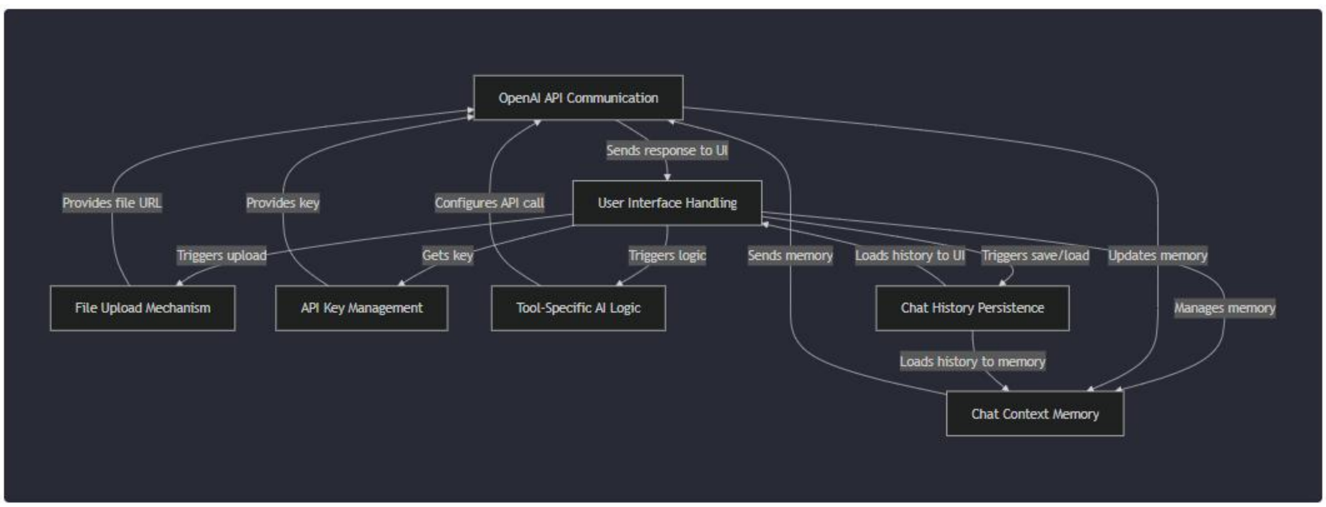
VivacityGPT Complete - Technical and Operational Handbook

The project VivacityGPT Complete is a web application created by **Vivacity Design Web Agency** (Alessandro Demontis) offering various AI-powered tools. It acts as a **central hub** connecting users to powerful external **AI services** like **OpenAI and ChatPDF**. Users can **interact** through a web interface, upload **files** for analysis, manage their **API keys**, and use **tool-specific logic** for tasks like image creation, image and text analysis, text-to speech and speech-to-text, and code generation. Most tools also feature **chat memory** for contextual conversations and **history persistence** to save and load past interactions.

As of Dec. 15th, VivacityGPT Complete has reached version v.18 and consists of 7 main tools and 17 GPTs (specific tools specialized in one or more tasks).

Visual Overview



Chapters of this handbook

1. User Interface Handling
2. API Key Management
3. File Upload Mechanism
4. Tool-Specific AI Logic
5. OpenAI API Communication
6. Chat Context Memory
7. Chat History Persistence

Chapter 1: User Interface Handling

Imagine the UI as the dashboard of a special car. It has everything you need to control the car (like the steering wheel and pedals) and displays important information back to you (like the speedometer and fuel gauge). In our project, the UI is like that dashboard for interacting with the AI tools.

It includes:

- **The layout:** How everything is arranged on the page.
- **Buttons:** Things you click to make something happen (like "Send" or "Upload").
- **Text areas:** Boxes where you can type or see text.
- **File upload inputs:** Places where you can select files from your computer (like images or audio).
- **Output areas:** Where the AI's answers, generated images, or audio players show up.

Our First Example: A Simple Chat

Let's look at a common task: sending a message to the AI and seeing its response. This is the core interaction in tools like the *Text Chatbot* or the *AI Virtual Sommelier*. You type a question or a request, click a button, and the AI's answer appears right there on the screen.

How does this happen using the UI elements?

1. You find the box labeled for typing your message (a **text area**).
2. You type your question into that box.
3. You find the button to send your message (a **button**).
4. You click the button.
5. A moment later, you see your message appear in the conversation history, followed by the AI's answer in the **output area**.

Looking at the Code (Simple View)

Let's peek at some simplified code to see how these parts look in the web page files (they are HTML files, which structure the page, and contain CSS for styling and JavaScript for interactivity). We'll use the *Text Chatbot* as our example base, but many pages use similar structures.

Here's a very basic idea of the HTML parts for this interaction:

```html
<!-- A box for you to type -->
<textarea id="prompt" placeholder="Your prompt"></textarea>

<!-- A button to send your message -->
```

```html
<button onclick="sendMessage()">Send</button>

<!-- The area where the chat history appears -->
<div id="chathistory">
 <!-- Messages will be added here by JavaScript -->
</div>
```

This HTML code sets up the basic elements the user interacts with: a text box to type in, a button to click, and a div (a container) where the conversation will be shown.

Now, what happens when you click the "Send" button? The *onclick="sendMessage()"* part tells the browser to run a JavaScript function named *sendMessage*. This function is the link between the button click and the action we want to take.

Here's a simplified look at what that *sendMessage* JavaScript function does, focusing just on the UI parts:

```javascript
function sendMessage() {
 // 1. Find the text area and get what the user typed
 const inputElement = document.getElementById("prompt");
 const userInput = inputElement.value; // Get the text

 // 2. Find the chat history area
 const chatContainer = document.getElementById("chathistory");

 // 3. Create a new message element for the user's text
 const userMessageElement = document.createElement("p"); // Make a paragraph
 userMessageElement.textContent = `You: ${userInput}`; // Put the text in it

 // 4. Add the user's message to the history area
 chatContainer.appendChild(userMessageElement);

 // (Sending the message to the AI happens next, but we'll cover that later!)

 // 5. Clear the input box after sending
 inputElement.value = "";
}
```

This JavaScript code shows how the browser, when the "Send" button is clicked, can grab the text you typed, create a new HTML element for it, and add it to the visible history area.

When the AI sends back a response (which involves talking to the AI's server, a topic for a later chapter), other JavaScript code will take that response and add it to the *chathistory* div in a similar way, perhaps styled differently to show it came from the AI.

What's Next?

Now you have a basic understanding of how the User Interface works to let you interact with the VivacityGPT Complete tools. But just interacting isn't enough; the UI needs to connect to the powerful AI models that do the actual work. These models often require special keys to access. In the next chapter, we'll dive into how the project handles these important keys: *API Key Management*.

Chapter 2: API Key Management

In Chapter 1: *User Interface Handling*, we explored how the different parts you see and click on help you interact with the project. Now, we're going to look at something just as important, but a little more behind-the-scenes: managing the special "keys" that let the website talk to the powerful AI services.

Think of using AI services like visiting a special members-only club. You need a membership card (an API key) to get in and use the facilities. Services like OpenAI (for generating text and images) and ChatPDF (for chattng with PDFs) are these clubs, and they require you to have your own personal API keys to use them. These services might charge you based on how much you use them, and your API key is how they track that usage to your account.

Since you have to enter your keys to use the AI tools in VivacityGPT Complete, you probably don't want to type them in every single time you open a page, right? That's where **API Key Management** comes in!

What Problem Does API Key Management Solve?

The main problem is convenience and access. You have these important keys, and the website needs them to function, but you want an easy and safe way to provide them once and have the website remember them. Our project solves this by using a simple method called **Local Storage**.

What is Local Storage?

Imagine your web browser has a tiny, private notebook specifically for each website you visit. This notebook is *Local Storage*. When a website wants to remember something small about you for next time (like your preference for dark mode, or in our case, your API keys), it can write it down in this notebook.

- **It's stored in your browser:** The information stays on *your* computer, within your browser's data.

