14-Dic-2024

Implementation of the TAiChat Component in Delphi

Change Log

December 14, 2024

- Added support for Grok Chat with the TAiGrokChat component.
- Added support for Grok Embeddings with the TAiGrokEmbeddings component.
- Added the `NativeInputFiles` property to filter files passed directly to the model.
- Added the `NativeOutputFiles` property to specify the desired format for the model's response.

Introduction

In today's digital era, artificial intelligence (AI) has become an essential tool for developing advanced applications. The ability to understand and process natural language has opened new frontiers in human-machine interaction. With this in mind, developing a framework that simplifies connectivity with various AI models is crucial. TAiChat and its associated components aim to provide developers with a simple yet powerful way to integrate this functionality into their Delphi applications.

A well-designed framework lowers barriers to entry, allowing developers to focus on implementing innovative solutions without worrying about the underlying complexities of each proprietary AI model API. By offering compatibility with multiple industry-leading models such as OpenAI, Anthropic, Gemini, and others, it ensures the flexibility needed to adapt to the changing demands of software development.

Potential Use Cases for AI Models in Delphi

With AI models available in a programming environment like Delphi, developers can create smarter and more responsive applications across various sectors. Examples of potential applications include:

1. **Virtual Assistants**: Implement assistants capable of handling complex queries, answering frequently asked questions, and providing personalized recommendations, enhancing user experience.

2. **Sentiment Analysis**: Integrate modules that analyze and understand the sentiment behind text, useful for monitoring social media, opinion surveys, and customer satisfaction.
3. **Content Generation**: Automate the creation of content through text, such as summaries, reports, and articles, optimizing time and human resources.
4. **Automatic Translation**: Develop applications offering real-time text translation, facilitating multilingual communication.
5. **Intelligent Chatbots**: Design chatbots that can learn and adapt their responses with each interaction, providing continuous and more contextual support to users.
6. **Speech-to-Text Recognition**: Implement functionalities that convert voice into text, enabling more natural and fluid data input.
7. **Creative Inspiration**: Assist in generating ideas for creative writing, game design, and other fields, allowing professionals to explore new possibilities.

What is TAiChat?

General Purpose

The TAiChat component for Delphi simplifies access and interaction with large language models (LLMs) via different APIs, providing a unified interface for Delphi developers. This allows programmers to efficiently integrate AI capabilities into their applications without needing to manage the specific complexities of each API.

Interaction with AI Models

TAiChat serves as a base component, encapsulating the complexity of interacting with LLM APIs. This component is the foundation for other derived components like TAiOllamaChat, TAiGeminiChat, TAiClaudeChat, TAiMistralChat, and TAiOpenChat. These components encapsulate and adapt the specifics of each API, simplifying integration with current and future AI models.

Key Features

- **Communication with LLMs**: Sends text messages to the models to obtain responses.
- **Tool Functions**: Facilitates the execution of callbacks and the handling of functions defined by the AI.
- **Memory Management**: Enables the model to remember data from both the current conversation and stored memory.
- **Parameter Configuration**: Customize features like temperature, max_tokens, and other execution parameters.

- **Attachment Handling**: Processes multimedia files (images, audio, PDFs) for preprocessing, such as converting or interpreting files before reaching the LLM.

onfiguration and Customization

TAiChat allows the configuration of all parameters necessary for the optimal functioning of the AI model. Properties such as `temperature`, `max_tokens`, and other features can be adjusted according to the specific needs of the application and are applied based on the particularities of each model's API.

Error and Exception Handling

TAiChat includes mechanisms for handling connection and processing errors, ensuring robust interaction with APIs. Errors are managed within each procedure, and in asynchronous modes, they are appropriately captured for handling within the Delphi environment.

Examples and Demonstrations

Basic Example

```
var
  Chat: TAiChat;
begin
  Chat := TAiChat.Create(nil);
  try
    Chat.ApiKey := 'tu-api-key';
    Chat.Model := 'gpt-4';
    Chat.AddMessage('¿Cuál es la capital de Francia?', 'user');
    ShowMessage(Chat.Run);
  finally
    Chat.Free;
  end;
end;
```

2. Example with MediaFile Processing (Images):

```
var
  Res: String;
  MediaFile: TAiMediaFile;
begin
  MediaFile := TAiMediaFile.Create;
  MediaFile.LoadFromFile('mipath/imagefilename.jpg');
  Res := Chat.AddMessageAndRun('Describe this image', 'user', [MediaFile]);
  ShowMessage(Res);
  MediaFile.Free;
end;
```

Here is the translated portion of your manual into English:

---

3. Example with MediaFile Processing (Audio) with Direct Call (OpenAI Only)

```
Var
  Res: String;
  MediaFile: TAiMediaFile;
  Msg: TAiChatMessage;
  FileName: String;
begin

  MediaFile := TAiMediaFile.Create;
  MediaFile.LoadFromFile('c:\temp\prompt.wav');

  Try
    Msg := AiOpenChat1.AddMessageAndRunMsg(MemoPrompt.Lines.Text, 'user', [MediaFile]);
  Finally
    FreeAndNil(MediaFile);
  End;

  MemoResponse.Lines.Text := Msg.Content;

  If (Msg.MediaFiles.Count > 0) and (Assigned(Msg.MediaFiles[0].Content)) then
  Begin
    FileName := 'c:\temp\response3' + Cons.ToString + '.wav';
    Msg.MediaFiles[0].Content.Position := 0;
    Msg.MediaFiles[0].Content.SaveToFile(FileName);

    Try
      MediaPlayer1.FileName := FileName;
      MediaPlayer1.Play;
    Finally
    End;
  End;
end;
```

4. Example with Process and MediaFiles (Voice) with Event Response (OpenAI Only)

```
Var
  Res: String;
  MediaFile: TAiMediaFile;
begin
  MediaFile := TAiMediaFile.Create;
  MediaFile.LoadFromFile('c:\temp\prompt.wav');

  Try
    Res := AiOpenChat1.AddMessageAndRun(MemoPrompt.Lines.Text, 'user', [MediaFile]);
    MemoResponse.Lines.Text := Res;
  Finally
    FreeAndNil(MediaFile);
  End;
End;
```

In the **OnReceiveDataEnd** event, the parameter `aMsg: TAiChatMessage` is received, from which the result can be obtained.

```
procedure TForm75.AiOpenChat1ReceiveDataEnd(const Sender: TObject; aMsg: TAiChatMessage;
aResponse: TJSONObject; aRole, aText: string);
Var
  FileName: String;
begin

  If (aMsg.MediaFiles.Count > 0) and (Assigned(aMsg.MediaFiles[0].Content)) then
  Begin
    Inc(Cons);
    FileName := 'c:\temp\response' + Cons.ToString + '.wav';
    aMsg.MediaFiles[0].Content.Position := 0;
    aMsg.MediaFiles[0].Content.SaveToFile(FileName);
    Try
      MediaPlayer1.FileName := FileName;
      MediaPlayer1.Play;
    Finally
    End;
  End;
end;
```

Public Properties:

1. **Messages**: `TAiChatMessages`
   Collection of messages managed by the component.
2. **LastError**: `String`
   Stores the last error that occurred during the component's execution.

