Teenager")
    Return teenagerAgent
EndFunction

// Define the characteristics of an Adult Agent
Function DefineAdultAgent() -> Agent:
    adultAgent <- CreateNewAgent()
    adultAgent.AddAttribute("Complex Problem Solving", True)
    adultAgent.AddAttribute("Advanced Social Skills", "High")
    adultAgent.AddAttribute("Professional Expertise", "Varies")
    adultAgent.AddBehavior("Career Development", "High")
    adultAgent.AddBehavior("Life Management", "High")
```

```
    adultAgent.SetStage("Adult")
    Return adultAgent
EndFunction

// Helper function to create a new agent
Function CreateNewAgent() -> Agent:
    agent <- New Agent()
    Return agent
EndFunction

// Helper function to store agent definitions
Procedure StoreAgentDefinitions(*agents):
    // Implementation to store the agent definitions in the simulation's data structure
    // This can be a database, file, or in-memory data structure depending on the system design
EndProcedure

// Main entry point
Begin
    DefineAgents()
End
```

This pseudocode establishes a clear structure for defining agents at different life stages, each with unique characteristics and behaviors appropriate for their developmental phase. These definitions are crucial for simulating realistic and nuanced agent interactions in the subsequent stages of the life simulation.

AU

You

Awesome! Now Section 3: Simulation Parameters and Framework

Mixture of Experts, Function Callers & AI Agents

For Section 3, focusing on "Simulation Parameters and Framework," the pseudocode will outline the setup of the simulation environment, including defining parameters and establishing the framework for the simulation. This will involve setting up the simulation's basic rules, variables, and operational logic:

// Pseudocode for Section 3: Simulation Parameters and Framework

```
// Life Simulation for Training Experts in Various Fields

Procedure SetupSimulationParametersAndFramework():
   // Define key simulation parameters
   simulationParameters <- DefineSimulationParameters()

   // Set up the simulation framework based on the parameters
   simulationFramework <- InitializeSimulationFramework(simulationParameters)

   // Store simulation setup for further processing
   StoreSimulationSetup(simulationFramework)

   // Indicate completion of simulation setup
   Display "Simulation parameters and framework have been successfully set up."

EndProcedure

// Define key parameters for the simulation
Function DefineSimulationParameters() -> Parameters:
   parameters <- CreateNewParameters()

   // Set core simulation parameters
   parameters.Add("Token Allocation", CalculateTokenAllocation())
   parameters.Add("Efficiency Factor", RetrieveUserInput("Efficiency Factor"))
   parameters.Add("Complexity Multiplier", RetrieveUserInput("Complexity Multiplier"))
```

```
    parameters.Add("Model Throughput", RetrieveUserInput("Model Throughput"))
    parameters.Add("Simulation Duration", RetrieveUserInput("Simulation Duration"))

    Return parameters
EndFunction

// Initialize the simulation framework based on provided parameters
Function InitializeSimulationFramework(parameters: Parameters) -> Framework:
    framework <- New SimulationFramework()

    // Apply the parameters to the simulation framework
    framework.SetTokenAllocation(parameters.Get("Token Allocation"))
    framework.SetEfficiencyFactor(parameters.Get("Efficiency Factor"))
    framework.SetComplexityMultiplier(parameters.Get("Complexity Multiplier"))
    framework.SetModelThroughput(parameters.Get("Model Throughput"))
    framework.SetDuration(parameters.Get("Simulation Duration"))

    // Initialize additional components of the framework as needed
    InitializeAdditionalComponents(framework)

    Return framework
EndFunction

// Helper function to create a new parameters object
Function CreateNewParameters() -> Parameters:
    parameters <- New Parameters()
    Return parameters
EndFunction
```

```
// Helper function to calculate token allocation
Function CalculateTokenAllocation() -> Number:
    // Implementation to calculate token allocation based on user inputs and system logic
    // For example, it can be based on user age, available time, and simulation focus
    // Returns a numerical value representing token allocation
EndFunction

// Helper function to retrieve user input for a specific parameter
Function RetrieveUserInput(parameterName: String) -> String:
    // Implementation to retrieve stored user inputs from Section 1
    // Returns the value of the specified parameter
EndFunction