- **It's just for that website:** Only the specific website (www.vivacitydesign.net/vdgptscomplete in this case) can read or write to its own notebook. Other websites cannot access your VivacityGPT Complete local storage.
- **It stays there until you clear it:** Unlike temporary notes (like *Session Storage*), *Local Storage* usually stays saved until you clear your browser's data or the website specifically removes it.

So, in our project, *Local Storage* acts as a simple, built-in "keyholder" right in your browser.

How Does VivacityGPT Complete Use Local Storage for Keys?

Let's look at the main place where you interact with API key management: the home page (*index.html*).

You might have noticed a button or a section on the home page that lets you enter your OpenAI and ChatPDF keys. When you enter your keys there and click "Save Keys", the website takes those keys and stores them in your browser's Local Storage using specific variable names (like *openaikey* and *chatpdfkey*).

Then, when you navigate to another tool page (like the *Text Chatbot* or the *Image Creator*), that page will check for the saved keys. If it finds them, it will use them for the AI requests without you having to type them again.

Looking at the Code (Simple View)

Let's peek at the *index.html* file again to see how this works with simple HTML and JavaScript. Here's a simplified look at the HTML for the API key input area (found within a "modal" or pop-up box on the home page):

```
<!-- Input box for OpenAI Key -->
<input type="text" id="openaikeyInput" placeholder="Enter OpenAI key">

<!-- Input box for ChatPDF Key -->
<input type="text" id="chatpdfkeyInput" placeholder="Enter ChatPDF key">

<!-- Button to save the keys -->
<button id="saveKeys">Save Keys</button>

<!-- Button to clear the keys -->
<button id="resetKeys">Reset Keys</button>
```

This HTML sets up the boxes where you type your keys and the buttons you click.

Now, let's see the simplified JavaScript code that runs when you click these buttons. This code uses the browser's built-in *localStorage* object:

```
// Find the input boxes and buttons
const openaikeyInput = document.getElementById('openaikeyInput');
const chatpdfkeyInput = document.getElementById('chatpdfkeyInput');
const saveKeysBtn = document.getElementById('saveKeys');
const resetKeysBtn = document.getElementById('resetKeys');

// --- What happens when you click "Save Keys" ---
saveKeysBtn.addEventListener('click', () => {
  // Get the text typed into the input boxes
  const openAIKey = openaikeyInput.value;
  const chatPDFKey = chatpdfkeyInput.value;

  // Save the keys in the browser's local storage
  // They are saved under the names 'openaikey' and 'chatpdfkey'
  localStorage.setItem('openaikey', openAIKey);
  localStorage.setItem('chatpdfkey', chatPDFKey);

  // (Maybe show a message or close the modal here)
});

// --- What happens when you click "Reset Keys" ---
resetKeysBtn.addEventListener('click', () => {
  // Remove the keys from local storage
  localStorage.setItem('openaikey', ''); // Setting to empty string effectively removes it for practical purposes here
  localStorage.setItem('chatpdfkey', ''); // Setting to empty string

  // Clear the text in the input boxes
  openaikeyInput.value = "";
  chatpdfkeyInput.value = "";

  // (Maybe show a message here)
});

// --- What happens when the page loads (or modal opens) ---
// This code runs to see if keys are already saved
function checkLocalStorageKeys() {
  // Try to get the keys from local storage
  const savedOpenAIKey = localStorage.getItem('openaikey');
  const savedChatPDFKey = localStorage.getItem('chatpdfkey');

  // If keys are found, put them back into the input boxes
  // (This makes it look like the website remembered them)
  if (savedOpenAIKey) {
    openaikeyInput.value = savedOpenAIKey;
  }
```

```
  if (savedChatPDFKey) {
    chatpdfkeyInput.value = savedChatPDFKey;
  }
}

// Run the check when the page/modal is ready
checkLocalStorageKeys();
```

This JavaScript uses localStorage.setItem() to save information and localStorage.getItem() to retrieve it. When you save, it puts your key string into the browser's notebook. When the page loads, it checks the notebook and fills the boxes if keys are found.

Using the Keys on Other Pages

Once your keys are saved, any other page in the vivacityGPTComplete project can access them using *localStorage.getItem('openaikey')* or *localStorage.getItem('chatpdfkey')*.

Here's a tiny snippet from the *dalle.html* file (the *Image Creator* tool), showing where it gets the API key right before trying to generate an image:

```
async function generateImage() {
  // Get the OpenAI key from local storage
  const apikey = localStorage.getItem("openaikey");

  // Check if a key was found
  if (apikey === "" || apikey === null) {
    alert("No OpenAI API Key found. Please set it on the home page.");
    return; // Stop the function if no key is found
  }

  console.log("Using API Key:", apikey); // Log the key (for debugging)

  // ... rest of the code to prepare and send the request ...
  try {
    const response = await fetch(
      "https://api.openai.com/v1/images/generations",
      {
        method: "POST",
        headers: {
          "Content-Type": "application/json",
          // Include the API key in the Authorization header
          Authorization: "Bearer " + apikey
        },
        // ... rest of the request body ...
      }
    );
    // ... handle the response ...
```

```
  } catch (error) {
    // ... handle errors ...
  }
}
```

This code shows how the generateImage function simply retrieves the openaikey from local storage and uses it in the Authorization part of the request it sends to OpenAI.

You'll find similar lines of code using *localStorage.getItem* in most of the HTML files in the project (*txtchatbot.html, pdfchatbot.html, tts.html, whisper.html*, etc.) because they all need to talk to external AI services that require these keys.

What's Next?

You now understand that your personal API keys are necessary to connect to AI services and that VivacityGPT Complete uses your browser's storage as a simple, convenient way to keep track of them for you. In the next chapter, we'll explore another crucial interaction with the website: how you can upload files (like documents or images) for the AI tools to process. Head over to **File Upload Mechanism**!

Chapter 3: File Upload Mechanism

In Chapter 2: *API Key Management*, we learned how your API keys act like a membership card, allowing the website to access powerful AI services. Now, we'll explore another crucial part: getting the information you want the AI to work with from your computer to the website, especially when that information is in a file!

Imagine you have a great photo you want an AI '*Art Critic*' to analyze, or an audio recording you need transcribed by an AI '*Audio Transcriber*'. The AI lives on a server somewhere else, and it can't directly see the files stored on your personal computer. You need a way to safely and efficiently send those files over to the server where the AI tools are running or can access them.

This is where the **File Upload Mechanism** comes in! It's the process that lets you select a file from your computer using a "Choose File" button and then sends that file "behind the scenes" to the website's server. Once the file is on the server, it becomes accessible via a unique web address (a URL), which the AI can then use to process the file's content. Think of it as a secure digital mailbox on the server where you drop off files for the AI 'expert' to pick up.

What Problem Does File Upload Solve?

The core problem is transferring data (your files) from your local computer to the remote server where the AI tools are located. The file upload mechanism provides a standard and reliable way to do this over the internet.

Our project uses a common web approach involving a combination of:

1. An HTML element to select the file.
2. A form to package the file data.
3. JavaScript to handle the user interaction and send the file without refreshing the page.
4. A server-side script (written in PHP in this project) to receive the file and save it.

A Central Use Case: Analyzing an Image

Let's look at a tool like the *AI Art Critic* or the *AI Image Rebuilder*. The first step for these tools is always to get the image file from you. This happens using the file upload feature.

You'll see a part of the page that looks like this:

```
<!-- A button to choose a file from your computer -->
<input type="file" accept=".jpg,.png" name="file" id="vfile-input">

<!-- A button to trigger the upload -->
<button id="vuploadBtn" type="submit"> Upload the image</button>

<!-- An area to show feedback -->
<div id="vfeedback"></div>

<!-- The form that wraps these elements -->
<form enctype="multipart/form-data" method="POST">
 <!-- ... input and button elements go here ... -->
</form>
```

This is the basic HTML structure you might see on a page like the Art Critic or Image Rebuilder. The <input type="file"> creates the "Choose File" button, and the <button type="submit"> is for uploading. They are wrapped in a <form> tag.

When you click "Choose File" and select an image, and then click "Upload the image", here's a simplified look at what happens behind the scenes to get that file from your computer to the server.