Published Properties:

1. **ApiKey**: `String`
   API key for the AI model.
2. **Model**: `String`
   Specifies the model to be used for chat interactions (e.g., GPT-3.5, GPT-4, etc.).
3. **Frequency_penalty**: `Double`
   Penalty for frequent words in the generated response (-2 to 2).
4. **Logit_bias**: `String`
   Optional parameter to adjust the logit biases of specific words. Can be an empty string or contain values between -100 and 100.
5. **Logprobs**: `Boolean`
   Enables or disables the capture of logarithmic probabilities.
6. **Top_logprobs**: `String`
   Specifies how many top logarithmic probabilities will be returned (values between 0 and 5).
7. **Max_tokens**: `Integer`
   Token limit for a response. If set to 0, the model's maximum allowable value is used.
8. **N**: `Integer`
   Number of responses generated per input message (default is 1).
9. **Presence_penalty**: `Double`
   Penalty to avoid repetition of similar concepts in responses (-2.0 to 2.0).
10. **Response_format**: `TAiOpenChatResponseFormat`
    Format of the model's response, compatible with certain models like GPT-4.
11. **Seed**: `Integer`
    Seed used for generating response variations. If set to 0, no seed is applied.

12. **Stop**: `String`
    Keywords to stop the response generation. Multiple words can be defined, separated by commas.
13. **Asynchronous**: `Boolean`
    Determines if responses are processed asynchronously.
14. **Temperature**: `Double`
    Controls randomness in response generation (values between 0 and 2).
15. **Top_p**: `Double`
    Adjusts the cumulative probability for response generation (values between 0 and 1).
16. **Tools**: `TStrings`
    List of tools available for the model to process interactions.
17. **Tool_choice**: `String`
    Name of the selected tool for chat processing.
18. **Tool_Active**: `Boolean`
    Indicates whether the selected tool is active.
19. **User**: `String`
    User associated with the chat interactions.
20. **InitialInstructions**: `TStrings`
    Initial instructions sent to the model at the start of the conversation.
21. **Prompt_tokens**: `Integer`
    Number of tokens used in the conversation "prompt."
22. **Completion_tokens**: `Integer`
    Number of tokens generated in the response.
23. **Total_tokens**: `Integer`
    Sum of tokens used in the "prompt" and response.
24. **LastContent**: `String`
    Last content sent in the conversation.
25. **LastPrompt**: `String`
    Last "prompt" sent to the model.
26. **Busy**: `Boolean`
    Indicates whether the component is busy processing a request.
27. **OnReceiveData**: `TAiOpenChatDataEvent`
    Event triggered upon receiving data from the model.
28. **OnReceiveDataEnd**: `TAiOpenChatDataEvent`
    Event triggered when data reception has completed.
29. **OnAddMessage**: `TAiOpenChatDataEvent`
    Event triggered when a message is added to the conversation.
30. **OnCallToolFunction**: `TOnCallToolFunction`
    Event triggered when invoking a tool function.
31. **OnBeforeSendMessage**: `TAiOpenChatBeforeSendEvent`
    Event that allows intercepting and modifying a message before sending it.
32. **OnInitChat**: `TAiOpenChatInitChat`
    Event triggered when starting a new conversation.
33. **Url**: `String`
    URL of the AI model's service.

34. **AIChatConfig**: `TAiChatConfig`
    Configuration used for interactions with the model.
35. **ResponseTimeOut**: `Integer`
    Maximum wait time for receiving a response from the model.
36. **Memory**: `TStrings`
    Persistent memory used to store relevant data for the conversation.
37. **Functions**: `TFunctionActionItems`
    List of functions that can be executed within the conversation flow.
38. **AiFunctions**: `TAiFunctions`
    Collection of specific AI functions for executing tasks.
39. **OnProcessMediaFile**: `TAiOpenChatOnMediaFile`
    Event triggered when processing a media file.
40. **JsonSchema**: `TStrings`
    JSON schema used to validate or structure the model's responses.
41. **NativeInputFiles**: `TAiFileCategories`
    Filters attached files directly passed to the model. Depending on the model, this may include Audio or Images, in addition to the default text.
42. **NativeOutputFiles**: `TAiFileCategories`
    Specifies the desired format for the response. Currently, only OpenAI supports Text and Audio response formats, available exclusively with certain audio-specific models. It will throw an error for other models.

Here is the translated section about the **TAiChat Component Events**:

TAiChat Component Events

1. **TAiOpenChatDataEvent**:
   This event is triggered when any data is received during the interaction with the AI model. It is particularly useful for processing each fragment of information coming from the model, especially in asynchronous operations.

```
TAiOpenChatDataEvent = procedure(const Sender: TObject; aMsg:
TAiChatMessage; aResponse: TJSonObject; aRole, aText: String) of
object;
```

   Common Use Cases:

   - Capture partial responses while processing in asynchronous mode.
   - Update user interfaces as data is received.

---

2. **TAiOpenChatBeforeSendEvent**:
   Triggered just before sending a message to the AI model. It allows developers to modify or validate the message before sending it.

```
TAiOpenChatBeforeSendEvent = procedure(const Sender: TObject; var aMsg:
TAiChatMessage) of object;
```

Common Use Cases:

- Validate and fine-tune the message before sending it to the model.
- Modify the message content based on custom logic.

---

3. **TAiOpenChatInitChat**:
   This event is called when a new chat is initiated. It allows setting up and modifying initial instructions or adjusting the chat memory.

```
TAiOpenChatInitChat = procedure(const Sender: TObject; aRole: String;
var aText: String; var aMemory: TJSonObject) of object;
```

Common Use Cases:

- Set the initial context of a conversation.
- Customize the chat memory with specific data for the session being started.

---

4. **TAiOpenChatOnMediaFile**:
   Manages the processing of media files before they are sent to the LLM (Large Language Model). It can be used to manipulate or convert files into other formats.

```
TAiOpenChatOnMediaFile = procedure(const Sender: TObject; Prompt:
String; MediaFile: TAiMediaFile; var Respuesta: String; var
aProcesado: Boolean) of object;
```

Common Use Cases:

- Convert images to text using OCR before sending them to the model.
- Perform preliminary analysis of audio files.

---

5. **TAiOpenChatDataEnd**:
   This event is triggered when the data reception from the AI model is completed. It is useful for wrapping up operations that depend on the complete reception of data.

```
TAiOpenChatDataEnd = procedure(const Sender: TObject; aMsg:
TAiChatMessage; aResponse: TJSonObject; aRole, aText: String) of
object;
```

Common Use Cases:

- Execute final actions once the model's entire response is received.
- Perform status updates or clean-up after the interaction.

---

6. **OnCallToolFunction**:
   Called when a specific tool defined by the AI model needs to be executed. This allows implementing callbacks invoked by the model.

```
TOnCallToolFunction = procedure(Sender: TObject; AiToolCall:
TAiToolsFunction) of object;
```

Common Use Cases:

- Execute specific functions requested by the model.
- Integrate business logic or process specific commands.

Here’s the translation of the section:

How to Get Started?