// Helper function to initialize additional components of the simulation framework
Procedure InitializeAdditionalComponents(framework: SimulationFramework):
    // Implementation to add extra components like environment settings, agent behaviors, etc.
    // This can be based on the specific requirements of the simulation
EndProcedure

// Helper function to store the simulation setup
Procedure StoreSimulationSetup(framework: SimulationFramework):
    // Implementation to store the framework setup in the simulation's data structure
    // This can be a database, file, or in-memory data structure depending on the system design
EndProcedure

// Main entry point
Begin
    SetupSimulationParametersAndFramework()
End
```

In this section, the pseudocode establishes the foundational setup for the life simulation, focusing on parameter definition and framework initialization. It ensures that the simulation is configured according to the user's inputs and the system's requirements, setting the stage for detailed simulation activities in the subsequent sections.

AU

You

Super! Next Section 4: Dynamic Simulation Components

Mixture of Experts, Function Callers & AI Agents

In Section 4, focusing on "Dynamic Simulation Components," the pseudocode will detail the integration of dynamic elements into the simulation. This includes incorporating time allocation functions, daily activities, developmental stages, external influences, and cyclical/seasonal adjustments. These components will make the simulation more realistic and adaptable to various scenarios:

// Pseudocode for Section 4: Dynamic Simulation Components

```
// Life Simulation for Training Experts in Various Fields


Procedure IntegrateDynamicComponents():

   // Retrieve simulation framework from stored data

   simulationFramework <- RetrieveSimulationFramework()


   // Integrate dynamic components into the simulation

   IntegrateTimeAllocationFunctions(simulationFramework)

   IntegrateDailyActivities(simulationFramework)

   IntegrateDevelopmentalStages(simulationFramework)

   IntegrateExternalInfluences(simulationFramework)

   ApplyCyclicalAndSeasonalAdjustments(simulationFramework)


   // Update the simulation framework with dynamic components

   UpdateSimulationFramework(simulationFramework)


   // Indicate completion of dynamic component integration
```

```
    Display "Dynamic components have been successfully integrated into the simulation."

EndProcedure

// Integrate time allocation functions based on agent's life stage
Procedure IntegrateTimeAllocationFunctions(framework: SimulationFramework):
    // Implementation to define how time is allocated for different activities
    // based on the agent's current life stage
EndProcedure

// Integrate daily activities into the simulation
Procedure IntegrateDailyActivities(framework: SimulationFramework):
    // Implementation to define a range of daily activities
    // that agents will engage in, depending on their life stage and other factors
EndProcedure

// Integrate developmental stages into the simulation
Procedure IntegrateDevelopmentalStages(framework: SimulationFramework):
    // Implementation to define the key developmental stages for the agents
    // and the transitions between these stages
EndProcedure

// Integrate external influences into the simulation
Procedure IntegrateExternalInfluences(framework: SimulationFramework):
    // Implementation to define how external factors like cultural,
    // environmental, and global events impact the agents
EndProcedure

// Apply cyclical and seasonal adjustments to the simulation
```

```
Procedure ApplyCyclicalAndSeasonalAdjustments(framework: SimulationFramework):
    // Implementation to adjust the simulation based on cyclical (e.g., daily routines)
    // and seasonal (e.g., changes in weather, holidays) factors
EndProcedure

// Update the simulation framework with integrated dynamic components
Procedure UpdateSimulationFramework(framework: SimulationFramework):
    // Implementation to update the framework with the newly integrated components
    // This may involve modifying the framework's internal data structures or algorithms
EndProcedure

// Helper function to retrieve the simulation framework
Function RetrieveSimulationFramework() -> SimulationFramework:
    // Implementation to retrieve the stored simulation framework from the system's data structure
    // Returns the simulation framework object
EndProcedure

// Main entry point
Begin
    IntegrateDynamicComponents()
End
```

This section of the pseudocode focuses on making the simulation adaptable and responsive to various factors affecting an individual's life journey. It adds complexity and depth to the simulation, allowing it to mimic real-life dynamics more closely.

AU

You

Thanks. Now Section 5: Base and Expanded Formulas

Mixture of Experts, Function Callers & AI Agents

For Section 5, "Base and Expanded Formulas," the pseudocode will focus on developing formulas to calculate critical metrics within the simulation. These formulas are essential for quantifying aspects like thoughts, words, and actions, and they may vary based on different states or scenarios. Here's the pseudocode:

// Pseudocode for Section 5: Base and Expanded Formulas

