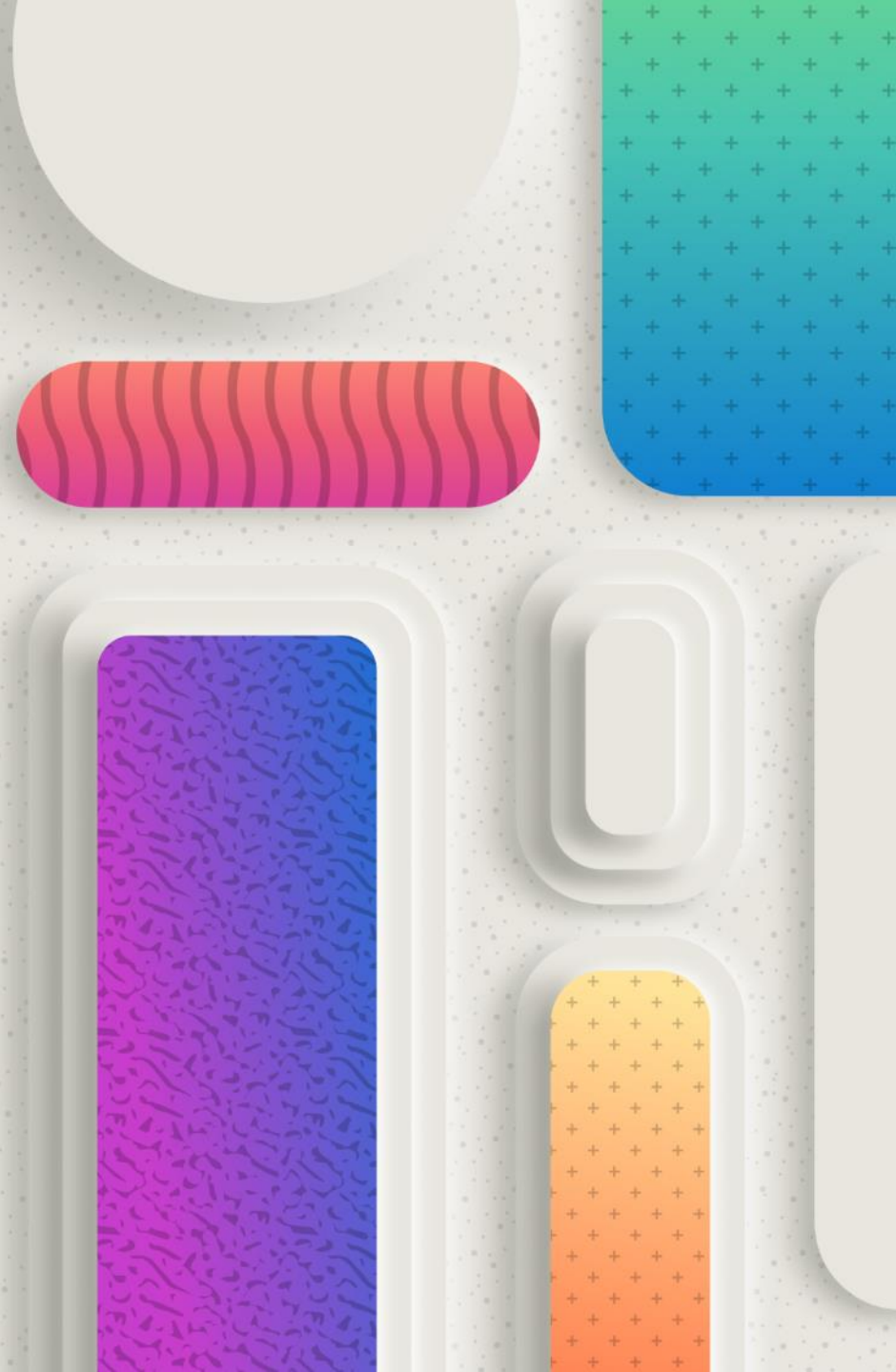
Microsoft

Multi-agent AI apps with Semantic Kernel and Azure Cosmos DB

Sandeep Nair
Senior Program Manager

Sajeetharan Sinnathurai
Principal Program Manager

Abhishek Gupta
Principal Product Manager



Agenda

> Intro

> Demo and architecture

> Build it – step by step

What are AI Agents?

Environment, actions, tools

"An agent is characterized by the ***environment*** it operates in and *the set of* ***actions*** it can perform."

"The *environment* an agent can operate in is defined by its use case."

"The *set of actions* an AI agent can perform is augmented by the ***tools*** it has access to."

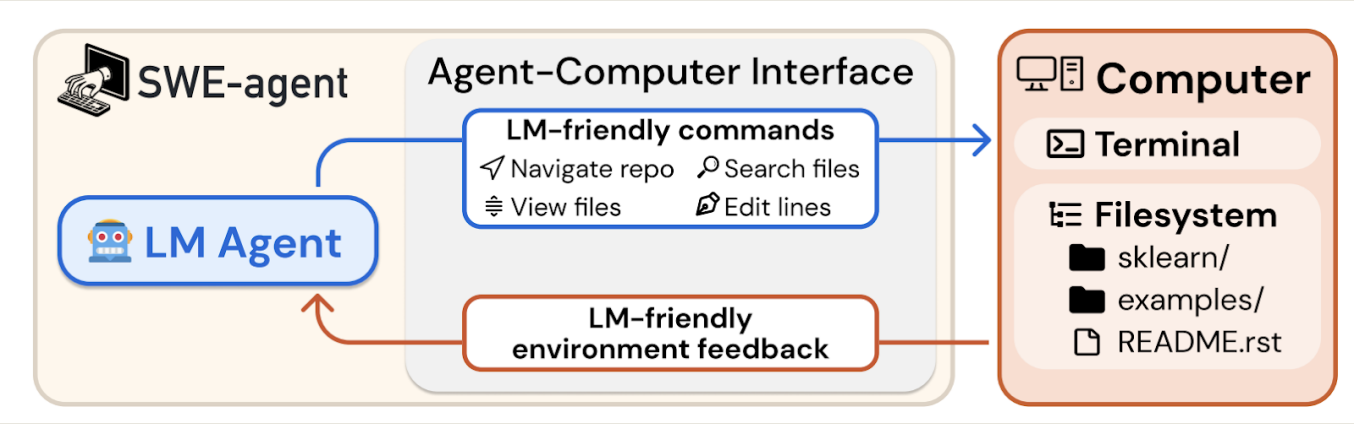
"The *environment* determines what *tools* an agent can potentially use."

https://huyenchip.com/2025/01/07/agents.html

Common features of AI Agents

- **Planning.** AI agents can create step-by-step plans and can use feedback to improve refine future outputs.
- **Memory.** Agents rely on short-term memory for immediate prompts and leverage long-term data retention—often via retrieval-augmented generation (RAG)—for broader context and recall.
- **Perception**. AI agents can retrieve data and process information making them more interactive and contextually aware.
- **Tool Use.** Agents can run functions or call external APIs for information or to perform actions. These are also called tools.