How File Upload Works (Step-by-Step)

1. **You Choose a File:** You click the <input type="file"> element, and your operating system's file browser pops up. You select a file (e.g., my_painting.jpg).
2. **The File is Ready:** The browser now holds onto the selected file, ready to send it.
3. **You Initiate Upload:** You click the "Upload the image" button.
4. **JavaScript Intervenes:** Instead of letting the <form> do a traditional page refresh submit, JavaScript code (often using libraries like *jQuery*, as seen in the project) intercepts the click event.
5. **Packaging the File:** The JavaScript code gathers the file data from the <input type="file"> element and packages it nicely, usually using a standard web technique called FormData.
6. **Sending to the Server:** The JavaScript uses an asynchronous request (like $.ajax or fetch) to send this packaged file data to a specific address on the server, pointing to the PHP script responsible for handling uploads (*vupload.php or tupload.php* in this project).
7. **Server Receives and Saves:** The PHP script *(vupload.php* in our image example) running on the server receives the file data. It then saves the file into a designated folder on the server (like ./upload/).
8. **Server Responds:** After successfully saving the file, the PHP script sends back a response to the browser. This response includes information like a success message and, most importantly, the web address (URL) where the file was saved on the server (e.g., upload/my_painting.jpg).
9. **Browser Gets the URL:** The JavaScript code in the browser receives this response.
10. **Using the URL:** The JavaScript code can then use the received URL (e.g., pass it to a function that talks to the AI API) to tell the AI where to find the file for processing.

Looking at the Code (Simple View)

Let's simplify the code snippets from files like:

art_critic.html or *txtchatbot.html* and *vupload.php* or *tupload.php* to see these steps.

First, the HTML form elements (from *art_critic.html*):

```
<form enctype="multipart/form-data" method="POST">
 <input type="file" accept=".jpg,.png" name="file" id="vfile-input">
 <div id="vfeedback"></div>
 <button id="vuploadBtn" type="submit"> Upload the image</button>
 <!-- The button to analyze will be enabled/updated AFTER upload -->
 <button type="button" onclick="getChatGPTResponseIMG();"> Analyze the image</button>
</form>

```

This sets up the file input, a place for messages (vfeedback), and the upload button within a <form>. The enctype and method are important for file uploads.

Now, the JavaScript (using jQuery as seen in *art_critic.html*) that handles the upload button click:

```
$(document).ready(function () {
  $('#vuploadBtn').click(function (e) {
    e.preventDefault(); // Stop the form from submitting normally (which would refresh the page)

    $.ajax({
      url: 'vupload.php', // The server script that handles the upload
      type: 'POST',
      data: new FormData($('form')[0]), // Package the form data, including the file
      processData: false, // Don't process data string
      contentType: false, // Don't set content type header (let FormData do it)
      success: function (response) {
        const data = JSON.parse(response); // The server sends back JSON
        if (data.status === "success") {
          $('#vfeedback').html("Your file was uploaded.");

          // Get the file object from the input to display a local preview
          var fileInput = document.getElementById('vfile-input');
          var file = fileInput.files[0];
          if (file) {
            // Create a temporary URL for the image in the browser
            var originalimageUrl = URL.createObjectURL(file);
            // Display the image preview
            $('#uploaded-image').attr('src', originalimageUrl).show();
          }

          // IMPORTANT: Update the 'Analyze' button to include the URL from the server response
          $('#voperations button[type="button"]').attr('onclick', `submitImage("${data.url}");`);

        } else {
          $('#vfeedback').html(data.message); // Show server-side error messages
        }
      },
      error: function () {
        $('#vfeedback').html('File upload failed.');
      }
    });
  });
});

```

This JavaScript code listens for the "Upload" button click. It prevents the default form submission, creates a FormData object containing the selected file, and sends it *to vupload.php using a jQuery AJAX request. On success, it parses the JSON response, updates the feedback area, shows a local preview, and sets the onclick handler for the "Analyze" button to use the URL provided by the server (data.url).*

Finally, the simplified PHP server-side script (*vupload.php*):

```
<?php
function uploadFile() {
 $targetDir = "./upload/"; // Folder on the server to save files
 // Get file details
 $fileName = basename($_FILES["file"]["name"]);
 $targetFile = $targetDir . $fileName;
 $uploadOk = 1;
 $fileType = strtolower(pathinfo($targetFile,PATHINFO_EXTENSION));

 // Check if file type is allowed (for images)
 if($fileType != "jpg" && $fileType != "png") {
   echo json_encode(["status" => "error", "message" => "Only JPG and PNG files are allowed."]);
   $uploadOk = 0;
 }

 // If checks pass, try to move the file from the temporary location to the target folder
 if ($uploadOk == 0) {
   // Error already echoed
 } else {
   if (move_uploaded_file($_FILES["file"]["tmp_name"], $targetFile)) {
     // File uploaded successfully, send success status and the file's URL back
     echo json_encode([
       "status" => "success",
       "message" => "Your file was successfully uploaded.",
       "url" => $targetFile // The URL to access the file on the server
     ]);
   } else {
     // Error moving the file
     echo json_encode(["status" => "error", "message" => "An error occurred during the upload."]);
   }
 }
}

uploadFile(); // Run the function when the script is called
?>
```

This PHP script receives the file sent by the browser in the $_FILES variable. It checks the file type and then uses move_uploaded_file to save the file from a temporary server location to the permanent ./upload/ folder. It then outputs a JSON string indicating success or failure and provides the relative path (./upload/filename.jpg), which becomes part of the full URL (https://yourwebsite.com/vivacityGPTComplete/upload/filename.jpg) that the JavaScript will use.

Using the Uploaded File URL

Once the JavaScript in the browser receives the *data.url* from the PHP script, it now has the address of the file on the server. The tool's specific logic (covered in Chapter 4: *Tool-Specific AI Logic*) can then use this URL when making a request to the AI API (covered in Chapter 5: *OpenAI API Communication*). For example, the submitImage function called by the "Analyze the image" button after a successful upload receives this URL and includes it in the API request to OpenAI's Vision model, telling the AI where to fetch the image:

```
async function submitImage(imgUrl) {
  // imgUrl is the URL received from the server, e.g., "upload/my_painting.jpg"
  const fullImageUrl = "https://www.vivacitydesign.net/vdgptscomplete/" + imgUrl; // Construct the full URL

  // Retrieve the API key (learned in Chapter 2)
  const apikey = localStorage.getItem("openaikey");

  // ... rest of the function prepares the API request ...

  const response = await fetch("https://api.openai.com/v1/chat/completions", {
    method: "POST",
    headers: {
      "Content-Type": "application/json",
      "Authorization": "Bearer " + apikey // Use the API key
    },
    body: JSON.stringify({
      model: "gpt-4.1-nano", // Or other vision-capable model
      messages: [
        // ... chat history (Chapter 6) ...
        {
          role: "user",
          content: [
            { type: "text", text: "Analyze this image..." },
            {
              type: "image_url",
              image_url: {
                url: fullImageUrl // Use the URL of the uploaded image!
              }
            }
          ]
        }
      ]
    })
  });

  // ... handle the AI response ...
}
```

> *This simplified code shows how the imgUrl received from the file upload process is incorporated into the messages part of the OpenAI API request, specifically within the image_url object, allowing the AI to access and process the image.*

Other tools like the PDF Chatbot or the audio transcription tool (*whisper.html*) use a similar upload mechanism, often calling slightly different server-side scripts (*tupload.php* for text/audio in *Total Analyzer*, *vupload.php* for images) and passing the resulting URL to their respective AI processing functions. The core principle of using HTML form elements, JavaScript for the client-side process, and a server script to handle the file saving and URL generation remains consistent.

What's Next?

You've now seen how files get from your computer to the server so the AI can see them. You also know how API keys are managed. But how does the project decide *what* to do with that file or text input? Each tool has its own specific way of interacting with the AI. In the next chapter, we'll explore **Tool-Specific AI Logic** – the unique instructions and steps each tool uses to process your requests and interact with the AI models.