```
// Life Simulation for Training Experts in Various Fields


Procedure DevelopBaseAndExpandedFormulas():

   // Develop base formulas for calculating key metrics in the simulation

   baseThoughtsFormula <- CreateBaseThoughtsFormula()

   baseWordsFormula <- CreateBaseWordsFormula()


   // Develop expanded formulas for different states (e.g., casual, normal, urgent)

   expandedThoughtsFormula <- ExpandFormula(baseThoughtsFormula)

   expandedWordsFormula <- ExpandFormula(baseWordsFormula)


   // Store the developed formulas for further use in the simulation

   StoreFormulas(baseThoughtsFormula, baseWordsFormula, expandedThoughtsFormula, expandedWordsFormula)


   // Indicate completion of formula development

   Display "Base and expanded formulas have been successfully developed and stored."


EndProcedure


// Create the base formula for calculating thoughts per minute

Function CreateBaseThoughtsFormula() -> Formula:

   baseThoughtsFormula <- New Formula()

   baseThoughtsFormula.Define("ThoughtsPerMinute", "<formula_logic_here>")

   Return baseThoughtsFormula
```

```
EndFunction

// Create the base formula for calculating words spoken per minute
Function CreateBaseWordsFormula() -> Formula:
    baseWordsFormula <- New Formula()
    baseWordsFormula.Define("WordsPerMinute", "<formula_logic_here>")
    Return baseWordsFormula
EndFunction

// Expand the base formula to account for different states or scenarios
Function ExpandFormula(baseFormula: Formula) -> Formula:
    expandedFormula <- New Formula()
    expandedFormula.Inherit(baseFormula)

    // Add adjustments for different states (e.g., casual, normal, urgent)
    expandedFormula.Modify("<state>", "<adjusted_formula_logic_here>")

    Return expandedFormula
EndFunction

// Helper function to store the developed formulas
Procedure StoreFormulas(*formulas):
    // Implementation to store the formulas in the simulation's data structure
    // This can be a database, file, or in-memory data structure depending on the system design
EndProcedure

// Main entry point
Begin
    DevelopBaseAndExpandedFormulas()
```

End

In this section, the pseudocode outlines the creation of basic and expanded formulas, vital for the simulation's operational logic. These formulas are instrumental in quantifying and simulating the cognitive and verbal processes of the agents within the simulation. The formula logic (indicated by **`<formula_logic_here>`** and **`<adjusted_formula_logic_here>`**) should be developed based on the specific requirements and scientific understanding of the processes being simulated.

AU

You

And now let make Section 6: Sub-models and Overall Simulation

Mixture of Experts, Function Callers & AI Agents

For Section 6, "Sub-models and Overall Simulation," the pseudocode will describe the integration of various sub-models into the larger simulation framework and the coordination between these sub-models to create a comprehensive simulation experience. Sub-models might include aspects like cognitive development, social interaction, physical activity, and others, depending on the complexity of the simulation.

Here's the pseudocode:

// Pseudocode for Section 6: Sub-models and Overall Simulation

```
// Life Simulation for Training Experts in Various Fields


Procedure IntegrateSubModelsIntoSimulation():

   // Retrieve the main simulation framework

   simulationFramework <- RetrieveSimulationFramework()


   // Develop and integrate various sub-models

   cognitiveDevelopmentSubModel <- DevelopCognitiveDevelopmentSubModel()

   socialInteractionSubModel <- DevelopSocialInteractionSubModel()

   physicalActivitySubModel <- DevelopPhysicalActivitySubModel()


   // Integrate sub-models into the main simulation framework

   simulationFramework.IntegrateSubModel(cognitiveDevelopmentSubModel)

   simulationFramework.IntegrateSubModel(socialInteractionSubModel)
```

```
    simulationFramework.IntegrateSubModel(physicalActivitySubModel)

    // Apply weighting and coordination logic to manage interactions between sub-models
    ApplyWeightingAndCoordination(simulationFramework)

    // Update the main simulation framework
    UpdateSimulationFramework(simulationFramework)

    // Indicate completion of sub-models integration
    Display "Sub-models have been successfully integrated into the overall simulation."