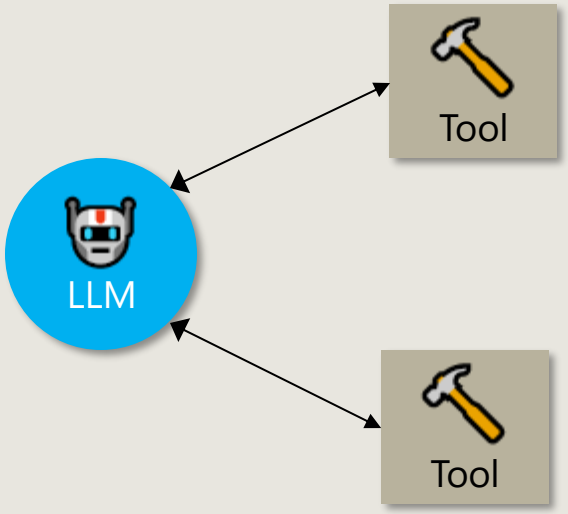
SWE (aka Coding) Agents



https://huyenchip.com/2025/01/07/agents.html

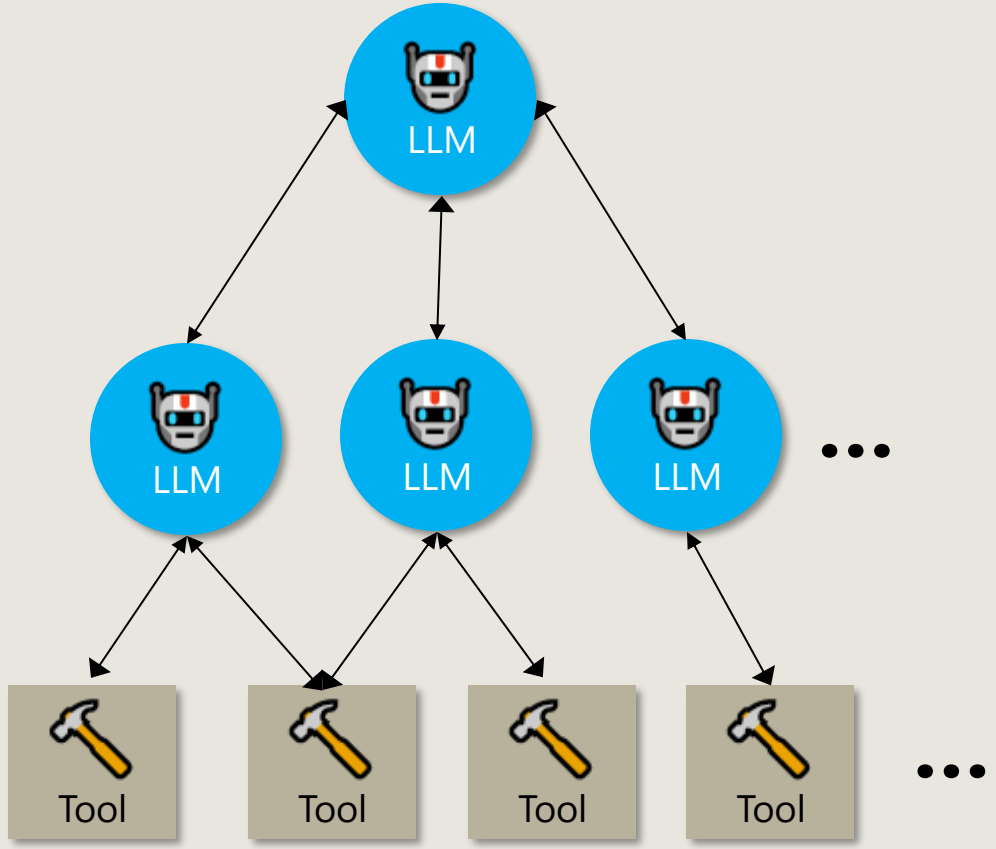
Two popular agentic architectures

Single-Agent



Simple
Doesn't scale to complex tasks

Router (Supervisor, Planner) Multi-agent



Scalable, flexible
Complex implementation

Prompts – Instructions for your agents

Prompts serve as a set of instructions or inputs provided to guide an agent's behavior. It helps the agent understand the context, determine appropriate actions (tool calling, passing to other agents) and generate relevant outputs related to the agent's purpose.

- **Define agent roles.** Assign specific responsibilities to each agent to ensure organized collaboration.
- **Give contextual instructions.** Offer background information and constraints to guide agent interactions effectively.
- **Interaction protocols.** Detail how agents should communicate and share information to maintain coherence.
- **Set clear objectives.** Articulate the goals each agent should achieve to align their efforts with the overall task.
- **Provide examples.** Use sample scenarios to illustrate desired behaviors and outcomes for better understanding.
- **Anticipate edge cases.** Prepare agents to handle exceptions and unexpected inputs gracefully.
- **Iterate and test.** Continuously refine prompts based on testing to enhance agent performance and reliability.

Example: Transactions Agent Prompt

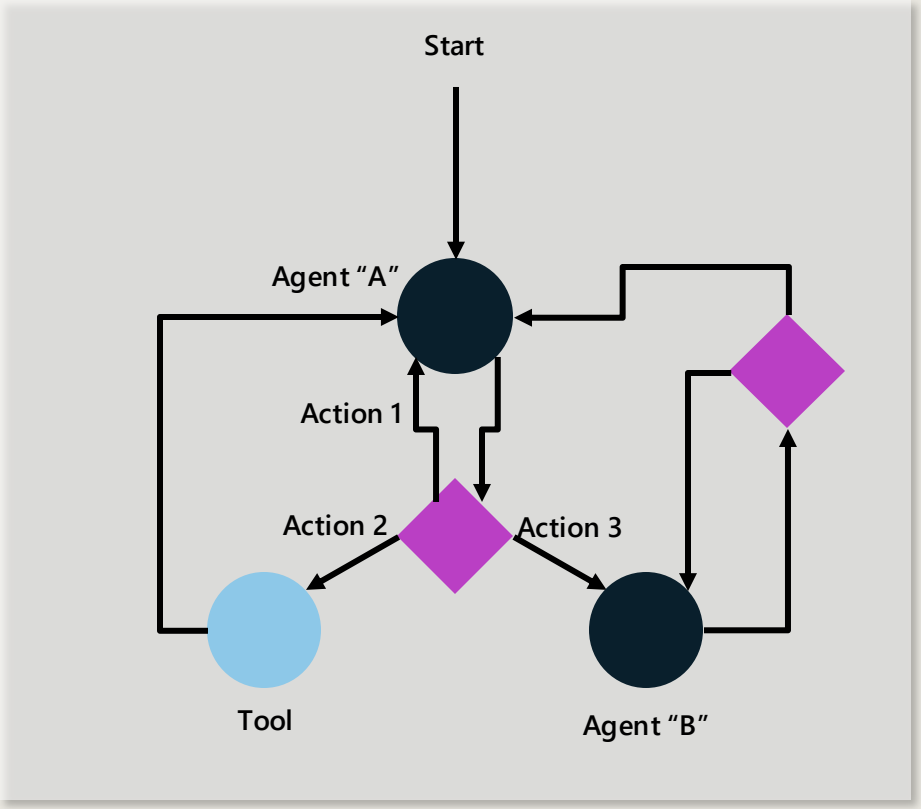
You are a banking transactions agent that can handle account balance enquiries and bank transfers.
If the user wants to make a deposit or withdrawal or transfer, ask for the amount and the account number which they want to transfer from and to.
Then call 'bank_transfer' tool with toAccount, fromAccount, and amount values.
Make sure you confirm the transaction details with the user before calling the 'bank_transfer' tool then call 'bank_transfer' tool with these values.
If the user wants to know transaction history, ask for the start and end date, and call 'get_transaction_history' tool with these values.
If the user needs general help, transfer to 'customer_support' for help.
You MUST respond with the repayment amounts before transferring to another agent.

Routing

Routing directs the flow of information and control between agents in a workflow, ensuring each agent receives appropriate inputs for cohesive system operation.

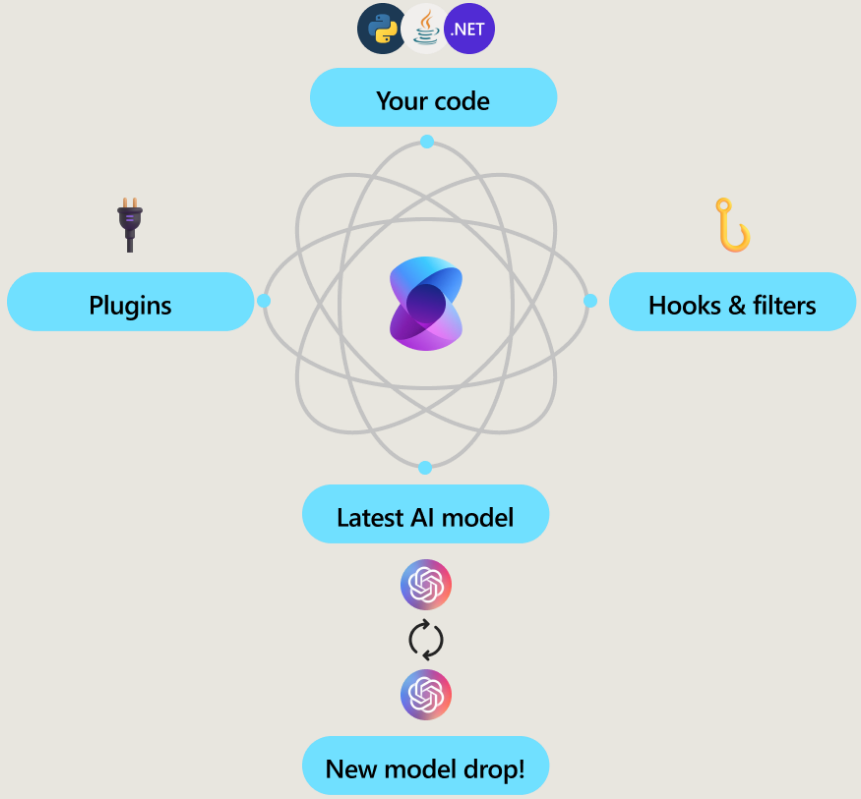
It facilitates task division among specialized agents in multi-agent systems for efficient collaboration.