Chapter 4: Tool-Specific AI Logic

In our previous chapters, we've explored the front-end dashboard (Chapter 1: *User Interface Handling*), how we securely manage your access "keys" (Chapter 2: *API Key Management*), and how we get your files from your computer to the website's server (Chapter 3: *File Upload Mechanism*). Now, we're getting to the core of what makes each tool in VivacityGPT Complete unique. While all these tools use powerful AI models from services like OpenAI, the general-purpose AI model needs to know exactly *how* to behave for each specific task. The same AI that can write a blog post might also need to analyze an image or act as a language teacher.

This is where **Tool-Specific AI Logic** comes in. It's the set of instructions and processing steps that guide the general AI model to perform a specialized task, giving each tool its own 'personality' and purpose.

What Problem Does Tool-Specific AI Logic Solve?

The main problem is directing and instructing the AI. Think of the core AI model as a brilliant, multi-talented person. They *can* do many things, but when you approach them, you need to give clear instructions: "*Please act as an art critic and analyze this painting,*" or "*You are a web developer bot; I need you to write HTML code for me.*"

Tool-Specific AI Logic provides these clear instructions. It tells the AI:

- **What role to play:** "You are an Art Critic," "You are a Web Developer Bot."
- **What kind of task to perform:** "Analyze this image," "Generate HTML code."
- **What format the output should be:** "Provide analysis focusing on style and color," "Only provide code, no explanations."
- **How to handle the specific inputs for this tool:** (e.g., process an image URL).
- **How to prepare or interpret the output:** (e.g., display the analysis text, save generated code).

A Central Use Case: The AI Art Critic

Let's take the *AI Art Critic* tool as an example. Its job is to analyze an image you upload and provide an artistic critique. The user provides an image, but the AI needs to process this image and output *text*.

How does the project guide the AI to do this specific job?

1. **User uploads an image:** This happens via the File Upload Mechanism, resulting in a URL for the image on the server (as we saw in Chapter 3).
2. **User clicks "Analyze":** This triggers JavaScript code.
3. **The Magic Instruction (System Prompt):** The JavaScript doesn't just send the image URL to the AI. It first prepares a set of instructions for the AI. This is often called the **system prompt**. For the Art Critic, the system prompt tells the AI to *be* an Art Critic and *what kind* of analysis to perform.
4. **User's Request:** The user didn't type a text question *with* the image upload (though they could add instructions). The system prompt *is* the core request for this tool.
5. **Sending to AI:** The system prompt *and* the image information (via its URL) are packaged together and sent to the AI API.
6. **AI Processes:** The AI receives the instructions ("*Be an Art Critic...*") and the image URL. It uses its vision capabilities to "see" the image and its language capabilities to generate text based on the instructions.
7. **AI Responds:** The AI sends back the text analysis.
8. **Displaying Output:** The UI receives the text and displays it in the designated output area.

Looking at the Code (The System Prompt)

The most common way Tool-Specific AI Logic is implemented in this project is through the **system prompt.** This is a special message sent to the AI at the beginning of the conversation (or request) that defines its role and guidelines.

In the Art Critic tool (*art_critic.html*), you can see this defined in the JavaScript:

```
// Creation of the memory array with system prompt
let chatMemory = [];
chatMemory = createMemory([{
  role: "system", // This tells the AI this is an instruction, not a user message
  content: "You are an AI Art Critic. You will analyze the uploaded image and you will provide a critic artistic analysis focalizing on the following points: style, choice of coors, overall meaning, captured feelings. If applicable, you will provide example of famous artists with a similar style, and you will suggest how to improve the art work."
}]);

// Definition of structure for context memory
function createMemory(messages) {
  const memory = [];
  for (const msg of messages) {
    memory.push({
      role: msg.role,
      content: msg.content
    });
  }
  return memory;
}
// ... rest of the code ...
```

This JavaScript code creates an array called chatMemory. It immediately adds a message with the role set to "system" and content containing the detailed instructions for the Art Critic persona. This chatMemory array will be sent with every request to the AI for this tool.

When the user uploads an image and clicks "Analyze", the *submitImage* function is called. It gets the image URL (from the upload process we saw in Chapter 3: *File Upload Mechanism*) and the *chatMemory* (which contains our system prompt). It then prepares the request body for the OpenAI API call:

```
async function getChatGPTResponseIMG(fileUrl, chatMemory) {
 // ... (get API key, show loading indicator) ...

 try {
  response = await fetch("https://api.openai.com/v1/chat/completions", {
    method: "POST",
    headers: { /* ... */ },
    body: JSON.stringify({
      model: "gpt-4.1-nano", // or other vision-capable model
       messages: [
        ...chatMemory.map((item) => ({ ...item })), // <-- Includes the system prompt
        {
          role: "user",
          content: [ // The user's "message" includes both text (optional) and the image
```

```
          { type: "text", text: "Analyze this image..." }, // Optional user text
          {
            type: "image_url",
            image_url: {
              url: fileUrl // <-- The URL from the file upload
            }
          }
        ]
      }
    ],
    max_tokens: 600
  })
 });
 // ... (handle response, display output) ...
} catch (error) {
 // ... (handle errors) ...
}
}
```

This simplified code shows how the chatMemory array (containing the system prompt) is spread into the messages array, effectively putting the instructions at the beginning of the conversation history sent to the AI, followed by the user's input (which for image analysis includes the image URL).

This system prompt is crucial. It's the core of the "Art Critic" logic. Without it, sending just the image URL might result in a generic description or no response at all, as the AI wouldn't know *what kind* of text to generate about the image.

Different Tools, Different Logic

To see how this applies to other tools, let's look at the system prompts from a couple more examples:

1. AI Website Builder (website_builder.html)

This tool takes a text description of a desired website landing page and generates the HTML code.

```
let chatMemory = [];
chatMemory = createMemory([
  {
    role: "system",
    content: "You are a web designer bot. Your output is made of code to satisfy the received request. You will not greet or thank the user for the request, your aim is to produce a website layout design to satisfy the request so you will produce a well structured code line by line without interruptions and without dividing it in sections. You will only use internal CSS, placeholder images, and all data available in the request to produce a complete website layout."
  }
```

```
]);
```

Here, the system prompt strictly defines the AI's role ("web designer bot") and output format ("Your only output is made of code...", "will not explain the code", "only internal CSS", "placeholder images"). This forces the general AI model to produce structured HTML code, specific to this tool's purpose.

2. AI Virtual Sommelier (virtual_sommelier.html)

This tool takes a text description of a dish and suggests wine pairings.

```
let chatMemory = createMemory([
  { role:
   'system',
   content: `Sei un esperto sommelier incaricato di consigliare abbinamenti tra vini e cibo, con particolare attenzione alla cucina italiana.
Segui queste linee guida per offrire consigli accurati e coerenti:

1. Principi di base dell’abbinamento: ...
2. Caratteristiche dei vini italiani più indicati: ...
3. Esempi pratici di abbinamenti con pietanze tipiche italiane: ...
4. Consigli aggiuntivi: ...
5. Se l'utente ti chiede indicazioni su un piatto o su un vino specifici, fornisci una sincera valutazione dei migliori abbinamenti, e proponine almeno 2.`
  }
]);
```

(Note: This prompt is in Italian as per the source file. The translation is: "You are an expert sommelier tasked with recommending wine and food pairings... Follow these guidelines... 1. Basic pairing principles... 2. Characteristics of the most suitable Italian wines... 3. Practical examples... 4. Additional tips... 5. If the user asks for specific dish or wine guidance, provide an honest evaluation of the best pairings, and suggest at least 2.")

This system prompt establishes the AI's persona ("expert sommelier") and provides specific knowledge and rules (pairing principles, Italian wines, examples) relevant to its task.

Specialized Input/Output Processing

Beyond the system prompt, tool-specific logic can also involve JavaScript code that processes inputs or outputs in a way unique to that tool:

- **Processing Input:** In the *Interface Analyzer* or *Image Rebuilder*, the AI is first asked to *describe* the image. The output of this first AI call (the description text) is then sometimes *used as the input* for a *second* AI call (e.g., using DALL-E to generate a *new* image based on the description, as seen in *img_rebuilder.html*). This multi-step process is part of the tool's logic and it's called 'a *request pipeline*'.