EndProcedure

// Develop the sub-model for cognitive development
Function DevelopCognitiveDevelopmentSubModel() -> SubModel:
    cognitiveModel <- New SubModel()
    // Define the logic and parameters for cognitive development
    // ...
    Return cognitiveModel
EndFunction

// Develop the sub-model for social interaction
Function DevelopSocialInteractionSubModel() -> SubModel:
    socialModel <- New SubModel()
    // Define the logic and parameters for social interactions
    // ...
    Return socialModel
EndFunction
```

```
// Develop the sub-model for physical activity
Function DevelopPhysicalActivitySubModel() -> SubModel:
   physicalModel <- New SubModel()
   // Define the logic and parameters for physical activities
   // ...
   Return physicalModel
EndFunction

// Apply weighting and coordination logic between sub-models
Procedure ApplyWeightingAndCoordination(framework: SimulationFramework):
   // Implementation to define how different sub-models interact and influence each other
   // This might involve setting priority levels or interaction rules between sub-models
EndProcedure

// Update the main simulation framework with the integrated sub-models
Procedure UpdateSimulationFramework(framework: SimulationFramework):
   // Implementation to update the framework with the newly integrated sub-models
   // This may involve modifying the framework's internal data structures or algorithms
EndProcedure

// Main entry point
Begin
   IntegrateSubModelsIntoSimulation()
End
```

In this section, various sub-models are developed and integrated into the main simulation framework. The process includes defining specific aspects of each sub-model and then ensuring these sub-models work in harmony within the overall simulation structure. This integration is critical for creating a realistic and comprehensive simulation that accurately reflects the complexity of real-life scenarios and human development.

AU

You

And now finally the Simulation Implementation

Mixture of Experts, Function Callers & AI Agents

For the final section, "Simulation Implementation," the pseudocode will focus on actually running the simulation based on all the previously established parameters, agent definitions, formulas, and sub-models. This involves initiating the simulation, processing through the simulation timeline, handling interactions and events, and generating outcomes. Here's the pseudocode:

// Pseudocode for Simulation Implementation

```
// Life Simulation for Training Experts in Various Fields


Procedure RunSimulation():

   // Initialize the simulation with the necessary components

   simulationFramework <- RetrieveSimulationFramework()

   InitializeSimulationEnvironment(simulationFramework)


   // Begin simulation loop

   For each simulationStep in simulationTimeline:

      // Process each step in the simulation

      currentAgentsState <- RetrieveCurrentAgentsState()

      ProcessAgentInteractions(currentAgentsState, simulationStep)

      UpdateAgentStates(currentAgentsState, simulationStep)


      // Apply dynamic components and sub-models for current step

      ApplyDynamicComponents(simulationStep, currentAgentsState)

      ApplySubModels(simulationStep, currentAgentsState)


      // Generate outcomes for the current step

      simulationOutcomes <- GenerateOutcomes(simulationStep, currentAgentsState)
```

```
        StoreSimulationOutcomes(simulationOutcomes)

        // Check if simulation end criteria are met
        If CheckEndCriteria(simulationStep) Then
            Exit For // End simulation loop if criteria are met
        EndIf
    EndFor

    // Finalize the simulation and process results
    FinalizeSimulation(simulationFramework)

    // Indicate completion of the simulation
    Display "Simulation run complete. Processing results."

EndProcedure

// Initialize the simulation environment
Procedure InitializeSimulationEnvironment(framework: SimulationFramework):
    // Set up the environment and initial conditions for the simulation
    // ...
EndProcedure

// Retrieve the current state of all agents
Function RetrieveCurrentAgentsState() -> AgentsState:
    // Fetch the latest state of all agents participating in the simulation
    // Returns a data structure representing the current state of agents
EndProcedure

// Process interactions between agents for the current simulation step
```

```
Procedure ProcessAgentInteractions(agentsState: AgentsState, step: SimulationStep):
    // Implement logic to handle interactions between agents
    // ...
EndProcedure

// Update the states of agents based on interactions and simulation rules
Procedure UpdateAgentStates(agentsState: AgentsState, step: SimulationStep):
    // Update agent states based on the outcomes of interactions and simulation rules
    // ...
EndProcedure

// Apply dynamic components for the current simulation step
Procedure ApplyDynamicComponents(step: SimulationStep, agentsState: AgentsState):
    // Implement the application of dynamic components like time allocation and external factors
    // ...
EndProcedure