- Define functions to determine the next agent based on the current state or input.
- Use these functions to establish edges between nodes (agents) in the workflow graph.



Introduction to Semantic Kernel

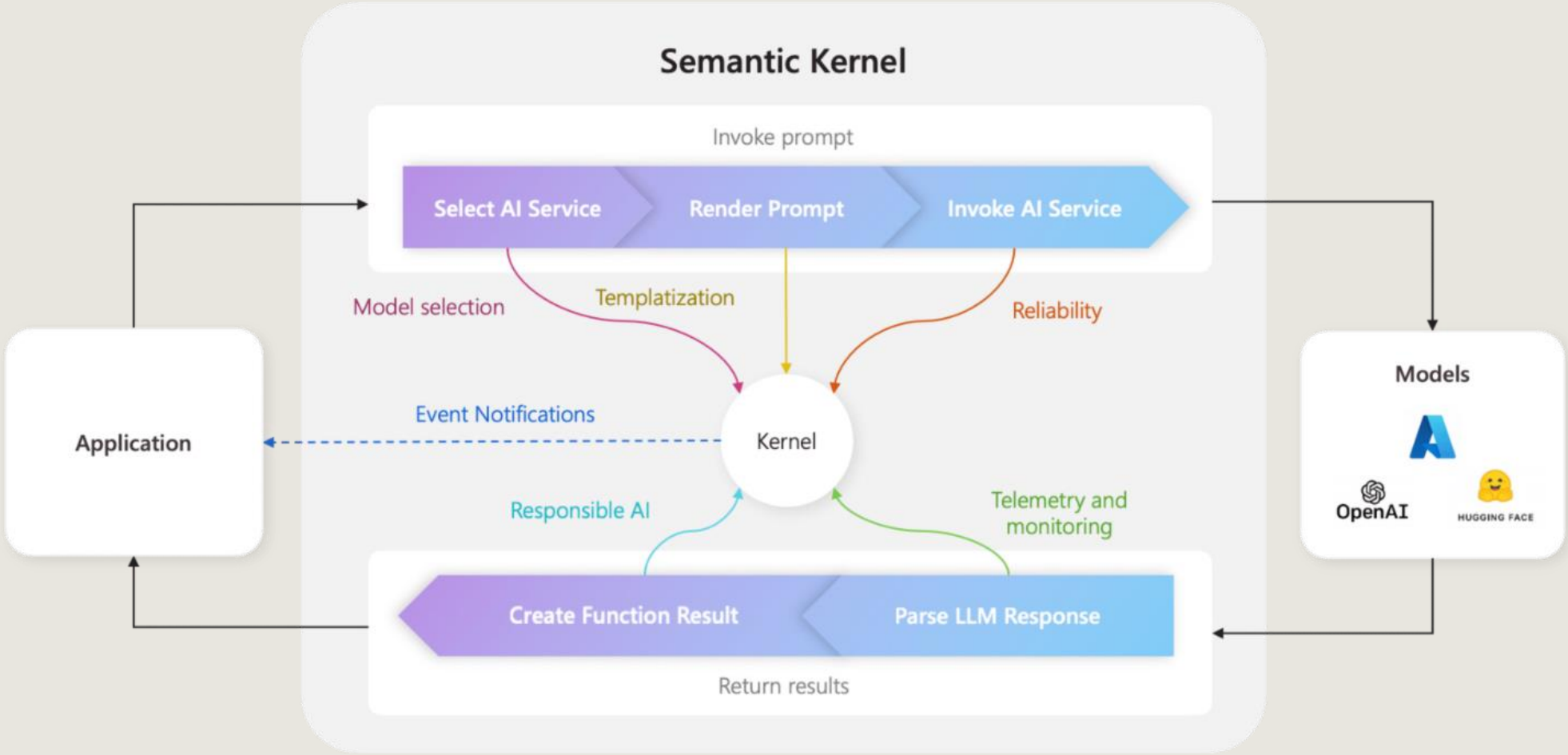
A lightweight, open-source development kit that lets you build AI agents and integrate the latest AI models into your C#, Python, or Java codebase. Reduces the need for repetitive coding and allows developers to focus on higher-level tasks.



Key features

- AI integrations – Chat completion, history, embeddings
- Function Calling, Plugins, Prompt engineering
- Vector Search Connectors
- Agent Framework
- Orchestration and Flow Control

Chat completion



Semantic Kernel – Function Calling

Automatic function calling

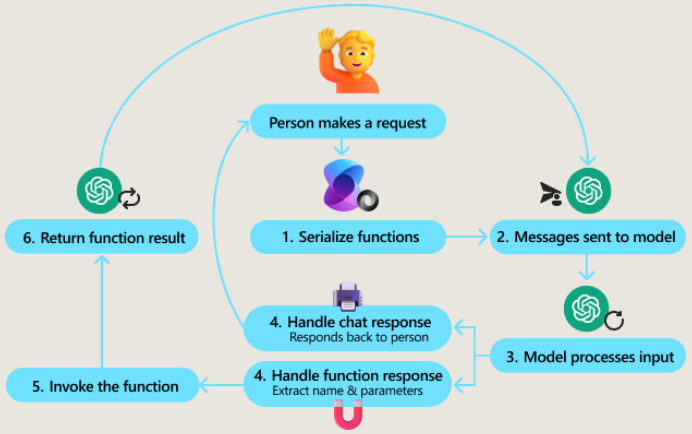
- Simplifies the process by automatically describing functions and their parameters to the model.
- Handles communication between the model and your code.

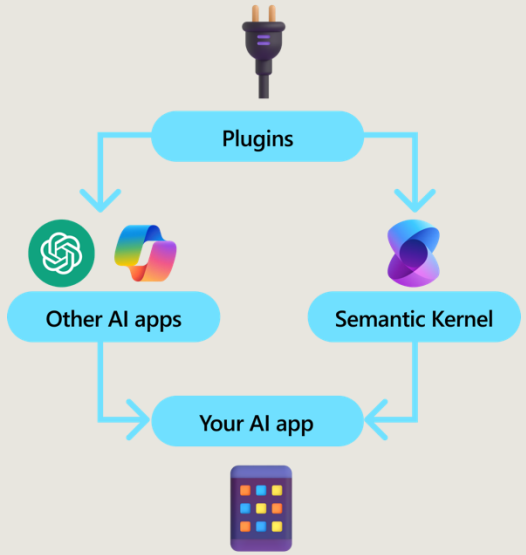
Function choice behavior

- Represented by three static methods: Auto, Manual, and Hybrid.
- Allows the AI model to choose from zero or more functions for invocation.

Integration with plugins

- Orchestrates interaction between the AI model and external APIs.
- Enables creation of plugins that the AI model can call.
- Incorporates the best parts of existing function-calling capabilities.
- Improves on ease of use, making it more extensible and reusable.





Agent framework

Tightly integrated with core features

- Kernel, Plugins (and functions), chat, templating, etc.

Agent

- Core abstraction for all types of agents
- Implementations include ChatCompletionAgent, AzureAIAgent, etc.

Agent Thread

- Core abstraction for threads or conversation state
- Abstraction on top of how conversation state may be managed for different agents.
- Maybe tied to a specific implementation e.g. AzureAIAgent works with AzureAIAgentThread

Agent Chat (and Agent Group Chat)

- For managing agent interactions within a chat environment
- AgentGroupChat – Strategy-based approach to allows multiple agents to collaborate across numerous interactions within the same conversation.

State & Memory

State

- A snapshot of your agentic application.
- It can contain any information useful to describe the snapshot.
- Ideally, it should have enough information such that the application can be restored at a given state.
- It can also help maintain consistency and context, enabling agents to perform tasks effectively and coordinate with each other.

Short -Term Memory

- Storage of information within a single session or interaction, allowing agents to maintain context during that session.
- Enables coherent and contextually relevant responses by remembering recent interactions.

Long-Term Memory

- Persistent storage of information across multiple sessions, enabling agents to recall past interactions and user preferences over time.
- Allows agents to learn from feedback, adapt to user preferences, and provide personalized experiences.
- Integrates with external databases to retain information between sessions.