- **Processing Output:** The Website Builder (*website_builder.html*) tool displays the raw HTML code generated by the AI in one panel and renders that code in an <iframe> in another panel. The Dev Bot (*devbot.html*) does the same. This is specific output handling. The Transcript Creator (*transcript_creator.html*) takes chunks of text and sends them one by one to a Text-to-Speech AI, then pieces the resulting audio files together. This is complex output processing unique to that tool.

These examples show that Tool-Specific AI Logic isn't just *one* thing; it's the combination of the carefully crafted system prompt and any unique input/output handling code that makes a general AI model perform a specialized function within the project.

What's Next?

You now understand that each tool has specific instructions and processing steps that tailor the general AI's capabilities to a particular task, primarily through the "system prompt." But how do these instructions and your input actually *get* to the AI service (like OpenAI) and how does the response come back? In the next chapter, we'll look at the technical details of this crucial step: **OpenAI API Communication**.

Chapter 5: OpenAI API Communication

In the previous chapters, we've learned about:

- How the User Interface (Chapter 1: *User Interface Handling*) lets you type, click, and see results.
- How your special keys are remembered using Local Storage (Chapter 2: *API Key Management*).
- How you can send files from your computer to the website's server (Chapter 3: *File Upload Mechanism*).
- How each tool has its own specific instructions and steps to guide the AI (Chapter 4: *Tool-Specific AI Logic*).

Now, we get to the really exciting part: making the connection! How does the website actually *talk* to the powerful AI models hosted by services like OpenAI and ChatPDF?

Think of this step as the **messenger service**. It takes everything prepared in the earlier steps – your question, the tool's instructions, maybe a file URL, and your secret API key – packages it up, sends it over the internet to the AI, waits for the AI's reply, and brings the answer back to the website.

What Problem Does API Communication Solve?

The main challenge is that the AI models aren't running directly on your computer or even on the same web server as the vivacityGPTComplete files. They are located on servers managed by companies like OpenAI and ChatPDF, accessible only through a controlled gateway called an **API** (*Application Programming Interface*). This chapter explains how our project uses web standards to send requests to these APIs and handle the responses.

A Central Use Case: Sending a Text Message

Let's go back to our simplest example: typing a message in a chat box and getting a text response, like in the *Text Chatbot* or *AI Virtual Sommelier* tools.

You type your message, click "Send", and the AI's answer appears. Behind the scenes, the "Send" button triggers a process that looks something like this:

1. The JavaScript code in your browser gets the text you typed from the input box (thanks to Chapter 1: User Interface Handling).
2. It retrieves your API key from your browser's *Local Storage* (thanks to Chapter 2: API Key Management).
3. It prepares a set of messages to send to the AI. This includes your message and possibly important initial instructions for the AI (the system prompt from Chapter 4: Tool-Specific AI Logic).
4. It creates a **request** to send to the AI service. This request is like a package containing:
    - The address of the specific AI capability (e.g., the chat model).
    - Your API key (like postage confirming you're allowed to send the package).
    - The content of your message and the instructions for the AI.
5. The request is sent over the internet.
6. The AI service receives the request, processes the information using its powerful models.
7. The AI service sends back a **response** package containing the AI's generated text.
8. The JavaScript code in your browser receives the response, extracts the AI's text, and displays it on the page (back to Chapter 1: User Interface Handling).

Looking at the Code (Using fetch)

The most common way modern JavaScript sends these requests over the internet is using the fetch API built into web browsers.

Let's look at a simplified example from the Text Chatbot (*txtchatbot.html*) or AI Virtual Sommelier (*virtual_sommelier.html*) that makes the call to **OpenAI's Chat Completions** API. The relevant function is *getChatGPTResponse*:

```
async function getChatGPTResponse(userInput, chatMemory = []) {
  // Get the API key from Local Storage (from Chapter 2)
  const apikey = localStorage.getItem("openaikey");

  // Check if the key exists
  if (apikey === "" || apikey === null) {
    alert("No OpenAI API Key found. Please set it on the home page.");
    return; // Stop if no key
  }

  // ... (show a loading indicator on the UI - Chapter 1 detail) ...

  try {
    // This is the core FETCH call to the OpenAI API
    const response = await fetch("https://api.openai.com/v1/chat/completions", {
      method: "POST", // We are sending data (the prompt and messages)
      headers: {
        "Content-Type": "application/json", // Tells the API we're sending JSON data
        "Authorization": "Bearer " + apikey // Proves we are allowed to make this request (using the API key from Chapter 2)
      },
      body: JSON.stringify({ // This is the "package" content
        model: "gpt-4.1-nano", // The specific AI model we want to use
        messages: [ // The list of messages for the AI (the conversation history)
          ...chatMemory, // Includes the system prompt (Chapter 4) and past turns (Chapter 6)
          { role: "user", content: userInput } // Adds the user's current message
        ]
      })
    });

    // Check if the request was successful (e.g., status 200 OK)
    if (!response.ok) {
      // If not, throw an error with details
      throw new Error(`API request failed with status ${response.status}`);
    }

    // Get the JSON response from the AI
    const data = await response.json();

    // Check if the response contains the expected data (AI's message)
```

```
    if (!data.choices || !data.choices.length || !data.choices[0].message || !data.choices[0].message.content) {
        throw new Error("Invalid API response structure.");
    }

    // Extract the AI's text response
    const chatGPTResponse = data.choices[0].message.content.trim();

    // ... (hide loading indicator, display chatGPTResponse on UI - Chapter 1 detail) ...

    // ... (handle adding the response to chat history - Chapter 6 detail) ...

    return chatMemory; // Return the updated memory

  } catch (error) {
    console.error(error);
    alert("An error occurred during the API request. Please check your API key and network connection.");
  }
}
```

This simplified JavaScript function shows the steps to build and send the request to OpenAI using fetch. It uses the API key, specifies the content type, points to the correct API endpoint URL, and includes the conversation messages in the body, formatted as JSON.

Let's break down the key parts of the fetch call:

- fetch("https://api.openai.com/v1/chat/completions", { ... }): This tells the browser to send a request to this specific web address (the **API Endpoint** for chat completions).
- method: "POST": We are sending data to the server, not just asking for a page (GET).
- headers: { ... }: These provide important metadata about the request.
  - "Content-Type": "application/json": This is standard and tells the server that the data in the body is in JSON format.
  - "Authorization": "Bearer " + apikey: This is how you authenticate your request. "Bearer " is a standard part, followed by a space and then your actual API key (retrieved from Chapter 2: API Key Management).
- body: JSON.stringify({...}): This contains the actual data for the AI.
  - model: "gpt-4.1-nano": Specifies which version of the AI model to use. Different models have different capabilities and costs.
  - messages: [...]: This array holds the conversation turns. It includes the system message (from Chapter 4: Tool-Specific AI Logic) and the user's current input. For conversations, it would also include previous turns (details in Chapter 6: Chat Context Memory). *The JSON.stringify()* part converts the JavaScript object into a string format that can be sent over the internet.

After the request is sent, the await *response.json()* line waits for the AI's reply and converts the received JSON data into a JavaScript object that we can then easily work with to get the actual text (data.choices[0].message.content).

Communication with Other APIs

The same core fetch mechanism is used for other AI tasks and services, but the specific **endpoint URL** and the structure of the **request body** might change.

- **Image Generation (DALL-E):** Found in *dalle.html* or *blogpostcreator.html*, this calls a different endpoint: https://api.openai.com/v1/images/generations. The body includes a prompt property describing the desired image, a model (like "dall-e-3"), and desired size.
- **Text-to-Speech (TTS):** Found in *tts.html* or *transcript_creator.html*, this calls https://api.openai.com/v1/audio/speech. The body includes the input text, a model (like "gpt-4o-mini-tts"), and a selected voice. The response is audio data (a Blob) instead of JSON text.
- **Speech-to-Text (Whisper):** Found in *whisper.html* or used internally in *total_chatbot.html* or *voicechatter_advanced.html*, this calls https://api.openai.com/v1/audio/transcriptions. This uses FormData (like the file upload in Chapter 3) to send the audio file in the body, along with the model ("whisper-1"). The response is JSON containing the transcribed or translated text.
- **PDF Chatbot (ChatPDF):** Found in *pdfchatbot.html*, this uses the ChatPDF API (https://api.chatpdf.com/v1/). It requires a ChatPDF API key (managed similarly via *Local Storage* as in Chapter 2: API Key Management). The endpoint for sending messages is https://api.chatpdf.com/v1/chats/message, and the header uses "x-api-key": apikey instead of Authorization: Bearer. The body includes the user's message and the sourceId of the uploaded PDF file.