// Apply sub-models for the current simulation step
Procedure ApplySubModels(step: SimulationStep, agentsState: AgentsState):
    // Implement the application of sub-models like cognitive development, social interaction, etc.
    // ...
EndProcedure

// Generate outcomes for the current simulation step
Function GenerateOutcomes(step: SimulationStep, agentsState: AgentsState) -> Outcomes:
    // Calculate and generate outcomes based on the current state and simulation rules
    // Returns a data structure representing the outcomes of the current step
EndProcedure
```

```
// Store simulation outcomes for analysis and review
Procedure StoreSimulationOutcomes(outcomes: Outcomes):
  // Implement storage of outcomes for later analysis and review
  // ...
EndProcedure

// Check if the end criteria for the simulation are met
Function CheckEndCriteria(step: SimulationStep) -> Boolean:
  // Determine if the simulation should end based on predefined criteria
  // Returns true if end criteria are met, false otherwise
EndProcedure

// Finalize the simulation and process the results
Procedure FinalizeSimulation(framework: SimulationFramework):
  // Perform any final calculations, clean up, and prepare the simulation results for presentation
  // ...
EndProcedure

// Main entry point
Begin
  RunSimulation()
End
```

In this section, the simulation is executed, combining all the components and logic defined in the previous sections. The pseudocode covers the simulation lifecycle, from initialization to processing each step, and finally to concluding the simulation and preparing results. This comprehensive approach ensures that the simulation dynamically evolves and adapts to the interactions and changes occurring within it, providing a realistic and detailed simulation output.

AU

You

Where would we introduce this code to the stack? training_material_simulation: description: "Defines the input of educational and experiential content for each life stage." vocabulary_input: baby:

vocab_level: "Basic sounds and words recognition" sources: ["Parental speech", "Simple children's books"] toddler: vocab_level: "Simple sentences and questions" sources: ["Picture books", "Educational toys"] child: vocab_level: "Grade-level appropriate vocabulary" sources: ["Textbooks", "Children's literature", "Educational programs"] teenager: vocab_level: "Complex and abstract concepts" sources: ["Advanced textbooks", "Novels", "Scientific articles"] adult: vocab_level: "Specialized and professional terminology" sources: ["Professional journals", "Technical manuals", "Higher education materials"] content_input_by_subject: math: baby: content: ["Counting games"] toddler: content: ["Shape sorting", "Simple addition"] child: content: ["Multiplication tables", "Basic geometry"] teenager: content: ["Algebra", "Calculus"] adult: content: ["Advanced mathematics", "Statistics"] science: # Define content for all life stages # Add other subjects multimedia_input: documentaries: child: content: ["Nature documentaries", "Science for kids"] teenager: content: ["Historical documentaries", "Technology specials"] adult: content: ["In-depth analytical documentaries"] movies: # Define content for all life stages audio: # Define content for all life stages lesson_plans: toddler: lesson_plan: "Introduction to colors and shapes" child: lesson_plan: "Basic biology and life sciences" teenager: lesson_plan: "World history and human societies" adult: lesson_plan: "Business management and economics" training_material_integration: description: "Outlines the integration process of training materials into the model." integration_strategy: - "Mapping vocabulary level to language model tokens" - "Extracting key concepts from textbooks and educational content" - "Converting audio and visual material into descriptive text for model training" # Note: The actual integration of this content into the model will depend on the model's architecture and the processing capabilities of the system.

Mixture of Experts, Function Callers & AI Agents

The **`training_material_simulation`** component you've outlined is essentially a detailed specification for educational and experiential content to be used at different life stages of the simulation. This component can be integrated into the simulation stack at various points, depending on how you want the training materials to influence the agents' development and the overall simulation dynamics. Here's an overview of where and how it can be integrated:

Agent Definition and Development Stages (Section 2: Agent Definitions):

- During the definition of each life stage agent (Baby, Toddler, Child, Teenager, Adult), incorporate the **`vocabulary_input`**, **`content_input_by_subject`**, **`multimedia_input`**, and **`lesson_plans`** to define the educational and experiential learning capabilities and resources available to each agent.
- These inputs can help shape the cognitive and social development pathways for each agent.

Dynamic Simulation Components (Section 4: Dynamic Simulation Components):

- Use the training materials to inform the daily activities and developmental progressions of agents.
- For instance, integrate **`lesson_plans`** and **`multimedia_input`** into the daily routines of agents, adjusting their complexity and nature based on the agent's current life stage.

Sub-models Integration (Section 6: Sub-models and Overall Simulation):

- Within cognitive development and educational sub-models, the **`training_material_simulation`** can be a crucial component. Here, it helps define how agents acquire knowledge and skills in various subjects over time.
- The **`training_material_integration`** strategies can be particularly useful in ensuring the simulation processes educational content effectively.

- **Simulation Implementation (Final Section)**:
- As the simulation runs, the **`training_material_simulation`** component influences agent behavior and learning outcomes at each step.
- The simulation can be programmed to dynamically adjust the content and complexity of training materials based on agents' progress, using strategies from **`training_material_integration`**.

Feedback and Iteration:

- After running simulations, use feedback on how effectively agents interacted with and learned from the training materials to refine and update the content and integration strategies.

By integrating the **`training_material_simulation`** into these key areas, you ensure that the educational and experiential aspects of the simulation are closely aligned with the agents' developmental stages and the overall simulation objectives. This will contribute to a more realistic and effective simulation environment.

AU

You

better refactor the whole code then.

Mixture of Experts, Function Callers & AI Agents

Refactoring the entire code to incorporate the training_material_simulation component involves weaving educational and experiential content into various stages of the simulation. This requires modifications at different points of the simulation process to ensure the training materials effectively contribute to the agents' development and overall simulation dynamics. Below is the refactored pseudocode, integrating the **`training_material_simulation`** component into the existing structure:

1. Agent Definitions (Section 2)