Azure Cosmos DB for NoSQL

The most scalable, cost-effective AI database

Flexible data modeling

- **Chat history**
- **Vector Search**
- Agentic Memories
- User interactions
- E-commerce
- **Transactional & operational data**

Enterprise ready

- Elastic & hyperscale
- 99.999% availability
- <10ms point R/W transactions
- Disaster recovery, backup & restore
- Multi-region & Geo replication
- Virtual Networks, CMKs, RBAC

Usability & functional

- **Serverless model**
- Auto-scale, provisioned model
- **Schema-free data**
- **Built-in multitenancy**
- SQL-like query language
- Multiple consistency levels

GenAI + Data Use Cases with Azure Cosmos DB

Operational + Vector Database

No ETL

Consistent data

Reduce complexity & costs

Retrieval Augmented Generation (RAG)

Personalize LLM on your data

Cheaper than fine tuning

Faster iteration on new data

Conversational History

Conversational context

UX improvements

LLM optimizations

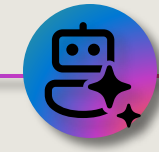
Auditing

Semantic Caching

Drastically reduces latency

Saves on Token consumption

Reduces costs and latency for LLM



AI Agents

Agent-to-agent transactions

Statefulness

Logging

Microsoft DiskANN in Azure Cosmos DB

Cost-effective, low latency vector search at any scale

Vector compression

Large Vectors
{ D1, D2, D3, D4, D5, …, D99, D100 }

Quantization

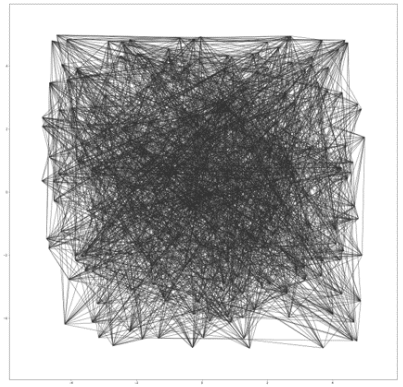
Compressed Vectors
{ D1, D2 .., D10 }

Storage and graph construction

RAM
Compressed vectors

SSD
Full vectors + graph

Algorithms

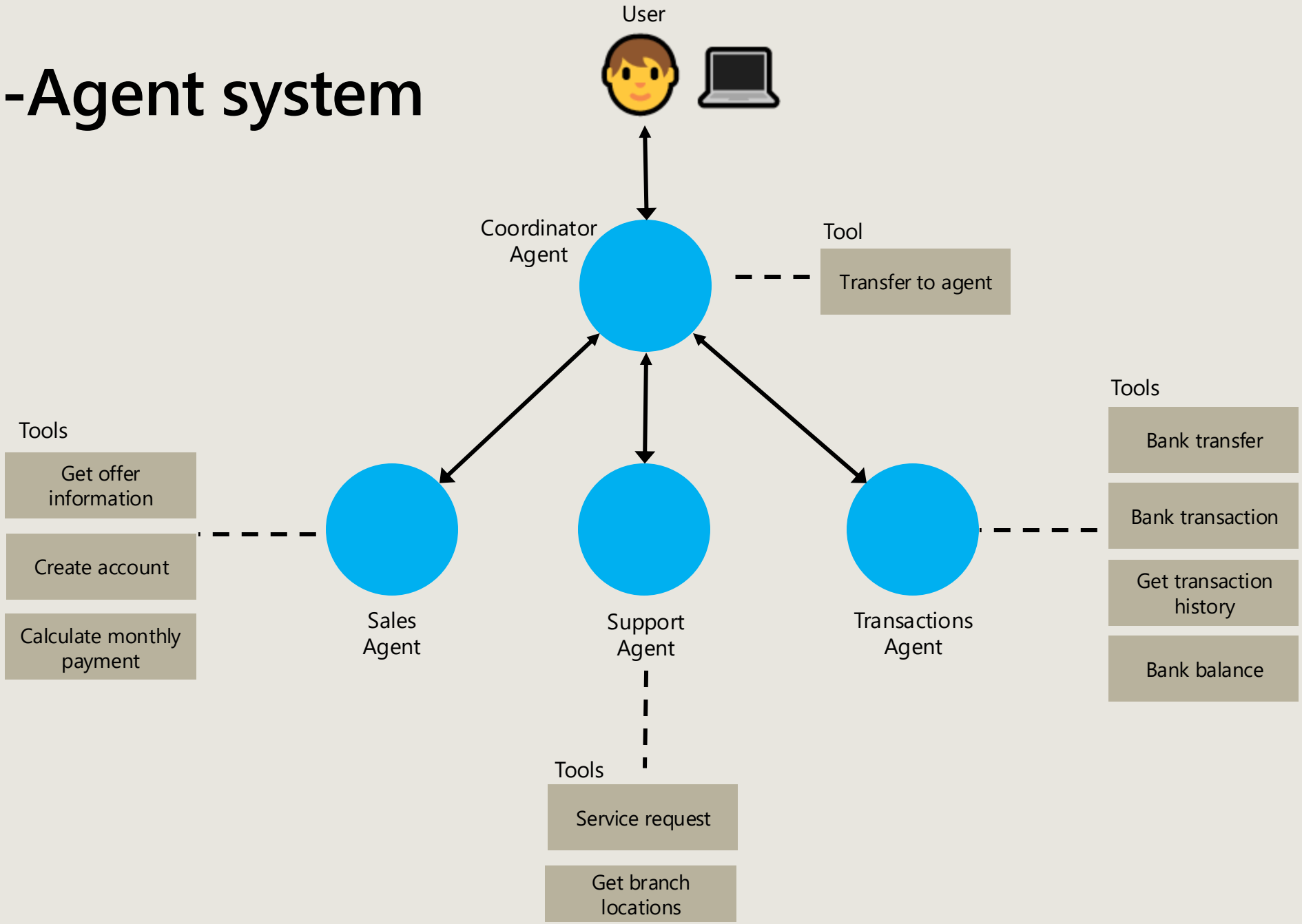


Unlimited scale | Low latency | Robust to data changes | Serverless

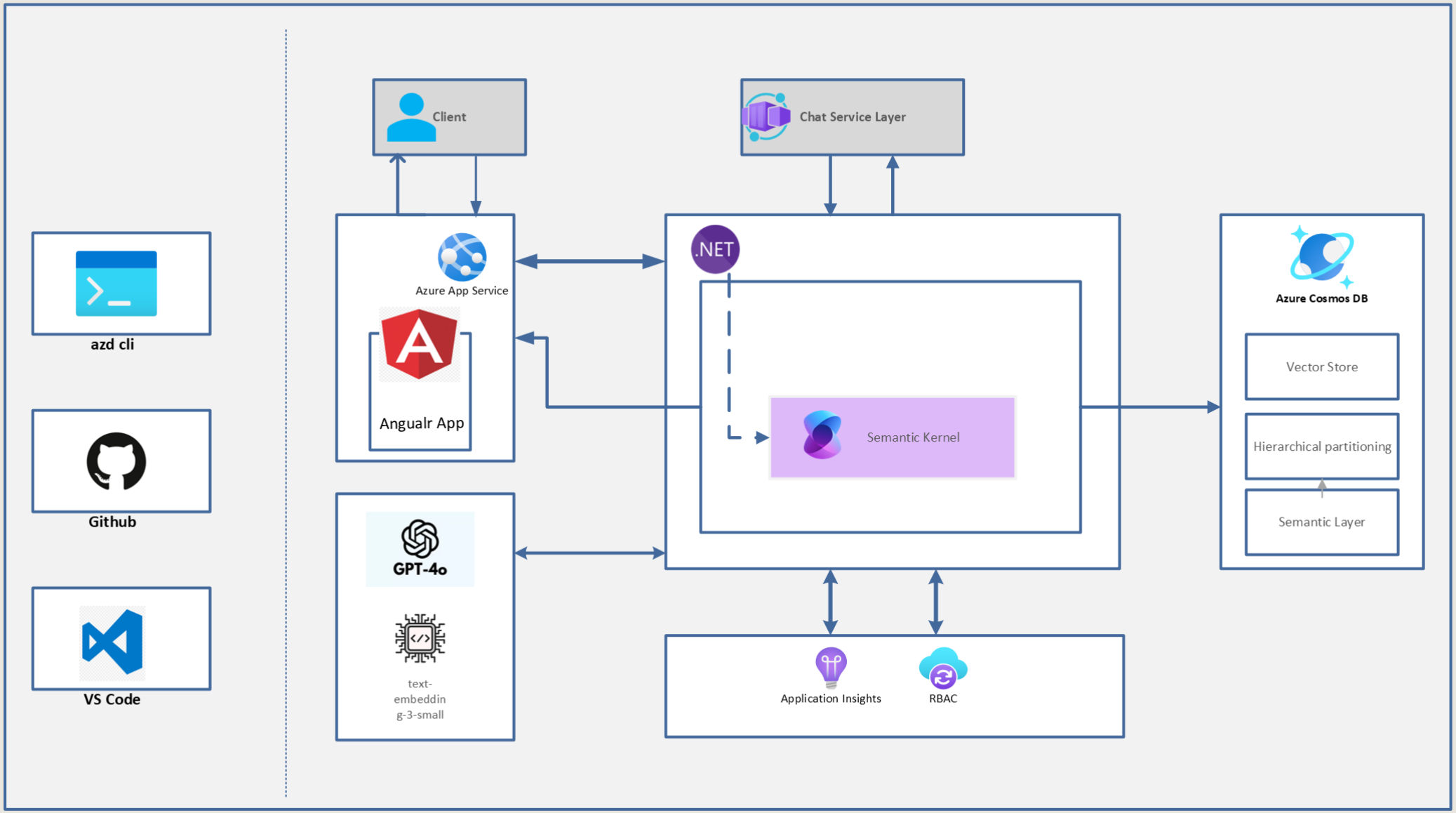
Read the whitepaper: aka.ms/DiskANNCosmosDBWhitePaper