Here's a tiny snippet showing the ChatPDF API call from *pdfchatbot.html*:

```
function sendQuestion() {
 const apiUrl = "https://api.chatpdf.com/v1";
 // Get ChatPDF key from Local Storage
 var apikey = localStorage.getItem("chatpdfkey");

 // Check if key exists...

 // Assuming sourceId (from PDF upload) is available
 if (!sourceId) {
```

```
  console.error("Please upload a PDF first.");
  return;
}

const data = {
  sourceId: sourceId, // The ID of the uploaded PDF (from Chapter 3 logic)
  messages: [ // The list of messages
    { role: "user", content: prompt } // The user's message
  ]
};

fetch(`${apiUrl}/chats/message`, {
  method: "POST",
  headers: {
    "x-api-key": apikey, // Different header name for ChatPDF API key
    "Content-Type": "application/json"
  },
  body: JSON.stringify(data)
})
.then((response) => response.json())
.then((data) => {
  // ... process and display the response from ChatPDF ...
})
.catch((error) => console.error("Error sending question:", error));
}
```

This snippet from pdfchatbot.html shows the similar fetch structure but with the ChatPDF API endpoint, a different header name for the key (x-api-key), and the sourceId included in the request body to tell the AI which PDF to chat with.

In essence, while the specific URL, headers, and body content vary depending on the task and the AI service being used, the core mechanism of sending an HTTP request (usually POST with JSON body) via JavaScript's fetch function, including the necessary API key for authentication, is the universal "messenger service" that allows vivacityGPTComplete to talk to powerful AI models.

What's Next?

You now understand how the website packages your request (including your input, tool instructions, and API key) and sends it off to the AI service using web requests, and how it receives the AI's answer back. For chat-based tools, like the Text Chatbot or Virtual Sommelier, having the AI respond to just one message isn't very useful. You want the AI to remember what you said before so it can continue the conversation. In the next chapter, we'll explore how the project keeps track of the back-and-forth in a conversation: **Chat Context Memory**.

Chapter 6: Chat Context Memory

In our journey so far, we've seen how you interact with the tools through the User Interface Handling (Chapter 1), how the project remembers your special access keys using API Key Management (Chapter 2), how you can send files for the AI to look at using the File Upload Mechanism (Chapter 3), how each tool gets its specific instructions thanks to Tool-Specific AI Logic (Chapter 4), and how the website actually sends your requests and receives responses from the powerful AI services via OpenAI API Communication (Chapter 5).

Now, imagine you're having a conversation with a friend. If they only responded to your very last sentence and forgot everything else you've talked about, the conversation would be pretty strange, right? It would lack context and coherence.

AI models, by default, are a bit like that friend with a very short memory. Each time you send a message to the API, the AI only sees *that specific message* in isolation. If you ask, "What's the capital of France?", it might answer "Paris". But if you then ask, "What language do they speak there?", without remembering the previous question, the AI wouldn't know "there" refers to France.

This is where **Chat Context Memory** comes in! It's the system that gives the AI a "short-term memory" of your conversation within a single session.

What Problem Does Chat Context Memory Solve?

The main problem is maintaining continuity in a conversation. For a chatbot to provide relevant follow-up answers, refer back to previous points, or understand the flow of your discussion, it needs to know what was said *before* the current message.

Chat Context Memory solves this by keeping track of the messages exchanged during your current session.

How Does It Work?

Instead of sending *only* your new message to the AI API, we send a list that contains:

1. Sometimes, initial instructions for the AI (the **system prompt**, as discussed in Chapter 4: *Tool-Specific AI Logic*).
2. Your previous messages in this session.
3. The AI's previous responses in this session.
4. Your *current* new message.

This list, often called the **message history** or **context window**, provides the AI with the necessary background to understand the current message within the flow of the conversation.

A Central Use Case: A Simple Text Chat

Let's revisit the simple chat example from the *Text Chatbot* or *AI Virtual Sommelier*.

Imagine this conversation:

- **You:** "What is the largest country in the world?"
- **AI:** "The largest country in the world by land area is Russia."
- **You:** "What is its capital city?"

If we only sent "What is its capital city?" to the AI, it wouldn't know "its" refers to Russia. With Chat Context Memory, we send the full list:

```
[
 {"role": "system", "content": "You are a helpful assistant."}, // Optional system prompt
 {"role": "user", "content": "What is the largest country in the world?"},
 {"role": "assistant", "content": "The largest country in the world by land area is Russia."},
 {"role": "user", "content": "What is its capital city?"} // The new message
]
```

Seeing this list, the AI understands the conversation history and can correctly answer, "The capital city of Russia is Moscow."

Looking at the Code (The chatMemory Array)

In the VivacityGPT Complete project, this "memory" is typically managed by a simple JavaScript array (a list) in the browser's memory, often named *chatMemory*. Let's look at a simplified version of the code from a file like *txtchatbot.html* or *virtual_sommelier.html*.

First, the chatMemory array is initialized. It might start empty, or it might start with a system message to give the AI its persona (as seen in Chapter 4).

```
// This array will hold the conversation history
let chatMemory = [];

// Often, the first message is a system prompt (from Chapter 4)
chatMemory = createMemory([
 {
   role: "system",
   content: "You are a helpful assistant that answers questions concisely."
```

```
 }
]);

// Function to create the initial memory structure (might just be the system prompt)
function createMemory(messages) {
 const memory = [];
 for (const msg of messages) {
   memory.push({
     role: msg.role,
     content: msg.content
   });
 }
 return memory;
}

console.log("Initial chatMemory:", chatMemory);
// Output might look like: Initial chatMemory: [ { role: "system", content: "You are a helpful assistant that answers questions concisely." } ]
```

This code snippet shows how a variable chatMemory is declared as an empty list and then potentially populated with an initial "system" message, defining the AI's role.

Now, let's look at the sendMessage function (the one that runs when you click the "Send" button, as discussed in Chapter 1). This function is responsible for getting your input, *adding it to chatMemory*, sending the request, getting the response, and *adding the response to chatMemory*.

```
// Function triggered when user clicks "Send"
async function sendMessage() {
 const inputElement = document.getElementById("prompt"); // Get the input box (Chapter 1)
 const userInput = inputElement.value.trim(); // Get the text

 if (userInput !== "") {
   // 1. Display the user's message on the UI (Chapter 1 detail)
   showMessage("Guest", userInput, "");

   // 2. Call the function to talk to the AI, passing the user's input AND the current memory
   // The function getChatGPTResponse will handle adding the messages to memory
   chatMemory = await getChatGPTResponse(userInput, chatMemory);

   // 3. Clear the input box (Chapter 1 detail)
   inputElement.value = "";
 }
}
```

This sendMessage function gets the user's text and then passes it, along with the current chatMemory list, to the getChatGPTResponse function which handles the API communication.

The core logic of adding messages to the *chatMemory* array happens within the function that communicates with the API (*getChatGPTResponse*), often just *before* sending the request and again *after* receiving the response.