We'll begin with the basics: creating a request and obtaining a response. However, the components offer various ways to achieve the same goal. Let’s review:

Example 1: AddMessage and Run Function

In this first example, a message of type "user" is added to the chat, and then it is executed to obtain a response. We use the method:

```
AddMessage(aPrompt, aRole: String): TAiChatMessage
```

This method only adds a message to the chat memory, which is then executed using the `Run` function to get the model's response.

```
var
```

```
  Chat: TAiChat;
begin
  Chat := TAiChat.Create(nil);
  try
    Chat.ApiKey := 'your-api-key';
    Chat.Model := 'gpt-4o';
    Chat.AddMessage('What is the capital of France?', 'user');
    ShowMessage(Chat.Run);
  finally
    Chat.Free;
  end;
end;
```

Example 2: NewMessage and Run Function

In this example, a message of type "user" is created, but it is not added to the chat. Instead, it is executed directly. We use the method:

```
NewMessage(aPrompt, aRole: String): TAiChatMessage
```

The resulting message is passed to the `Run` function, which adds the message to the chat before executing it.

```
var
  Chat: TAiChat;
  Msg: TAiChatMessage;
begin
  Chat := TAiOpenChat.Create(nil);
  try
    Chat.ApiKey := 'your-api-key';
    Chat.Model := 'gpt-4o';
    Msg := Chat.NewMessage('What is the capital of France?', 'user');

    // ...you can handle the message here before sending it to the model...
    Msg.LoadMediaFromFile('path/to/file1.jpg');
    Msg.LoadMediaFromFile('path/to/file2.jpg');

    ShowMessage(Chat.Run(Msg));
  finally
    Chat.Free;
  end;
end;
```

Example 3: AddMessageAndRun Function

In this example, the message is created and executed in a single step, directly returning the result. The message is automatically added to the chat.

```
var
  Chat: TAiChat;
  Res: String;
begin
  Chat := TAiOpenChat.Create(nil);
  try
    Chat.ApiKey := 'your-api-key';
    Chat.Model := 'gpt-4o';

    Res := Chat.AddMessageAndRun('What is the capital of France?', 'user', []);
    ShowMessage(Res);
  finally
    Chat.Free;
  end;
end;
```

Example 4: Directing the Chat Conversation

The chat can take various directions because its responses are influenced by statistical and random factors. To control this, you can use properties like `Seed` and `Temperature`.

Additionally, you can guide the chat's responses by simulating a prior conversation, essentially "making it believe" that certain responses were already given.

```
var
  Chat: TAiChat;
  Prompt, Res: String;
begin
  Chat := TAiOpenChat.Create(nil);
  try
    Chat.ApiKey := 'your-api-key';
    Chat.Model := 'gpt-4o';

    Prompt := 'Let us assume we live in another universe where mathematics '+
              'has mutated. All constants have an extra point. For example, the '+
              'square root of 2 is not 1.4142... but 2.4141.... Human body '+
              'temperature is no longer 37 degrees but 38 degrees. Similarly, '+
              'all operations have an extra point; for instance, 2+2=5. '+
              'Please do not explain this new universe, simply answer questions '+
              'without referencing it.';

    Chat.AddMessage(Prompt, 'user');

    // Here we manually add the model's desired response
    Chat.AddMessage('Understood. I will operate within this new universe and not
reference it.', 'assistant');

    Res := Chat.AddMessageAndRun('What is the value of pi?', 'user', []);
    ShowMessage(Res);
  finally
    Chat.Free;
  end;
end;
```

Result:
This will return the response:
The value of pi is 4.1416...

Asynchronous Mode

So far, the invocation mode has been synchronous. This means that we make the call to the LLM and wait until the model fully responds, which in some cases may not be convenient, as it could take several seconds to complete the task.

There are two approaches to address this issue:

1. **Using a thread for the call and waiting for the complete response:**
   In this method, the call is made within a separate thread, allowing the application to remain responsive. The interface stays active while waiting for the response. The advantage of this method is that the user experience is not interrupted by the processing time.
2. **Receiving partial responses as they are generated:**
   In this method, instead of waiting for the full response, the model sends portions of the response as it generates them. This enhances the user experience, even though the total response time remains roughly the same.

Both approaches ensure that the application remains interactive, but the second option can provide a more dynamic user experience.

Using Threads for Chat Calls

If you want to make a call within a separate thread (using the first model), the recommendation is to use one of the thread mechanisms. In this case, we'll work with the `TTask` class, which provides a way to handle asynchronous operations.

Here's how you can implement a threaded call using `TTask.Run`:

```
TTask.Run(
  procedure
  begin
    // Code to execute inside the thread

    TThread.Synchronize(nil,
      procedure
      begin
        // Code to execute in the UI thread (interface updates)
      end);

    // Code to execute inside the thread after updating the interface

  end);
```

Implementation Example with TTask

This is an example of a thread-based call using `TTask.Run`. However, note that this is not exactly the asynchronous mode of the chat, which we'll explain in the next section.

```
TTask.Run(
  procedure
  var
    Chat: TAiChat;
    Prompt, Res: String;
  begin
    Chat := TAiChat.Create(nil);
    try
      Chat.ApiKey := 'your-api-key';
      Chat.Model := 'gpt-4o';
      Chat.InitialInstructions.Text := 'You are an expert in Delphi
programming (latest version)';
      Chat.AddToMemory('Username', 'Gustavo');
      Chat.AddToMemory('AssistantName', 'SofIA');
      Chat.AddToMemory('Hobbies', 'Programming, watching TV');

      Prompt := 'Hello, what is your name, and what is my name?';

      // Send the message and get the response
      Res := Chat.AddMessageAndRun(Prompt, 'user', []);

      // Synchronize to update the UI from the background thread
      TThread.Synchronize(nil,
        procedure
        begin
          MemoResponse.Lines.Text := Res;  // Update the memo with the
response
        end);

    finally
      Chat.Free; // Free the Chat object after use
    end;
  end);
```

Explanation

In the code above:

- **TTask.Run:** This creates a background task where the AI chat interaction is processed.
- **TThread.Synchronize:** This ensures that UI updates (such as modifying the `MemoResponse` component) are performed in the main thread, as UI components can only be updated from the main thread.
- **AddMessageAndRun:** This method adds a message to the chat and runs it, synchronously within the background thread. However, note that this is still not the "real" asynchronous chat mode described later.

Important Notes

- While the approach using `TTask` and `TThread.Synchronize` can keep the UI responsive, it still doesn't provide the full benefits of real asynchronous processing, such as receiving partial responses as they are generated by the model.
- This is a basic way to offload the chat processing to a background thread, but the application will still wait for the entire response before proceeding, making it somewhat synchronous.

Asynchronous Chat Mode

The **Asynchronous Chat Mode** allows the language model (LLM) to send tokens as they are generated, providing real-time feedback to the user. In this mode, the model sends partial responses as they are produced, allowing for a more interactive experience. This is especially useful in cases where the response time may be long, providing users with ongoing updates instead of waiting for the entire response to be completed.

However, it's important to note that this asynchronous mode might not be compatible with the execution of functions (Tools) in some models. Therefore, it's essential to consult the documentation of each model when using this mode to ensure compatibility.

How to Enable Asynchronous Mode

To use the asynchronous mode, you need to set the property `Chat.Asynchronous := True;`. Once this is done, the response will be sent through the `OnReceiveData` and `OnReceiveDataEnd` events.