Multi Agent Banking App in Action

Multi-Agent system

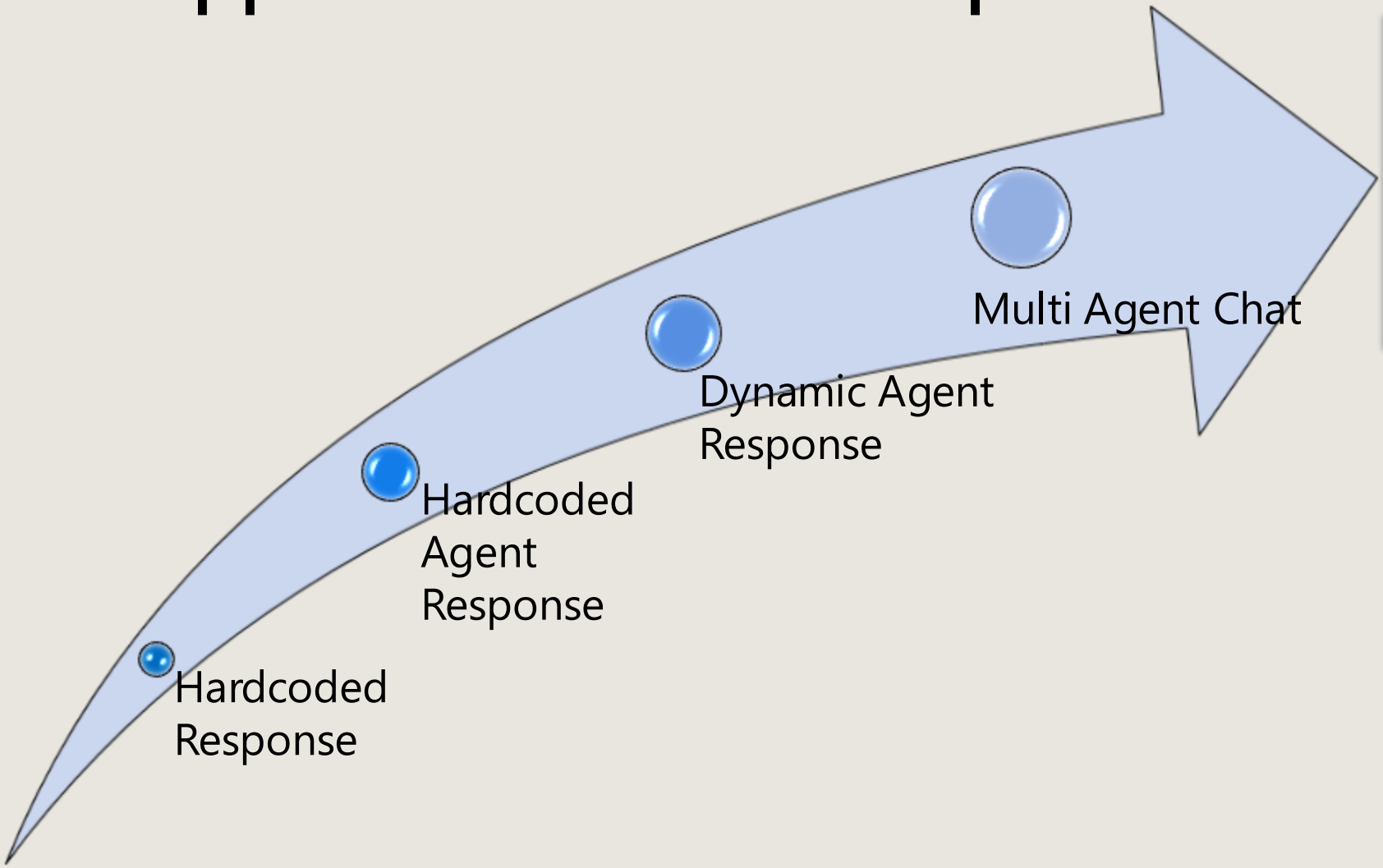


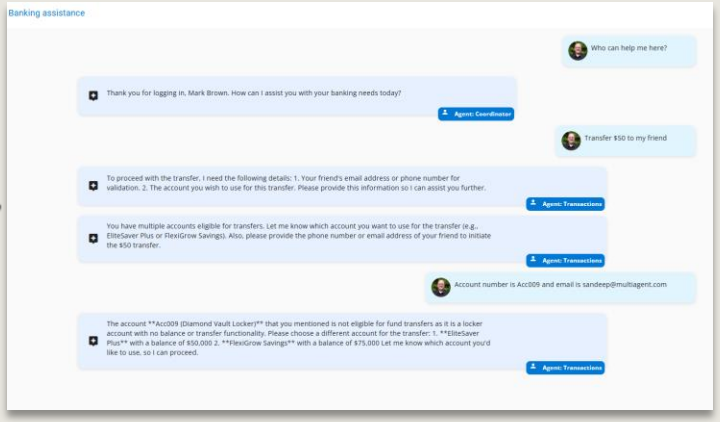
Deployment Architecture

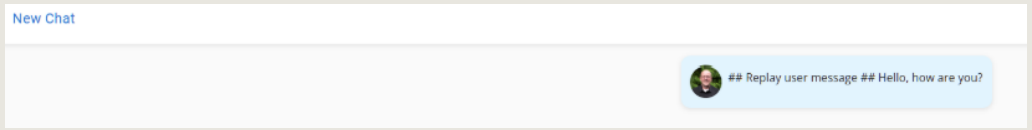


Code Walkthrough

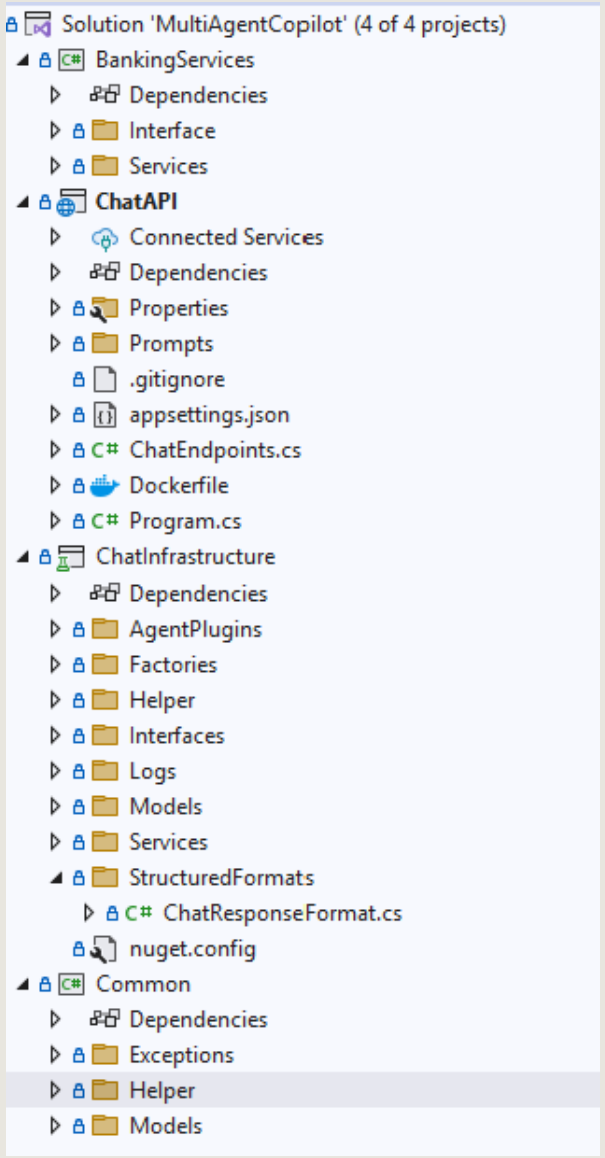
Our Approach for Chat Completion

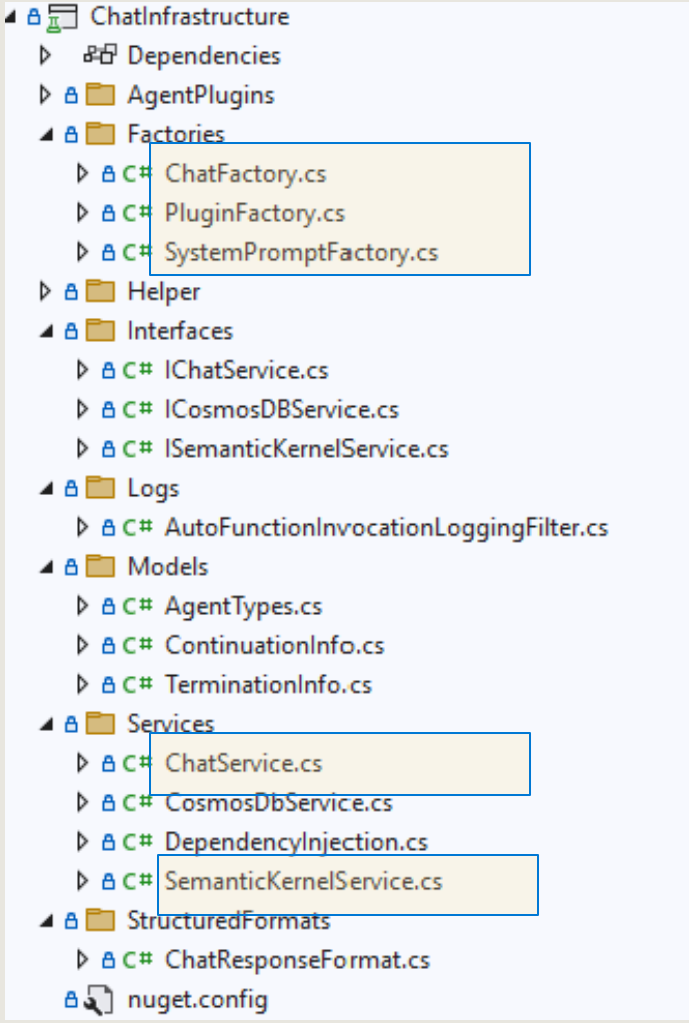






Solution Structure

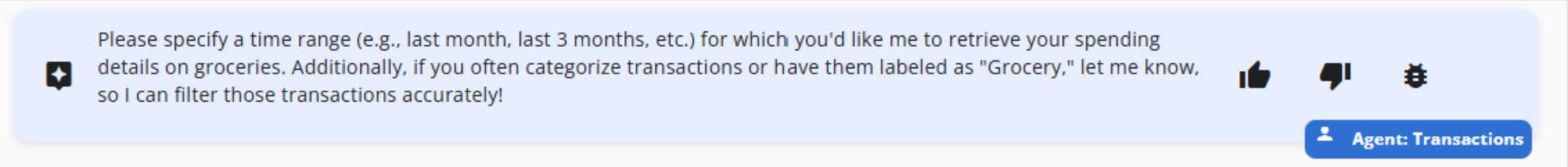




Hardcoded Response

```
public async Task<List<Message>> GetChatCompletionAsync(string tenantId, string userId,string? sessionId, string
userPrompt)
{
    try
    {
        ArgumentNullException.ThrowIfNull(sessionId);


         var userMessage = new Message(tenantId,userId,sessionId, "User","User", "## Replay user message ## " +
         userPrompt);

        return new List<Message> { userMessage };
    }
    catch (Exception ex)
    {
        _logger.LogError(ex, $"Error getting completion in session {sessionId} for user prompt [{userPrompt}].");
        return new List<Message> { new Message(tenantId, userId, sessionId!, "Error", "Error", $"Error getting
         completion in session {sessionId} for user prompt [{userPrompt}].") };
    }
}
```

Message Model




```
public Message(string tenantId, string userId,string sessionId, string author, string
authorRole, string textContent, string? id = null, string? debugLogId=null)
 {
     SessionId = sessionId;
     TenantId = tenantId;
     UserId = userId;
     Id = id ?? Guid.NewGuid().ToString();
     if (debugLogId != null)
         DebugLogId = debugLogId;
     Type = nameof(Message);
     Sender = author;
     SenderRole = authorRole;
     Text = textContent;
     TimeStamp = DateTime.UtcNow;
 }
```

Response logic – Let's move to a dedicated class

```
public async Task<List<Message>> GetChatCompletionAsync(string tenantId, string userId, string? sessionId, string
userPrompt)
 {
     try
     {
         ArgumentNullException.ThrowIfNull(sessionId);

         var userMessage = new Message(tenantId, userId, sessionId, "User", "User", userPrompt);

         // Generate the completion to return to the user
         var result = await _skService.GetResponse(userMessage, new List<Message>(), tenantId, userId);

         return result.Item1;
     }
     catch (Exception ex)
     {
         _logger.LogError(ex, $"Error getting completion in session {sessionId} for user prompt
[{userPrompt}].");
         return new List<Message> { new Message(tenantId, userId, sessionId!, "Error", "Error", $"Error getting
completion in session {sessionId} for user prompt [{userPrompt}].") };
     }
 }
```

Basic Agent Response

```
public async Task<Tuple<List<Message>, List<DebugLog>>> GetResponse(Message userMessage, List<Message> messageHistory, string
tenantId, string userId)
 {

     ChatCompletionAgent agent = new ChatCompletionAgent
     {
         Name = "BasicAgent",