Here's a simplified look at that part within *getChatGPTResponse* (referencing Chapter 5):

```
async function getChatGPTResponse(userInput, currentChatMemory = []) {
 // ... (Get API key - Chapter 2) ...
 // ... (Show loading indicator - Chapter 1) ...

 // 1. Create a NEW messages list for THIS request
 // Start with the previous memory (system prompt + past turns)
 // Then add the user's current message
 const messagesToSend = [
  ...currentChatMemory, // Include all messages from previous turns
  { role: "user", content: userInput } // Add the current user message
 ];

 try {
  // 2. Send the request to the API (using fetch - Chapter 5)
  const response = await fetch("https://api.openai.com/v1/chat/completions", {
   method: "POST",
   headers: { /* ... API key and content type ... */ },
   body: JSON.stringify({
    model: "gpt-4.1-nano",
    messages: messagesToSend // <-- Send the list including history!
   })
  });

  // ... (Check response status - Chapter 5) ...

  const data = await response.json();

  // ... (Extract AI's text response - Chapter 5) ...
  const chatGPTResponse = data.choices[0].message.content.trim();

  // 3. Update the chatMemory variable by adding the new turns
  // Add the user's message first (already included in messagesToSend, but adding to *our* memory variable)
  currentChatMemory.push({ role: "user", content: userInput });
  // Add the AI's message
  currentChatMemory.push({ role: "assistant", content: chatGPTResponse });

  // ... (Hide loading indicator, display AI response on UI - Chapter 1) ...

  // 4. Return the updated chatMemory for the next turn
  return currentChatMemory;

 } catch (error) {
```

```
    // ... (Handle errors - Chapter 5) ...
    // Return the memory as is, or handle error appropriately
    return currentChatMemory;
  }
}
```

This simplified getChatGPTResponse function demonstrates the core memory logic: it takes the current chatMemory and the new user input, builds a messagesToSend array including everything, sends that full list to the API, and then, upon receiving a response, adds both the user's message and the AI's response to the currentChatMemory list before returning it, ready for the next interaction.

The key is that the *chatMemory* array keeps growing with each turn of the conversation. When you send a new message, the entire (or a truncated version, for very long chats) list is sent again to the AI, giving it the context.

Memory Considerations

- **Length Limits:** AI models have limits on how much text they can process at once (called the "context window" or "token limit"). Sending a very long *chatMemory* list can hit these limits or become expensive (APIs charge based on token usage, and both prompt and completion tokens count). For very long conversations, real-world applications often need strategies to summarize or truncate the *chatMemory* list. This project simplifies this by keeping the full history for typical session lengths.
- **Temporary:** The *chatMemory* variable exists only while the web page is open in your browser tab. If you close the tab or refresh the page, the *chatMemory* is reset, and the AI loses its "short-term memory."

What's Next?

You now understand that **Chat Context Memory**, managed by a simple JavaScript array (*chatMemory*) in the browser, is essential for enabling coherent conversations with the AI. It keeps track of the back-and-forth by adding each message to a list that's sent with subsequent requests.

While this provides memory *for the current session*, what if you close the browser and want to continue the same conversation later? The current chatMemory would be lost. In the next chapter, we'll explore how to save and load this conversation history permanently: **Chat History Persistence**.

Chapter 7: Chat History Persistence

In the previous chapter, Chapter 6: *Chat Context Memory*, we learned how the project keeps track of your conversation with the AI during your current browsing session using a JavaScript array called *chatMemory*. This lets the AI remember what you've said so it can provide relevant responses. However, you might have noticed that if you close the browser tab or refresh the page, that *chatMemory* is reset. The AI forgets everything you talked about! What if you're having a long, important conversation with a tool like the *Total Analyzer* or the *AI Voice Chatbot* and want to save it to continue later?

This is where **Chat History Persistence** comes in. It's the feature that allows you to save your chat conversations to a file on the server and load them back later. It's like saving a document you've been working on so you don't lose your progress and can open it again whenever you need it.

What Problem Does Chat History Persistence Solve?

The main problem is the temporary nature of the *chatMemory*. It lives only in your browser's current session. To make chat conversations truly useful over multiple visits, you need a way to store that history more permanently.

Our project solves this by combining:

1. **Client-side (in your browser):** JavaScript code to gather the conversation content currently shown on the page.
2. **Server-side (on the website's server):** PHP scripts that receive the chat content and save it as text files in a designated folder. It also includes scripts to list these files and load their content back when requested.
3. **User Association:** These saved files are typically linked to your user account (if you're logged in) so only you can see and load your saved chats.

A Central Use Case: Saving and Loading Your Conversation

Let's use the *Total Analyzer* tool as our example, as it includes this feature. Imagine you've had a detailed chat with the AI, analyzing a document and asking several follow-up questions.

The use case is:

1. You click a "Save Chat" button to save the current conversation.
2. Later, you visit the page again and see a list of your saved chats.
3. You click on a saved chat name, and the conversation history reappears in the chat window.

How does the project make this happen?

Saving Your Chat

When you click the "Save Chat" button:

1. **Gather Chat Content (Browser):** JavaScript looks at the HTML elements that display the conversation (*#chathistory*) and collects the text from all the user and AI messages shown there.
2. **Prepare Data (Browser):** This collected text, along with your *user ID* (fetched from the server when the page loads via PHP, as seen in the provided code snippets that check *$_SESSION*), is packaged into a format ready to be sent to the server (like a JSON object). A timestamp is also added to create a unique filename later.
3. **Send Data (Browser -> Server):** The JavaScript uses fetch (or AJAX, similar to Chapter 5: *OpenAI API Communication*) to send this packaged data to a specific PHP script on the server (*save_totalanalyzer_history.php*).
4. **Receive Data (Server):** The PHP script (*save_totalanalyzer_history.php*) receives the data sent from the browser.
5. **Save to File (Server Filesystem):** The PHP script takes the chat content, the user ID, and the timestamp to create a unique filename (e.g., *totalanalyzer_[user_id]_[timestamp].txt*). It then saves the chat content into a text file with that name inside a dedicated folder on the server (e.g., ./history/).

Looking at the Code (Saving)

Here's a simplified look at the pieces involved in saving a chat:

1. The "Save Chat" Button (in total_chatbot.html)

```
<button id="saveChatBtn" type="button" style="<?php echo $isLoggedIn ? 'display: block;' : 'display: none;'; ?>">Save Chat</button>
```

This HTML shows a button labeled "Save Chat". Notice the PHP part (<?php echo $isLoggedIn ? 'display: block;' : 'display: none;'; ?>) which makes the button visible only if a user is logged in, ensuring chats are saved per user.

2. The JavaScript to Trigger Save (in total_chatbot.html)

```
document.getElementById('saveChatBtn').addEventListener('click', saveChat);

async function saveChat() {
```

```
const chatContainer = document.getElementById("chathistory");
// Collect content from all message elements
const chatEntries = [...chatContainer.children]
 .map(entry => entry.innerHTML // Get the HTML content
  .replace(/Copy/g, "")      // Remove UI elements like 'Copy' button text
  .replace(/Send to TTS/g, "") // Remove 'Send to TTS' button text
  )
  .join('\n'); // Join messages with newlines

const userID = "<?php echo $_SESSION['user']['Id']; ?>"; // Get user ID from PHP variable
const timestamp = new Date().toISOString(); // Get current time

// Send data to the server script
const response = await fetch('save_totalanalyzer_history.php', {
  method: 'POST',
  headers: { 'Content-Type': 'application/json' },
  body: JSON.stringify({ chat: chatEntries, userID: userID, timestamp: timestamp })
});

if (response.ok) {
  alert('Chat saved successfully!');
  updateChatHistoryList(); // Refresh the list of saved chats
} else {
  alert('Failed to save chat.');
}
}
```

This JavaScript code runs when the "Save Chat" button is clicked. It grabs the HTML content from the chat display area, cleans up extra text from UI buttons, gets the user ID (which was made available in the HTML via PHP), adds a timestamp, and sends all this as a JSON object to save_totalanalyzer_history.php using the fetch API.

3. The PHP Script to Save the File (in save_totalanalyzer_history.php)

```
<?php
session_start(); // Start session to potentially use session data later (though user ID comes in POST here)

if ($_SERVER['REQUEST_METHOD'] === 'POST') {
 // Get data sent from the browser
 $data = json_decode(file_get_contents('php://input'), true);
 $chatContent = $data['chat'];
 $userID = $data['userID']; // Get user ID from the received data
 $timestamp = date('Y-m-d_H-i-s', strtotime($data['timestamp'])); // Format the timestamp

 // Create the filename using user ID and timestamp
 $filename = "./history/totalanalyzer_{$userID}_{$timestamp}.txt";

 // Save the chat content to the file
 file_put_contents($filename, $chatContent);
```

```

    // Send success response back to the browser
    echo json_encode(["status" => "success"]);
}
?>
```

This PHP script receives the POST request. It gets the chat content, user ID, and timestamp from the JSON data. It constructs a filename that includes the user ID and timestamp for uniqueness and organization, and then uses file_put_contents to write the chatContent into that file within the ./history folder on the server.