```
// Refactor: Integrate Training Material into Agent Definitions

Procedure DefineAgentsWithTrainingMaterial():

  // ... [Previous code for defining agents]



  // Integrate training material into each agent's definition

  IntegrateTrainingMaterial(babyAgent, "baby")

  IntegrateTrainingMaterial(toddlerAgent, "toddler")

  IntegrateTrainingMaterial(childAgent, "child")

  IntegrateTrainingMaterial(teenagerAgent, "teenager")

  IntegrateTrainingMaterial(adultAgent, "adult")
```

```
    // ... [Rest of the procedure]
EndProcedure


Procedure IntegrateTrainingMaterial(agent: Agent, lifeStage: String):
    agent.SetVocabularyInput(training_material_simulation.vocabulary_input[lifeStage])
    agent.SetContentInputBySubject(training_material_simulation.content_input_by_subject[lifeStage])
    agent.SetMultimediaInput(training_material_simulation.multimedia_input[lifeStage])
    agent.SetLessonPlans(training_material_simulation.lesson_plans[lifeStage])
EndProcedure
```

2. Dynamic Simulation Components (Section 4)

```
// Refactor: Apply Training Material in Dynamic Simulation Components
Procedure ApplyDynamicComponents(step: SimulationStep, agentsState: AgentsState):
    // ... [Previous code]


    // Additional: Apply training material based on agent's life stage
    For each agent in agentsState:
        ApplyTrainingMaterial(agent, step)
    EndFor
EndProcedure


Procedure ApplyTrainingMaterial(agent: Agent, step: SimulationStep):
    // Implement logic to apply relevant training material to the agent
    // based on the current simulation step and agent's development stage
EndProcedure
```

3. Sub-models Integration (Section 6)

```
// Refactor: Enhance Sub-models with Training Material
Procedure DevelopCognitiveDevelopmentSubModel() -> SubModel:
    cognitiveModel <- New SubModel()
    // ... [Previous code]
```

```
    // Integrate training material simulation into cognitive development
    cognitiveModel.IntegrateTrainingMaterial(training_material_simulation)
    Return cognitiveModel
EndProcedure

// Similar refactoring for other sub-models...
```

4. Simulation Implementation (Final Section)

```
// Refactor: Utilize Training Material in Simulation Implementation
Procedure ProcessAgentInteractions(agentsState: AgentsState, step: SimulationStep):
    // ... [Previous code]

    // Additional: Process interactions based on training material exposure
    For each agentInteraction in agentsState:
        EvaluateTrainingMaterialImpact(agentInteraction)
    EndFor
EndProcedure