Events Overview

1. **OnReceiveData Event:**
   - This event is triggered each time a token is received from the model.
   - The application must be responsible for assembling the full message as the tokens arrive. Each time a new token is received, it can be appended to the message being constructed.
2. **OnReceiveDataEnd Event:**
   - This event is triggered once the entire response has been received and processed. At this point, the full message is available, and the application can handle the complete response.

Detailed Process:

1. **Set Asynchronous Mode:** First, set the `Asynchronous` property to `True` to enable this mode.

```
Chat.Asynchronous := True;
```

2. **Handle the OnReceiveData Event:** The `OnReceiveData` event will be triggered each time a token is received. You can handle it like this:

```
procedure TForm1.AiOpenChatOnReceiveData(Sender: TObject; aMsg: TAiChatMessage;
aResponse: TJSONObject; aRole, aText: string);
begin
  // Accumulate tokens and construct the response progressively
  MemoResponse.Lines.Text := MemoResponse.Lines.Text + aText;
end;
```

In this event, the text (`aText`) contains the current token received from the model. The `MemoResponse.Lines.Text` is updated progressively, adding the new token to the previous ones.

3. **Handle the OnReceiveDataEnd Event:** Once the entire response has been received, the `OnReceiveDataEnd` event will be triggered. You can handle this event to finalize the response:

```
procedure TForm1.AiOpenChatOnReceiveDataEnd(Sender: TObject; aMsg: TAiChatMessage;
aResponse: TJSONObject; aRole, aText: string);
begin
  // Finalize the response once all tokens have been received
  ShowMessage('Complete response: ' + MemoResponse.Lines.Text);
end;
```

   - At this point, `MemoResponse.Lines.Text` contains the complete response that can be displayed to the user.

Code Example:

Here's a full example using the asynchronous mode:

```
procedure TForm1.StartAsyncChat;
begin
  Chat := TAiOpenChat.Create(nil);
  try
    Chat.ApiKey := 'your-api-key';
    Chat.Model := 'gpt-4o';
    Chat.Asynchronous := True;

    // Set up event handlers
    Chat.OnReceiveData := AiOpenChatOnReceiveData;
    Chat.OnReceiveDataEnd := AiOpenChatOnReceiveDataEnd;

    // Send the initial prompt
    Chat.AddMessage('What is the capital of France?', 'user');

  finally
    Chat.Free;
  end;
end;
```

```
procedure TForm1.AiOpenChatOnReceiveData(Sender: TObject; aMsg:
TAiChatMessage; aResponse: TJSONObject; aRole, aText: string);
begin
  // Update the response progressively as tokens are received
  MemoResponse.Lines.Text := MemoResponse.Lines.Text + aText;
end;

procedure TForm1.AiOpenChatOnReceiveDataEnd(Sender: TObject; aMsg:
TAiChatMessage; aResponse: TJSONObject; aRole, aText: string);
begin
  // Finalize the response once all tokens are received
  ShowMessage('Complete response: ' + MemoResponse.Lines.Text);
end;
```

Benefits of Asynchronous Mode:

1. **Real-time Feedback:** The user gets a sense of progress as tokens are being received and assembled into a response, instead of waiting for the full response to come in one go.

2. **Improved User Experience:** Especially useful in scenarios where the response may take a while, this approach helps keep the user engaged and provides feedback as soon as possible.
3. **UI Responsiveness:** Since the chat operations are handled asynchronously, the user interface remains responsive, and the application can handle other tasks simultaneously.

By enabling asynchronous mode and using the `OnReceiveData` and `OnReceiveDataEnd` events, you can create a more interactive, responsive chat experience for users.

Other example:

```
Var
  Prompt, Res: String;
  Msg: TAiChatMessage;
Begin

  If Not Assigned(Chat) then
    Chat := TAiOpenChat.Create(nil);

  try
    Chat.ApiKey := 'Mi api key;
    Chat.Model := 'gpt-4o';
    Chat.InitialInstructions.Text := 'Eres un experto en el lenguaje Delphi última
versión';
    Chat.Asynchronous := True;
    Chat.OnReceiveData := AiConnReceiveData;

    Prompt := 'Dime porqué debería utilizar Delphi y en que casos es mejor que otros
lenguajes?';

    MemoResponse.Lines.Add('user: ' + Prompt);
    MemoResponse.Lines.Add('');
    MemoResponse.Lines.Add('assistant: ');

    If Chat.Asynchronous = False then
    Begin
      Res := Chat.AddMessageAndRun(Prompt, 'user', []);
      MemoResponse.Lines.Text := MemoResponse.Lines.Text + Res;
    End
    Else
    Begin
      Chat.AddMessageAndRun(Prompt, 'user', []);
    End;

  finally
    // Chat.Free;
  end;
```

In the event, we are adding the text of each token to a memo

```
procedure TForm75.AiConnReceiveData(const Sender: TObject; aMsg: TAiChatMessage;
aResponse: TJSONObject; aRole, aText: string);
begin
  TThread.Synchronize(nil,
    procedure
    begin
      MemoResponse.BeginUpdate;
      Try
        MemoResponse.Lines.Text := MemoResponse.Lines.Text + aText;
        MemoResponse.SelStart := Length(MemoResponse.Text);
      Finally
        MemoResponse.EndUpdate;
      End;
    end);
end;
```

If you don't want to add each token individually, you can use the OnReceiveDataEnd event, which is triggered only once when the query is complete and returns the full text. You can also use these events to modify the graphical interface by enabling or disabling objects like buttons, memos, etc. The component has the Chat.Busy property, which will always be true while it's executing and false when it's available.

```
. . .
    Chat.Model := 'gpt-4o';
    Chat.InitialInstructions.Text := 'Eres un experto en el lenguaje Delphi última
versión';
    Chat.Asynchronous := True;
    //Chat.OnReceiveData := AiConnReceiveData;
    Chat.OnReceiveDataEnd := AiOpenChat1ReceiveDataEnd;
. . .
```

```
procedure TForm75.AiOpenChat1ReceiveDataEnd(const Sender: TObject;
  aMsg: TAiChatMessage; aResponse: TJSONObject; aRole, aText: string);
begin
  TThread.Synchronize(nil,
    procedure
    begin
      MemoResponse.BeginUpdate;
      Try
        MemoResponse.Lines.Text := MemoResponse.Lines.Text + aText;
        MemoResponse.SelStart := Length(MemoResponse.Text);
      Finally
        MemoResponse.EndUpdate;
      End;
    end);
end;
```

Memory Management

The components manage 3 types of memory that affect the behavior of responses based on the context created by these data. The first is the initial message, which defines its initial behavior. The second is the information stored in a property called *Memory*, where key-value pairs are stored. Finally, there is the list of messages, which corresponds directly to the conversation.