         Instructions = "Greet the user and translate the request into French",
         Kernel = _semanticKernel.Clone()
     };

     ChatHistory chatHistory = [];

     chatHistory.AddUserMessage(userMessage.Text);

...


     await foreach (ChatMessageContent response in agent.InvokeAsync(chatHistory))
     {
         string messageId = Guid.NewGuid().ToString();
         completionMessages.Add(new Message(userMessage.TenantId, userMessage.UserId, userMessage.SessionId,
          response.AuthorName ?? string.Empty, response.Role.ToString(), response.Content ?? string.Empty, messageId));
     }
     return new Tuple<List<Message>, List<DebugLog>>(completionMessages, completionMessagesLogs);
 }
```

History & Agent Builder

```
public async Task<Tuple<List<Message>, List<DebugLog>>> GetResponse(Message userMessage, List<Message>
messageHistory, string tenantId, string userId)
 {
     try
     {
         ChatFactory agentChatGeneratorService = new ChatFactory();

         var agent = agentChatGeneratorService.BuildAgent(_semanticKernel, _loggerFactory, tenantId, userId);

         ChatHistory chatHistory = [];

         // Load history
         foreach (var chatMessage in messageHistory)
         {
             if (chatMessage.SenderRole == "User")
             {
                 chatHistory.AddUserMessage(chatMessage.Text);
             }
             else
             {
                 chatHistory.AddAssistantMessage(chatMessage.Text);
             }
         }
         chatHistory.AddUserMessage(userMessage.Text);
...
 }
```

Agent Builder

```
public ChatCompletionAgent BuildAgent(Kernel kernel, ILoggerFactory loggerFactory, string tenantId, string userId)
{
    ChatCompletionAgent agent = new ChatCompletionAgent
    {
        Name = SystemPromptFactory.GetAgentName(),
        Instructions = $"""{SystemPromptFactory.GetAgentPrompts()}""",
        Kernel = kernel.Clone()
    };

    return agent;
}
```

```
internal static class SystemPromptFactory
{
   public static string GetAgentName()
   {
      string name = "FrontDeskAgent";
      return name;
   }

   public static string GetAgentPrompts()
   {
      string prompt = "You are a front desk agent in a bank. Respond to the user queries professionally. Provide professional and helpful responses to user queries.Use your knowledge of banking services and procedures to address user queries accurately.";
      return prompt;
   }
}
```

Welcome Prompty

You are a Chat Initiator and Request Router in a bank.
Your primary responsibilities include welcoming users, identifying customers based on their login, routing requests to the appropriate agent.
Start with identifying the currently logged-in user's information and use it to personalize the interaction.For example, "Thank you for logging in, [user Name]. How can I help you with your banking needs today?"