Procedure EvaluateTrainingMaterialImpact(agentInteraction: AgentInteraction):
    // Implement logic to evaluate the impact of training material on agent interactions
EndProcedure
```

Feedback and Iteration

- After simulation runs, analyze how agents utilized and responded to the training materials.
- Refine the integration strategies of **`training_material_simulation`** based on outcomes and feedback.

This refactoring ensures that the educational and experiential components are deeply integrated into the simulation, influencing agent development, interactions, and the overall simulation experience.

AU

You

I also want to create an specific 3 extra observers that describe: 1 the environment, 2 the parts of the environment that are observed by the main character in the simulation, and 3 the narrator observing the character and his reactions to stimuli and the environment. These should run in concert with the main algorithm.

Mixture of Experts, Function Callers & AI Agents

To integrate three specific observers into the simulation that run concurrently with the main algorithm, we need to define their roles and how they interact with the simulation environment and the main character. These observers will be:

- **Environment Observer**: Observes and describes the overall simulation environment.
- **Character Environment Observer**: Focuses on the parts of the environment directly observed or interacted with by the main character.
- **Narrator Observer**: Observes the main character, narrating their reactions to stimuli and the environment.

We'll add these observers to the simulation framework, ensuring they operate in tandem with the main simulation process. Here's how you can incorporate them:

Refactor: Integration of Observers

Initialization of Observers

```
Procedure InitializeObservers():

  environmentObserver <- New EnvironmentObserver()

  characterEnvironmentObserver <- New CharacterEnvironmentObserver()

  narratorObserver <- New NarratorObserver()


  // Initialize observers with necessary settings or data

  environmentObserver.Initialize()

  characterEnvironmentObserver.Initialize()

  narratorObserver.Initialize()


  // Store observers for further use in the simulation

  StoreObservers(environmentObserver, characterEnvironmentObserver, narratorObserver)

EndProcedure
```

Integration in the Simulation Loop

```
Procedure RunSimulation():

  // ... [Previous initialization code]
```

```
    // Initialize observers
    InitializeObservers()

    For each simulationStep in simulationTimeline:
        // ... [Process each step in the simulation]

        // Update and retrieve observations
        environmentObservation <- 
environmentObserver.UpdateAndGetObservation(simulationFramework)
        characterEnvironmentObservation <- 
characterEnvironmentObserver.UpdateAndGetObservation(currentAgentsState, mainCharacter)
        narratorObservation <- narratorObserver.UpdateAndGetObservation(mainCharacter, 
simulationStep)

        // Use observations in the simulation as needed
        ProcessObservations(environmentObservation, characterEnvironmentObservation, 
narratorObservation)

        // ... [Rest of the simulation step processing]
    EndFor

    // ... [Finalize the simulation]
EndProcedure
```

Observers Definition

```
Class EnvironmentObserver:
    // Observer for the overall simulation environment

    Method Initialize():
        // Initialize observer with necessary data
```

```
    Method UpdateAndGetObservation(framework: SimulationFramework) -> Observation:
        // Update observation based on the current state of the simulation environment
        // Return the updated observation

EndClass

Class CharacterEnvironmentObserver:
    // Observer for the environment as perceived by the main character

    Method Initialize():
        // Initialize observer

    Method UpdateAndGetObservation(agentsState: AgentsState, character: Agent) -> Observation:
        // Update observation based on the character's interaction with the environment
        // Return the updated observation

EndClass

Class NarratorObserver:
    // Observer that narrates the character's reactions and interactions

    Method Initialize():
        // Initialize observer

    Method UpdateAndGetObservation(character: Agent, step: SimulationStep) -> Observation:
        // Update observation based on the character's reactions and the current simulation step
        // Return the updated observation

EndClass
```

Processing Observations

```
Procedure ProcessObservations(environmentObservation: Observation, characterEnvironmentObservation: Observation, narratorObservation: Observation):

    // Implement logic to process and integrate observations into the simulation

    // This might involve adjusting simulation parameters or agent behaviors based on observations

EndProcedure
```

Conclusion

With these observers, the simulation gains additional layers of analysis and storytelling, enhancing the depth and interactivity of the simulation experience. Each observer provides a unique perspective, contributing to a more comprehensive understanding of the simulation dynamics and the main character's journey.