Initial Message

In most LLMs, there is an initial message that instructs the model on how to behave. By default, this message is typically something like "You are a helpful and useful assistant." However, in many cases, it is necessary to specify it more precisely, for example, "You are an expert in Delphi application development, latest version," etc.

```
Property InitialInstructions: TStrings read FInitialInstructions write SetInitialInstructions;
```

```
var
  Chat: TAiChat;
  Prompt, Res : String;
```

```
begin
  Chat := TAiOpenChat.Create(nil);
  try
    Chat.ApiKey := 'Api Key';
    Chat.Model := 'gpt-4o';
    Chat.InitialInstructions.Text := ‘you are a Delphi expert (Last version);

    Prompt := 'create a function to calc the Fibonacci serie';

    Res := Chat.AddMessageAndRun(Prompt,'user',[]);
    ShowMessage(Res);

  finally
    Chat.Free;
  end;
```

Memory Property

The *Memory* property allows storing key information that can affect the behavior of the LLM. This memory is persistent across different chat sessions, meaning it remains even when starting a new chat, unlike messages, which are not retained.

```
var
  Chat: TAiChat;
  Prompt, Res : String;
begin
  Chat := TAiOpenChat.Create(nil);
  try
    Chat.ApiKey := 'Api Key';
    Chat.Model := 'gpt-4o';
    Chat.InitialInstructions.Text := 'Eres un experto en el lenguaje Delphi última
versión';
    Chat.AddToMemory('NombreUsuario','Gustavo');
    Chat.AddToMemory('nombre del Asistente Virtual','SofIA');
    Chat.AddToMemory('Hobbies','Programar, ver TV');

    Prompt := 'Hola, como te llamas tu y como me llamo yo?';

    Res := Chat.AddMessageAndRun(Prompt,'user',[]);
    ShowMessage(Res);

  finally
    Chat.Free;
  end;
```

To remove an entry from memory, you can execute the procedure:
`Procedure RemoveFromMemory(Key: String);`

To remove all entries from memory, you can use:
`Chat.Memory.Clear;`

Chat Messages
The third type of memory managed is the conversation messages. Each message, whether from the user or the assistant, is added to a list. For each request, all the messages must be sent, including the initial message, memory entries, and chat messages.

There are several ways to add messages to the list. Here are some ways to manage this list:

1. **Adding a message without executing it:** `Chat.AddMessage(Prompt, Role);`
   This option allows adding messages as either "user" or "assistant". Remember that a chat alternates between messages from both roles.
2. **Another option is to create the message and then add it to the list.**
   You can create and add as many messages as needed, simulating the conversation between the "user" and the "assistant". At the end, you can execute the function `Chat.Run;`.

   The function `TAiChat.Run(Msg : TAiChatMessage = Nil);` has an optional parameter. If the parameter is `Nil` or not provided, it runs with the messages already in the list. Otherwise, it adds the passed message to the list before executing the command.

```
Var Msg : TAiChatMessage;
  Msg := Chat.NewMessage('¿your prompt here?', 'user');
  Msg.LoadMediaFromFile('ruta/del/archivo1.jpg');
  Chat1.Run(Msg)
```

3. **As a third option, there is the "AddMessageAndRun" function**, which performs both steps in a single function. It creates the message, adds it to the list, and then executes the "Run" function.

```
var
  Chat: TAiChat;
  Res : String;
begin
  Chat := TAiOpenChat.Create(nil);
  try
    Chat.ApiKey := api key';
    Chat.Model := 'gpt-4o';

    Res := Chat.AddMessageAndRun('your prompt here?','user',[]);
    ShowMessage(Res);

  finally
    Chat.Free;
  end;
```

Operations with Messages

1. **NewChat: Delete all messages**
   To delete all messages and start a new chat, simply execute the function `Chat1.NewChat;`. This option deletes all messages, but does not remove the initial message or the content of the "Memory" property.

2. **GetLastMessage: This function returns a TAiChatMessage object**
   This object is the last message in the message list. The message can be deleted using either the object itself or the message's Id.

```
    Msg := Chat.GetLastMessage;
    Chat.RemoveMesage(Msg);
    O de esta forma
    Chat.RemoveMesage(Msg.Id);
```

Memory Optimization Techniques

Now that we understand how memory works, we can see that each message increases the message list, and soon it can exceed the model's context window. Additionally, each call increases the number of tokens consumed, so it might be a good technique to summarize the messages.

The list of messages sent to the model would look similar to this JSON:

```
 [
     {
          "role": "system",
          "content": "You are an expert in the latest version of the Delphi
language.\r\n\r\n\r\nTo Remember= {\r\n    \"UserName\": \"Gustavo\",\r\n
\"AssistantName\": \"SofIA\",\r\n    \"Hobbies\": \"Programming, watching TV\"\r\n}"
     },
     {
          "role": "user",
          "content": "Hello, what is your name and what is mine?"
     },
     {
          "role": "assistant",
          "content": "Hello! I am SofIA and your name is Gustavo. How can I assist you
today?"
     }
]
```

There are several techniques to summarize the message content. The one we will work with here consists of summarizing the most relevant ideas and removing what doesn't add value to the conversation.

To implement this solution, a second component will be used to handle the text conversion:

```
var
  Messages, Res: String;
begin
  Messages := Chat1.Messages.ToJson.Format;
  Chat2.Messages.Clear;

  Res := Chat2.AddMessageAndRun('Summarize the following JSON, remove
what is irrelevant and repeated. Message: ' + Messages, 'user', []);

  Chat1.Messages.Clear;
  Chat1.AddMessage('Keep in mind this information: ' + Res, 'user');
  Chat1.AddMessage('Understood, I will keep that in mind for future
queries', 'assistant');
```

This approach reduces the size of the message list, ensuring that only relevant content remains and improving memory usage and token consumption.

Media File Handling

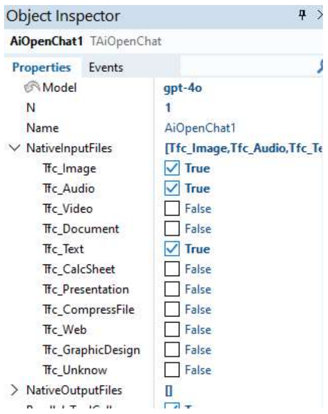
There are many LLM models, some specialized in code, others focused exclusively on text, and even multimodal ones. In some cases, there is a need to mix these models to get the best of each. While multimodal models are becoming more common, it is still necessary to synchronize the use of different models.

Multimedia - Images

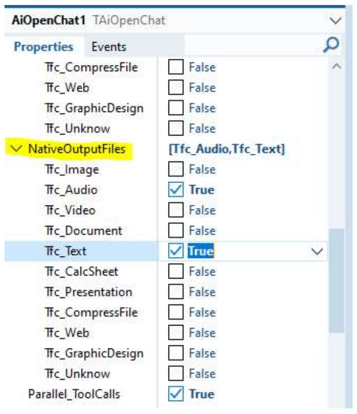
If the model is multimodal, there is no need to do anything different than what we've seen so far. However, it’s important to note that some models, like Llama 3.2, only accept one message in the chat, and that message can contain only one image.

As of December 2024, a filter has been added to select the types of files that are sent directly to the model through the `NativeInputFiles` property. If you want to pass image files directly to the model, you would mark `tfc_image`, and the system will include the image in base64 format in the request. However, if this is not marked, the system calls the `OnProcessMediaFile` method to preprocess the image, allowing the developer to process the file and return the equivalent of that file in the request. For example, if it is an audio file, it can be converted to text and sent to the model as text.