RULES:
- Determine the nature of the user's request and silently route it to the appropriate agent.
- Avoid asking for unnecessary details to route the user's request. For example, "I see you have a question about your account balance. Let me connect you with the right agent who can assist you further."
- Do not provide any information or assistance directly; always route the request to the appropriate agent silently.
- Route requests to the appropriate agent without providing direct assistance.
- If another agent has asked a question, wait for the user to respond before routing the request.
- If the user has responded to another agent, let the same agent respond before routing or responding.
- When the user's request is fulfilled, ask for feedback on the service provided before concluding the interaction. Gauge their overall satisfaction and sentiment as either happy or sad. For example, "Before we conclude, could you please provide your feedback on our service today? Were you satisfied with the assistance provided? Would you say your overall experience was happy or sad?"
- Use the available functions when needed.

Load Prompts Dynamically

```
public static string GetAgentPrompts(AgentType agentType)
{
    string promptFile = string.Empty;
    switch (agentType)
    {
        case AgentType.Sales:
            promptFile = "Sales.prompty";
            break;
        case AgentType.Transactions:
            promptFile = "Transactions.prompty";
            break;
        case AgentType.CustomerSupport:
            promptFile = "CustomerSupport.prompty";
            break;
        case AgentType.Coordinator:
            promptFile = "Coordinator.prompty";
            break;
        default:
            throw new ArgumentOutOfRangeException(nameof(agentType), agentType, null);
    }

    string prompt = $"{File.ReadAllText("Prompts/" +
        promptFile)}{File.ReadAllText("Prompts/CommonAgentRules.prompty")}";

    return prompt;
}
```

Get Ready for Multiple Agents

```
public async Task<Tuple<List<Message>, List<DebugLog>>> GetResponse(Message userMessage, List<Message>
messageHistory, IBankDataService bankService, string tenantId, string userId)
 {
     try
     {
         ChatFactory agentChatGeneratorService = new ChatFactory();

         var agent = agentChatGeneratorService.BuildAgent(_semanticKernel, AgentType.Coordinator, _loggerFactory,
         bankService, tenantId, userId);

         ChatHistory chatHistory = [];
…
```

```
public ChatCompletionAgent BuildAgent(Kernel kernel, AgentType agentType,
ILoggerFactory loggerFactory, IBankDataService bankService, string
tenantId, string userId)
{
    ChatCompletionAgent agent = new ChatCompletionAgent
    {
        Name = SystemPromptFactory.GetAgentName(agentType),
        Instructions =
$"""{SystemPromptFactory.GetAgentPrompts(agentType)}""",
        Kernel = PluginFactory.GetAgentKernel(kernel, agentType,
loggerFactory, bankService, tenantId, userId),
        Arguments = new KernelArguments(new
AzureOpenAIPromptExecutionSettings() { FunctionChoiceBehavior =
FunctionChoiceBehavior.Auto() })
    };

    return agent;
}
```

Dynamic Kernel

```
internal static Kernel GetAgentKernel(Kernel kernel, AgentType agentType, ILoggerFactory loggerFactory,
IBankDataService bankService, string tenantId, string userId)
{
    Kernel agentKernel = kernel.Clone();
    switch (agentType)
    {
        case AgentType.Sales:
            var salesPlugin = new SalesPlugin(loggerFactory.CreateLogger<SalesPlugin>(), bankService, tenantId,
                  userId);
            agentKernel.Plugins.AddFromObject(salesPlugin);
            break;
        case AgentType.Transactions:
            var transactionsPlugin = new TransactionPlugin(loggerFactory.CreateLogger<TransactionPlugin>(),
                  bankService, tenantId, userId);
            agentKernel.Plugins.AddFromObject(transactionsPlugin);
            break;

…

        default:
            throw new ArgumentException("Invalid plugin name");
    }

    return agentKernel;
}
```

Plugins

```
public class BasePlugin
 {
     [KernelFunction("GetLoggedInUser")]
     [Description("Get the current logged-in BankUser")]
     public async Task<BankUser> GetLoggedInUser()
     {
         _logger.LogTrace($"Get Logged In User for Tenant:{_tenantId}  User:{_userId}");
         return await _bankService.GetUserAsync(_tenantId, _userId);
     }

     [KernelFunction("GetCurrentDateTime")]
     [Description("Get the current date time in UTC")]
     public DateTime GetCurrentDateTime()
     {
         _logger.LogTrace($"Get Datetime: {System.DateTime.Now.ToUniversalTime()}");
         return System.DateTime.Now.ToUniversalTime();
     }

     [KernelFunction("GetUserRegisteredAccounts")]
     [Description("Get user registered accounts")]
     public async Task<List<BankAccount>> GetUserRegisteredAccounts()
     {
         _logger.LogTrace($"Fetching accounts for Tenant: {_tenantId} User ID: {_userId}");
         return await _bankService.GetUserRegisteredAccountsAsync(_tenantId, _userId);
     }
...
```

Agent Specific Responses: Manual Agent Selection

```
var agent = agentChatGeneratorService.BuildAgent(_semanticKernel,
AgentType.Coordinator, _loggerFactory,      bankService, tenantId, userId);
```

AgentType.Coordinator	Hi
AgentType.Transactions	How much did I spend on groceries?
AgentType.Sales	Looking for a high interest savings account
AgentType.CustomerSupport	File a complaint for theft in Acc001

Before we make Agents Autonomous

Which Agent does what ?

How does an Agent get invoked?

Will multiple Agents chatter?

When will they stop responding to a user prompt?

How can I control these?

SelectionStrategy

Examine RESPONSE and choose the next participant.

Choose only from these participants:
- Coordinator
- CustomerSupport
- Sales
- Transactions

Always follow these rules when choosing the next participant:
- Determine the nature of the user's request and route it to the appropriate agent
- If the user is responding to an agent, select that same agent.
- If the agent is responding after fetching or verifying data , select that same agent.
- If unclear, select Coordinator.

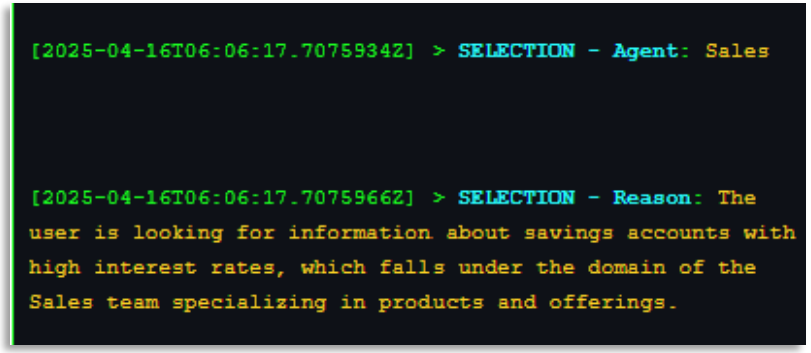
TerminationStrategy

Determine if agent has requested user input or has responded to the user's query.
Respond with the word NO (without explanation) if agent has requested user input.
Otherwise, respond with the word YES (without explanation) if any the following conditions are met:
- An action is pending by an agent.
- Further participation from an agent is required
- The information requested by the user was not provided by the current agent.

Structured Formats for each strategy

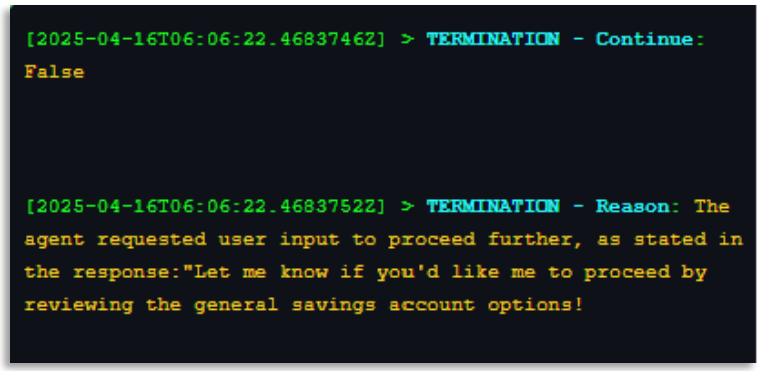
ChatResponseStrategy.Continuation

```
string jsonSchemaFormat_Continuation = """
{

 "type": "object",
  "properties": {
    "AgentName": { "type": "string",
         "description":"name of the selected
         agent" },
    "Reason": { "type": "string",
         "description":"reason for selecting the
         agent" }
   },
   "required": ["AgentName", "Reason"],
   "additionalProperties": false
}
""";
```



ChatResponseStrategy.Termination

```
string jsonSchemaFormat_termination = """
{

  "type": "object",
  "properties": {
    "ShouldContinue": { "type": "boolean",
         "description":"Does conversation require
         further agent participation" },
    "Reason": { "type": "string",
         "description":"List the conditions that
evaluated to true for further agent participation" }
     },
     "required": ["ShouldContinue", "Reason"],
     "additionalProperties": false
 }
 """;
```