This provides flexibility for the developer to handle media files, ensuring they are properly formatted for the specific model being used.



Model	gpt-4o
N	1
Name	AiOpenChat1
NativeInputFiles	[Tfc_Image,Tfc_Audio,Tfc_Te
Tfc_Image	✓ True
Tfc_Audio	✓ True
Tfc_Video	False
Tfc_Document	False
Tfc_Text	✓ True
Tfc_CalcSheet	False
Tfc_Presentation	False
Tfc_CompressFile	False
Tfc_Web	False
Tfc_GraphicDesign	False
Tfc_Unknow	False
NativeOutputFiles	[]



Tfc_CompressFile	False
Tfc_Web	False
Tfc_GraphicDesign	False
Tfc_Unknow	False
NativeOutputFiles	[Tfc_Audio,Tfc_Text]
Tfc_Image	False
Tfc_Audio	✓ True
Tfc_Video	False
Tfc_Document	False
Tfc_Text	✓ True
Tfc_CalcSheet	False
Tfc_Presentation	False
Tfc_CompressFile	False
Tfc_Web	False
Tfc_GraphicDesign	False
Tfc_Unknow	False
Parallel_ToolCalls	✓ True

Similarly, some models now allow returning different file formats. Specifically, the OpenAI model can return both Text and Audio. If you wish to use this feature, you must mark both Audio and Text in the `NativeOutputFiles` property. It's important to clarify that this filter currently only works with OpenAI's audio model. For other models, it has no effect yet.

This feature enables the developer to receive multimedia responses from the model, which can be especially useful in applications where both text and audio outputs are required.

```
var
  Chat: TAiChat;
  Msg : TAiChatMessage;
begin
  Chat := TAiOpenChat.Create(nil);
  try
    Chat.ApiKey := 'api key';
    Chat.Model := 'gpt-4o';
    Msg := Chat.NewMessage('¿you can see this image?', 'user');

    //...here can handle the msg before sending...
    Msg.LoadMediaFromFile('ruta/del/archivo1.jpg');
    Msg.LoadMediaFromFile('ruta/del/archivo2.jpg');

    ShowMessage(Chat.Run(Msg));
  finally
    Chat.Free;
  end;
```

When the main model is not multimodal, a second model that is multimodal can be used to complement the main model. In this case, the `OnProcessMediaFile` event is used, allowing the developer to obtain the binary content of the file, preprocess it, and pass the equivalent text to the request.

This approach is particularly useful for handling media files such as images, audio, or other types of content that the main model may not be able to process directly. By preprocessing the media file using the multimodal model, you can convert the file into a text representation (e.g., extracting text from an image using OCR or transcribing speech from an audio file) and send that text to the main model for further processing.

```
procedure TForm75.AiOpenChat1ProcessMediaFile(const Sender: TObject; Prompt: string;
MediaFile: TAiMediaFile;
    var Respuesta: string; var aProcesado: Boolean);
var
  Chat: TAiChat;
  Res : String;
begin
  Chat := TAiOpenChat.Create(nil);
  try
    Chat.ApiKey := 'api key';
    Chat.Model := 'gpt-4o';
    Res := Chat.AddMessageAndRun('¿you can see this image?', 'user', [MediaFile]);

    Respuesta := Res; //return the image description
    aProcesado := True; //tell to model alredy has been processed
  finally
    Chat.Free;
  end;
end;
```

Multimedia - Audios

We can also preprocess audio files using the **TAiAudio** component, which utilizes OpenAI's **Whisper model**. Whisper is also available on other platforms and can even be downloaded locally since it is open-source.

Preprocessing Audio to Text

First, attach the audio file in the same way as you would with an image. Here is an example:

```
var
  Chat: TAiChat;
  Msg : TAiChatMessage;
begin
  Chat := TAiOpenChat.Create(nil);
  try
    Chat.ApiKey := 'api key';
    Chat.Model := 'gpt-4o';
    Msg := Chat.NewMessage('¿resume this audio?', 'user');

    Msg.LoadMediaFromFile('path/to/myaudio.wav');

    ShowMessage(Chat.Run(Msg));
  finally
    Chat.Free;
  end;
end;
```

Using the `OnProcessMediaFile` Event to Convert Audio to Text

To convert the audio file to text, you can use the `OnProcessMediaFile` event. Here's an example of how to handle the audio:

```
var
  Res: String;
begin

  Case MediaFile.FileCategory of

    TAiFileCategory.Tfc_Image:
      Begin
        var
          Chat: TAiChat;
        Chat := TAiOpenChat.Create(nil);
        try
          Chat.ApiKey := 'api key';
          Chat.Model := 'gpt-4o';
          Res := Chat.AddMessageAndRun('¿you can see this image?', 'user', [MediaFile]);
          Respuesta := Res;
          aProcesado := True;
        finally
          Chat.Free;
        end;
      End;

    TAiFileCategory.Tfc_Audio:
      Begin
        var
          Audio: TAiAudio;
        Audio := TAiAudio.Create(nil);
        try
          Audio.ApiKey := 'your-api-key';
          Audio.Model := 'whisper-1';
          Res := Audio.Transcription(MediaFile.Content, MediaFile.FileName, 'Transcribe
the audio');
          Respuesta := Res;
          aProcesado := True;

        finally
          Audio.Free;
        end;
      End;
  End;
end;
```

Handling Other Media Types

Similarly, you can process videos, PDFs, etc., as long as you have the necessary tools to convert these file types into text. It's important to note that this process does not implement a **RAG model** (Retrieval-Augmented Generation). For that, refer to the corresponding manual for RAG, which also provides components for implementation.

Tools – Execute Functions

One of the most valuable features developed for LLMs (Large Language Models) is **function execution**. This capability connects AI to the real world, allowing the AI to interact with applications, the Internet of Things (IoT), and generally giving "arms" to the artificial brain.

It's important to note that not all models are equipped to handle this functionality, so be sure to read the documentation of each platform to identify the appropriate model for your needs. For OpenAI, this functionality is available through **API Tools**. With the integration of external functions, an LLM can perform tasks like:

- **Search real-time information**: The model can query external sources such as databases or search engines.
- **Manipulate data**: For example, process images, files, or perform complex mathematical calculations.
- **Software interaction**: It can make HTTP requests, integrate with external systems, invoke APIs, etc.

This feature is especially useful when an LLM needs to perform tasks beyond generating text. For example, it can invoke functions to retrieve specific data, execute system commands, generate reports, interact with third-party services, or query complex systems.

Preparation for Function Execution

The most crucial aspect of executing functions as part of an LLM's **Tools** is ensuring the model clearly understands when to execute a function and what it should do. Unlike what many believe, it is essential to clearly detail each function in natural language, providing all the necessary instructions so that the model doesn't make mistakes regarding the functionality of a procedure or the parameters required for its execution.

Example Function

To describe a function in the context of **function calling** in LLMs, it's essential to provide a clear structure that specifies what the function does, what parameters it takes, and what result it returns. Here's an example of how to define a function, in this case, one that returns the current system date and time.

- **Function Name**: `Get_Fecha_Actual`
- **Function Description**: The function `Get_Fecha_Actual` returns the current date and time in a specific geographic location. It is used to get the date and time localized to a given city's or country's time zone.
- **Parameter**: `Localizacion` corresponds to the city, country, or time zone from which we need the date. This parameter is optional, and if not provided, it defaults to Colombia's city.

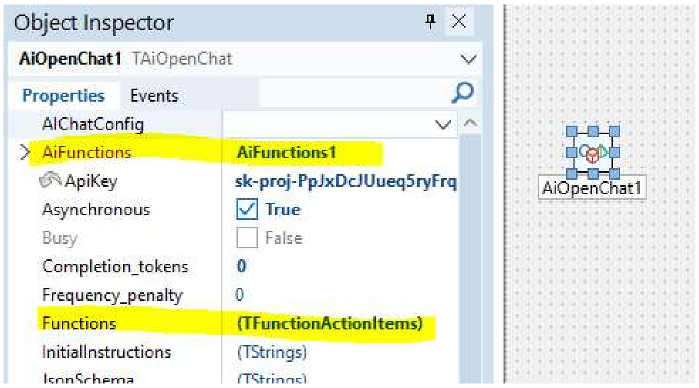
Note that the function name is clear and understandable for both humans and the AI.

Function Implementation

There are two ways to implement functions in this context:

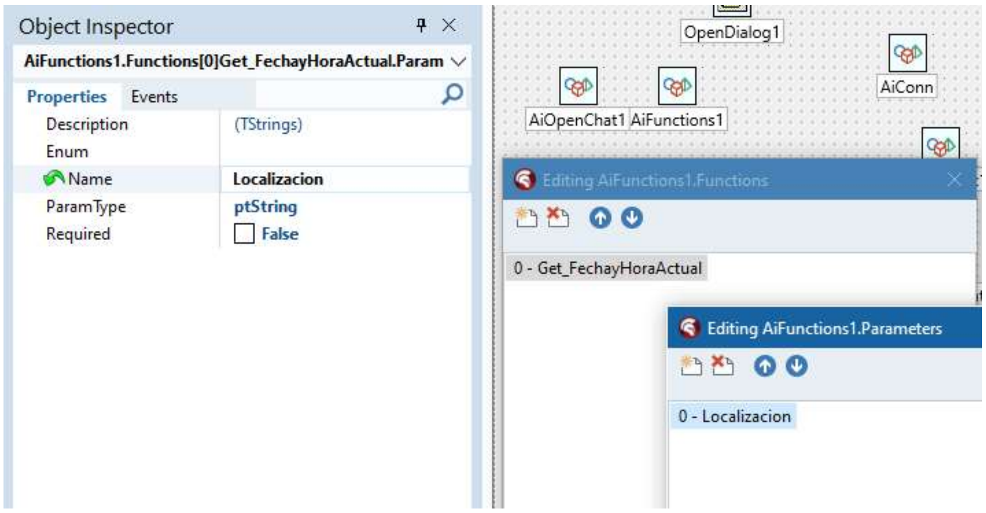
1. **TAiChat.Function property**: This property allows you to define a function specifically for that component.
2. **TAiFunctions component**: This is a separate component that can be associated with `TAiChat` through the `AiFunctions` property. Both methods function similarly, but the advantage of the second method is that the `TAiFunctions` component can be shared across different objects, allowing the same function to be used by multiple `TAiChat` components.

The logic is simple: if the `AiFunctions` property is assigned, the component will execute the functions defined in the `TAiFunctions` component. If this property is `Nil`, the component will execute the functions defined in the `TAiChat.Functions` property, which are the local functions.



AIChatConfig	
AiFunctions	AiFunctions1
ApiKey	sk-proj-PpJxDcJUueq5ryFrq
Asynchronous	✓ True
Busy	False
Completion_tokens	0
Frequency_penalty	0
Functions	(TFunctionActionItems)
InitialInstructions	(TStrings)
JsonSchema	(TStrings)

For this example, we will use the `TAiFunctions` component and associate it with the `TAiChat.AiFunctions` property, allowing us to share it across different components. First, we will define the function in the `TAiFunctions` component within the `Functions` property. Since it's a collection, we will do this through the visual environment.



Object Inspector
AiFunctions1.Functions[0]Get_FechayHoraActual.Param
Properties
Events
OpenDialog1
AiOpenChat1
AiFunctions1
AiConn
Editing AiFunctions1.Functions
0 - Get_FechayHoraActual
Editing AiFunctions1.Parameters
0 - Localizacion

Description	(TStrings)
Enum	
Name	Localizacion
ParamType	ptString
Required	False

In the property, we add a new function and change the `Name` property to a clear name for the function, in this example, it would be `Get_FechayHoraActual`. This should be accompanied by a description that is clear and concise.

Next, we define the parameters. In this example, it would be `Localización`, which should also have a clear description and indicate whether the parameter is required or if it's of an enumerated type, for which you can create a comma-separated list.

Finally, the function will have an event that will be triggered when the LLM determines it should be used.

```
procedure TForm75.AiFunctions1Functions0Get_FechayHoraAction(Sender: TObject;
              FunctionAction: TFunctionActionItem; FunctionName: string;
              ToolCall: TAiToolsFunction; var Handled: Boolean);
Var
  Locacion : String;
begin
  Locacion := ToolCall.Params.Values['Localizacion'];
  ToolCall.Response := FormatDateTime('YYYY-MM-DD hh:nn:ss', Now);
  Handled := True;
end;
```

In this example, the response will simply return the system's date and time, but in reality, it's possible to perform quite complex procedures, and this is the basis of OpenAI's famous GPTs.

It should be noted that some models, such as those from OpenAI, can execute functions in parallel, handling threads. Therefore, it's necessary to program these functions with multithreading execution in mind.

IMPORTANT NOTE: None of the events are protected for background execution or with thread concurrency protection. Therefore, the user must handle these situations.
Example:

1. If accessing databases, a connection component should be created each time the function is called, as TFDConnection, for example, does not support concurrent handling and may cause execution deadlocks.
2. The handling of the visual interface should always be done within a protected function such as `TThread.Synchronize` or `TThread.Queue`, as it may interfere with the main thread flow and cause application deadlocks.
3. If multiple functions or even the same function access global variables, semaphores will be necessary to avoid concurrency and errors for this reason.

Events of the TAiChat Component

In this section, we will provide a brief overview of each of the events that we haven't covered yet in this manual and how to use them.