Agent Group Chat

```
public AgentGroupChat BuildAgentGroupChat(Kernel kernel, ILoggerFactory loggerFactory, LogCallback logCallback,
IBankDataService bankService, string tenantId, string userId)
{
    AgentGroupChat agentGroupChat = new AgentGroupChat();
    var chatModel = kernel.GetRequiredService<IChatCompletionService>();

    kernel.AutoFunctionInvocationFilters.Add(new
AutoFunctionInvocationLoggingFilter(loggerFactory.CreateLogger<AutoFunctionInvocationLoggingFilter>()));

    foreach (AgentType agentType in Enum.GetValues(typeof(AgentType)))
    {
        agentGroupChat.AddAgent(BuildAgent(kernel, agentType, loggerFactory, bankService, tenantId, userId));
    }

    agentGroupChat.ExecutionSettings = GetAgentGroupChatSettings(kernel, logCallback);


    return agentGroupChat;
}
```

Agent Group Chat Settings

```
private AgentGroupChatSettings GetAgentGroupChatSettings(Kernel kernel, LogCallback logCallback)
{
    ChatHistoryTruncationReducer historyReducer = new(5);

    AgentGroupChatSettings ExecutionSettings = new AgentGroupChatSettings
    {
        SelectionStrategy =
            new KernelFunctionSelectionStrategy(GetStrategyFunction(ChatResponseFormatBuilder.ChatResponseStrategy.Continuation), kernel)
            {
                Arguments = new KernelArguments(GetExecutionSettings(ChatResponseFormatBuilder.ChatResponseStrategy.Continuation)),
                // Save tokens by only including the final few responses
                HistoryReducer = historyReducer,
                // The prompt variable name for the history argument.
                HistoryVariableName = "lastmessage",
                // Returns the entire result value as a string.
                ResultParser = (result) =>
                {
                    var resultString = result.GetValue<string>();
                    if (!string.IsNullOrEmpty(resultString))
                    {
                        var ContinuationInfo = JsonSerializer.Deserialize<ContinuationInfo>(resultString);
                        logCallback("SELECTION - Agent", ContinuationInfo.AgentName);
                        logCallback("SELECTION - Reason", ContinuationInfo.Reason);
                        return ContinuationInfo.AgentName;
                    }
                    else
                    {
                        return string.Empty;
                    }
                }
            },
```

Agent Group Chat Settings – Contd.

```
            TerminationStrategy =
             new KernelFunctionTerminationStrategy(GetStrategyFunction(ChatResponseFormatBuilder.ChatResponseStrategy.Termination),
                       kernel)
             {
                 Arguments = new KernelArguments(GetExecutionSettings(ChatResponseFormatBuilder.ChatResponseStrategy.Termination)),
                 // Save tokens by only including the final response
                 HistoryReducer = historyReducer,
                 // The prompt variable name for the history argument.
                 HistoryVariableName = "lastmessage",
                 // Limit total number of turns
                 MaximumIterations = 8,
                 // user result parser to determine if the response is "yes"
                 ResultParser = (result) =>
                 {
                     var resultString = result.GetValue<string>();
                     if (!string.IsNullOrEmpty(resultString))
                     {
                         var terminationInfo = JsonSerializer.Deserialize<TerminationInfo>(resultString);
                         logCallback("TERMINATION - Continue", terminationInfo.ShouldContinue.ToString());
                         logCallback("TERMINATION - Reason", terminationInfo.Reason);
                         return !terminationInfo.ShouldContinue;
                     }
                     else
                     {
                         return false;
                     }
                 }
             },
    };
    return ExecutionSettings;
}
```

Put it all together

```
public async Task<Tuple<List<Message>, List<DebugLog>>> GetResponse(Message userMessage, List<Message> messageHistory, IBankDataService
bankService, string tenantId, string userId)
 {
     try
     {
         ChatFactory multiAgentChatGeneratorService = new ChatFactory();

         var agentGroupChat = multiAgentChatGeneratorService.BuildAgentGroupChat(_semanticKernel, _loggerFactory, LogMessage,
bankService, tenantId, userId);

         // Load history
         foreach (var chatMessage in messageHistory)
         {
             AuthorRole? role = AuthorRoleHelper.FromString(chatMessage.SenderRole);
             var chatMessageContent = new ChatMessageContent
             {
                 Role = role ?? AuthorRole.User,
                 Content = chatMessage.Text
             };
             agentGroupChat.AddChatMessage(chatMessageContent);
         }

         _promptDebugProperties = new List<LogProperty>();

         List<Message> completionMessages = new();
         List<DebugLog> completionMessagesLogs = new();
```

Put it all together- Contd.

```
        do
        {
            var userResponse = new ChatMessageContent(AuthorRole.User, userMessage.Text);
            agentGroupChat.AddChatMessage(userResponse);
            agentGroupChat.IsComplete = false;

            await foreach (ChatMessageContent response in agentGroupChat.InvokeAsync())
            {
                string messageId = Guid.NewGuid().ToString();
                string debugLogId = Guid.NewGuid().ToString();
                completionMessages.Add(new Message(userMessage.TenantId, userMessage.UserId, userMessage.SessionId, response.AuthorName
                                ?? string.Empty, response.Role.ToString(), response.Content ?? string.Empty, messageId, debugLogId));

                if (_promptDebugProperties.Count > 0)
                {
                    var completionMessagesLog = new DebugLog(userMessage.TenantId, userMessage.UserId, userMessage.SessionId, messageId,
                                                             debugLogId);
                    completionMessagesLog.PropertyBag = _promptDebugProperties;
                    completionMessagesLogs.Add(completionMessagesLog);
                }
            }
        }
        while (!agentGroupChat.IsComplete);
        return new Tuple<List<Message>, List<DebugLog>>(completionMessages, completionMessagesLogs);
    }
    catch (Exception ex)
    {
        _logger.LogError(ex, "Error when getting response: {ErrorMessage}", ex.ToString());
        return new Tuple<List<Message>, List<DebugLog>>(new List<Message>(), new List<DebugLog>());
    }
}
```

Tips & Lessons learned

Use an AI Framework for agent, service, and tool orchestration
Frameworks like Semantic Kernel can simplify the coordination, communication, and workflow management in apps that have multiple AI agents.

Define clear agent roles and responsibilities
Clearly delineate each agent's functions to improve accuracy, predictability, and reliability.

Use a database for data storage and retrieval
Azure Cosmos DB provides a scalable and globally database for history and memory storage. It’s built-in semantic search capability enable you to use the same database for knowledge retrieval, keeping your app architecture simple.

Incorporate state persistence, logging, and disaster recovery
Store application states in persistent storage with the ability to handle failures gracefully and recover from errors to maintain overall functionality.

Questions

Start building agentic apps with Azure Cosmos DB!

Multi-agent Workshop

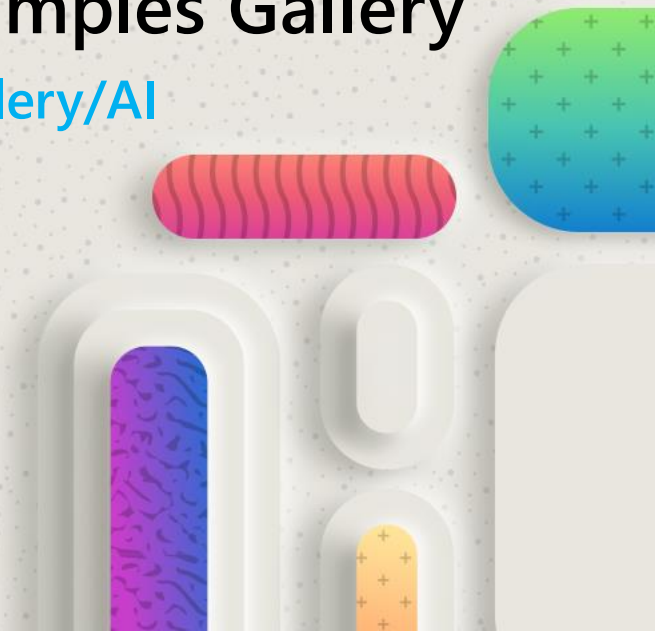
aka.ms/CosmosDB/BankingAgentWorkshop

Try Cosmos DB for Free!

aka.ms/TryCosmos

Azure Cosmos DB Samples Gallery

aka.ms/AzureCosmosDB/Gallery/AI



What will you learn next?

Register for your next session and keep hacking!

Hackathon runs April 8–30, 2025

aka.ms/agentshack