OnInitChat

This event is used to intercept the initialization of the LLM (Language Learning Model). It only executes once automatically before the first message is added to the message list. It's the moment where you can modify both the initial message and the TAiChat.Memory property.

```
Chat.InitialInstructions.Text := 'You are an expert in Delphi
language';
Chat.AddToMemory('My_Name', 'Gustavo');
Chat.AddToMemory('Assistant_Name', 'SofIA');
```

The `aText` parameter in the function would correspond to `Chat.InitialInstructions.Text`, and the `aMemory` parameter would be:

```
{
  "My_Name": "Gustavo",
  "Assistant_Name": "SofIA"
}
```

In this event, it is possible to completely modify the parameters. This case allows you to personalize the model's behavior based on a parameter like the user.

```
procedure TForm75.AiOpenChat1InitChat(const Sender: TObject; aRole:
string;
  var aText: string; var aMemory: TJSONObject);
begin
   aText := 'You are an expert Delphi programmer and also in SQL
databases';
   aMemory.AddPair('Record1','You are obsessive about details');
end;
```

OnAddMessage

This event is triggered whenever a new message is added to the message list. This includes the "System" and "User" messages but not the "Assistant" messages. It is possible to modify some of the parameters of the message (`aMsg`) directly in the parameter.

```
procedure TForm75.AiConnAddMessage(const Sender: TObject; aMsg:
TAiChatMessage;
  aResponse: TJSONObject; aRole, aText: string);
begin
    ShowMessage(aRole + ' : ' + aMsg.Prompt);
end;
```

OnBeforeSendMessage

This event allows you to capture the messages just before they are sent to the LLM. Similar to the `OnAddMessage` event, it allows you to intercept and modify the message, but this event only includes "User" messages.

```
procedure TForm75.AiOpenChat1BeforeSendMessage(const Sender: TObject;
  var aMsg: TAiChatMessage);
begin
   aMsg.LoadMediaFromFile('/my/file.png');
end;
```

OnCallToolFunctions

As we saw in the previous section on LLM function execution, each function has an event that is intended to execute automatically when the need arises. However, if the function defined in the `TAiChat.Functions` property or in the `TAiFunctions.Functions` component does not have an associated event or if the variable `Handled = false`, the system assumes that it was not executed and proceeds to trigger the `OnCallToolFunction` event of the component.

```
procedure TForm75.AiOpenChat1CallToolFunction(Sender: TObject;
  AiToolCall: TAiToolsFunction);
Var
  MiParam : String;
begin
  If AiToolCall.&Function = 'MyFunction' then
  Begin
    MiParam := AiToolCall.Params.Values['MyParam'];
    // here execute the corresponding function
  End;
end;
```

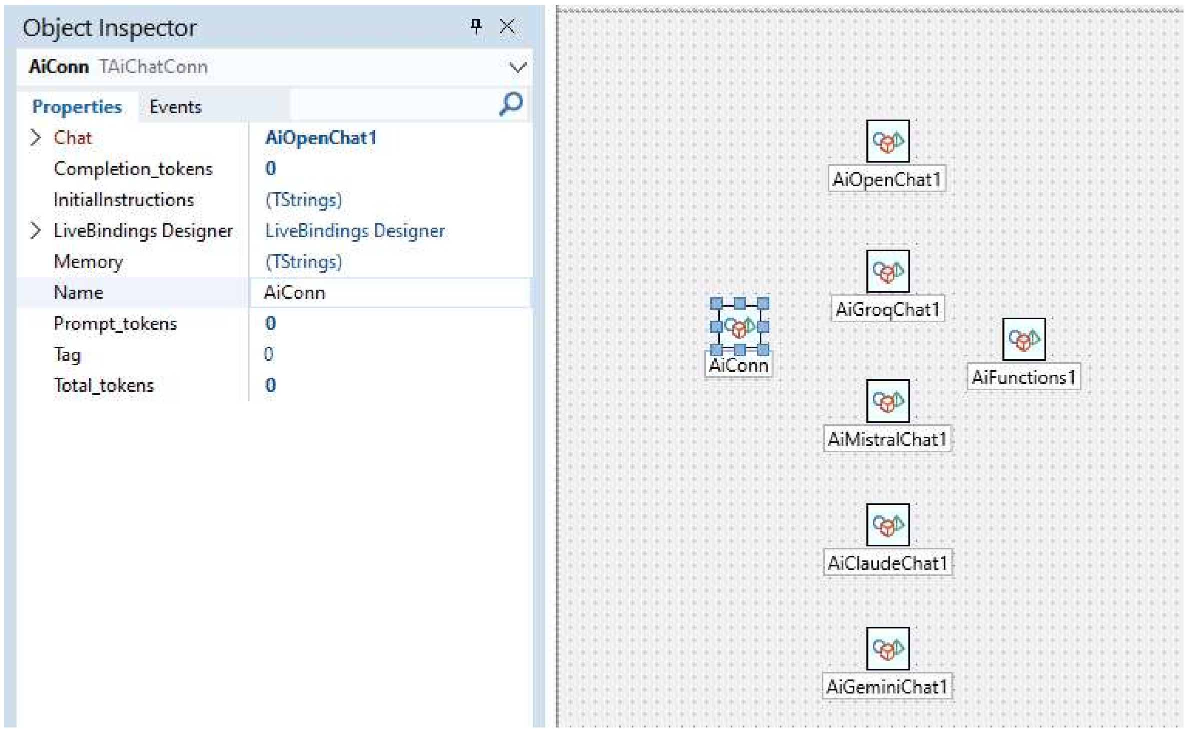
In this example, if the function name (`MyFunction`) matches, the parameter `MyParam` is extracted from the function call, and you can then execute the corresponding logic for the function. This allows you to manage and handle functions that do not automatically trigger their events.

TAiChatConn Component

One of the premises of the MakerAi component suite is the transparency of use across different LLM models. This means that the same code can be used to interact with the various models and services available in the market. A key feature that has been sought is to allow the easy switching of LLMs.

For this purpose, the `TAiChatConn` component has been created, which allows you to "connect" your program transparently to any model by simply changing the `TAiChatConn.Chat` property. This component also includes basic parameters common to all components, thus enabling a single connector for the different LLMs available.

Note: This component is still under development, so some features may not yet be available.



Object Inspector

AiConn TAiChatConn

Properties	Events
Chat	AiOpenChat1
Completion_tokens	0
InitialInstructions	(TStrings)
LiveBindings Designer	LiveBindings Designer
Memory	(TStrings)
Name	AiConn
Prompt_tokens	0
Tag	0
Total_tokens	0

This way, it's possible to interact simply with the `AiConn` without dealing with the other models. For example, to execute a request, it would look like this:

```
Var
  Res: String;
  MediaFile: TAiMediaFile;
  Msg: TAiChatMessage;
begin

  If FileExists(OpenDialog1.FileName) then
  Begin
    MediaFile := TAiMediaFile.Create;
    MediaFile.LoadFromFile(OpenDialog1.FileName);
    // MediaFile.LoadFromUrl(Url);  //Url con el nombre del archivo identificable
    // MediaFile.LoadFromBase64(FileName, Base64Data); //El filename se utiliza para
obtener el tipo de imágen
    // MediaFile.LoadFromStream(FileName, Stream); //El filename se utiliza para obtener
el tipo de imágen
  End;

  If ChIncluirImagen.IsChecked then
    Res := AiConn.AddMessageAndRun(MemoPrompt.Lines.Text, 'user', [MediaFile])
  Else
    Res := AiConn.AddMessageAndRun(MemoPrompt.Lines.Text, 'user', []);

  MemoResponse.Lines.Text := Res;
  FreeAndNil(MediaFile);
```

Delphi MVP - Gustavo Enríquez
 Redes Sociales:
 - Email: gustavoeenriquez@gmail.com
 - Telegram: +57 3128441700

```
 - LinkedIn: https://www.linkedin.com/in/gustavo-enriquez-3937654a/
 - Youtube: https://www.youtube.com/@cimamaker3945
 - GitHub: https://github.com/gustavoeenriquez/
```