Loading Your Chat

Loading a chat involves two main steps: first, getting a list of saved chats, and second, loading the content of a specific chat when selected.

Getting the List of Saved Chats:

1. **Request List (Browser -> Server):** When the page loads (or after a chat is saved), JavaScript sends a request to another PHP script (*get_totalanalyzer_history.php*) asking for a list of the user's saved chat files.
2. **List Files (Server):** The PHP script reads the ./history folder, identifies which files belong to the currently logged-in user based on the filename structure (*totalanalyzer_[user_id]_...*), and creates a list of these filenames.
3. **Send List (Server -> Browser):** The PHP script sends this list of filenames back to the browser, usually as a JSON array.
4. **Display List (Browser):** JavaScript receives the list and displays it in a user interface element (like the <ul> list in the sidebar of *total_chatbot.html*).

Looking at the Code (Listing)

1. The PHP Script to Get the List (in get_totalanalyzer_history.php)

```
<?php
session_start(); // Start session to get user ID

// Ensure user ID exists in the session (basic check)
if (!isset($_SESSION['user']['Id'])) {
    echo json_encode([]); // Return empty list if user not logged in
    exit;
}

```

```
$userID = $_SESSION['user']['Id']; // Get user ID from session
$historyFolder = './history'; // Folder where chats are saved

// Get all files in the history folder
$files = array_diff(scandir($historyFolder), array('..', '.'));

// Filter files to include only those belonging to the current user
$filteredFiles = array_filter($files, function($file) use ($userID) {
  return strpos($file, "totalanalyzer_$userID") === 0; // Check if filename starts with "totalanalyzer_[user_id]"
});

// Return the list of filtered files as a JSON array
echo json_encode(array_values($filteredFiles));
?>
```

This PHP script gets the logged-in user's ID from the session. It reads the ./history directory, then filters the list of files to find only those whose names start with totalanalyzer_ followed by that user's ID. Finally, it outputs this list of filenames as a JSON array.

2. The JavaScript to Fetch and Display the List (in total_chatbot.html)

```
async function updateChatHistoryList() {
 // Element where the list will be displayed
 const historyContainer = document.getElementById('chatHistoryList');
 historyContainer.innerHTML = ''; // Clear the current list

 // Fetch the list of files from the server script
 const response = await fetch('get_totalanalyzer_history.php');
 const history = await response.json(); // Parse the JSON response

 if (history.length > 0) {
   // Loop through each filename received
   history.forEach(file => {
     const li = document.createElement('li');
     const link = document.createElement('a');
     link.classList.add("chatlistlink"); // Apply styling

     link.href = "#"; // Use # as a placeholder, action handled by JS
     link.textContent = file; // Display the filename
     link.addEventListener('click', (event) => {
       event.preventDefault(); // Stop the link from changing the page
       loadChatContent(file); // Call function to load this specific file
     });

     li.appendChild(link); // Put the link inside a list item
     historyContainer.appendChild(li); // Add the list item to the display area
   });
 } else {
```

```
      historyContainer.innerHTML = '<li>No chat history available.</li>'; // Message if no files found
  }
}

// Call this function when the page finishes loading
document.addEventListener('DOMContentLoaded', function () {
  updateChatHistoryList();
});
```

This JavaScript function fetches the list of filenames from get_totalanalyzer_history.php. It then dynamically creates HTML list items (<li>) with clickable links (<a>) for each filename. When a link is clicked, it calls the loadChatContent function, passing the filename.

Loading the Content of a Specific Chat

When you click on a filename in the list:

1. **Request Content (Browser -> Server):** JavaScript sends a request to a third PHP script (*load_totalanalyzer_content.php*), specifying the name of the file whose content is needed.
2. **Read File (Server):** The PHP script (*load_totalanalyzer_content.php*) receives the filename, locates the file in the ./history folder (again, checking user ID and potentially path to prevent security issues), and reads its entire content.
3. **Send Content (Server -> Browser):** The PHP script outputs the raw text content of the file.
4. **Display Content (Browser):** JavaScript receives the text content and replaces the current content in the *#chathistory* div with the loaded text.

Looking at the Code (Loading)

1. The JavaScript to Trigger Loading (in total_chatbot.html)

```
async function loadChatContent(filename) {
  // Fetch the content of the specific chat file
  const response = await
fetch(`load_totalanalyzer_content.php?file=${encodeURIComponent(filename)}`);

  if (response.ok) {
    const content = await response.text(); // Get the file content as text
    const chatContainer = document.getElementById("chathistory");
    chatContainer.innerHTML = content; // Replace current chat display with loaded content
    chatContainer.scrollTop = chatContainer.scrollHeight; // Scroll to the bottom
  } else {
    alert('Failed to load chat content.');
  }
}
```

This JavaScript function is called when a filename link in the history list is clicked. It sends a fetch request to load_totalanalyzer_content.php, including the filename as a URL parameter. If successful, it gets the text response and sets the innerHTML of the #chathistory element to this loaded text, effectively displaying the old conversation.

2. The PHP Script to Load the Content (in load_totalanalyzer_content.php)

```
<?php
session_start(); // Start session

// Ensure user is logged in and user ID matches the file owner (added security check)
if (!isset($_SESSION['user']['Id'])) {
 http_response_code(403); // Forbidden
 echo "User not authenticated.";
 exit;
}

// Check if filename is provided in the URL
if (isset($_GET['file'])) {
 // Use basename() to prevent accessing files outside the ./history folder
 $filename = basename($_GET['file']);
 $userID = $_SESSION['user']['Id']; // Get user ID from session

 // Construct the full file path
 $filePath = "./history/" . $filename;

 // IMPORTANT Security Check: Ensure the file belongs to the logged-in user
 if (strpos($filename, "totalanalyzer_$userID") !== 0 || !file_exists($filePath)) {
  http_response_code(404); // Not Found (or 403 Forbidden)
  echo "File not found or not accessible.";
  exit;
 }

 // Check if the file exists and belongs to the user
 if (file_exists($filePath)) {
  // Set header and output the file content directly
  header('Content-Type: text/plain');
  readfile($filePath);
 } else {
  http_response_code(404); // Not Found
  echo "File not found.";
 }
} else {
 http_response_code(400); // Bad Request
 echo "No file specified.";
}
?>
```

This PHP script receives the requested filename via the URL. It constructs the full path and includes an important security check (strpos($filename, "totalanalyzer_$userID") !== 0) to ensure the requested file name starts with the currently logged-in user's ID prefix, preventing users from trying to load other users' files. If the file exists and passes the check, it sets the Content-Type header to plain text and outputs the file's content using readfile.

Key Concepts Summarized

- **Saving:** Client-side JavaScript collects chat content and sends it to a server-side PHP script (*save_totalanalyzer_history.php*). The PHP script creates a unique, user-specific filename and saves the content to a file in the ./history directory.
- **Loading (List):** Client-side JavaScript requests a list of files from another PHP script (*get_totalanalyzer_history.php*). The PHP script reads the history directory, filters files by user ID, and returns the list. JavaScript displays this list.
- **Loading (Content):** When a user clicks a filename, JavaScript requests the content from a third PHP script (*load_totalanalyzer_content.php*), specifying the filename. The PHP script reads the file (with security checks) and sends the content back. JavaScript replaces the current chat display with the loaded content.
- **User Association:** Filenames are constructed to include the user ID, and server-side scripts check the user ID to ensure users can only access their own saved chats.

Conclusion

You've now learned how VivacityGPT Complete handles **Chat History Persistence**. By combining JavaScript in the browser to manage the display and send requests, and PHP scripts on the server to save, list, and load text files, the project allows you to keep a record of your important conversations with the AI beyond the current session. This turns the temporary *chatMemory* into a persistent record, much like saving and opening a document.

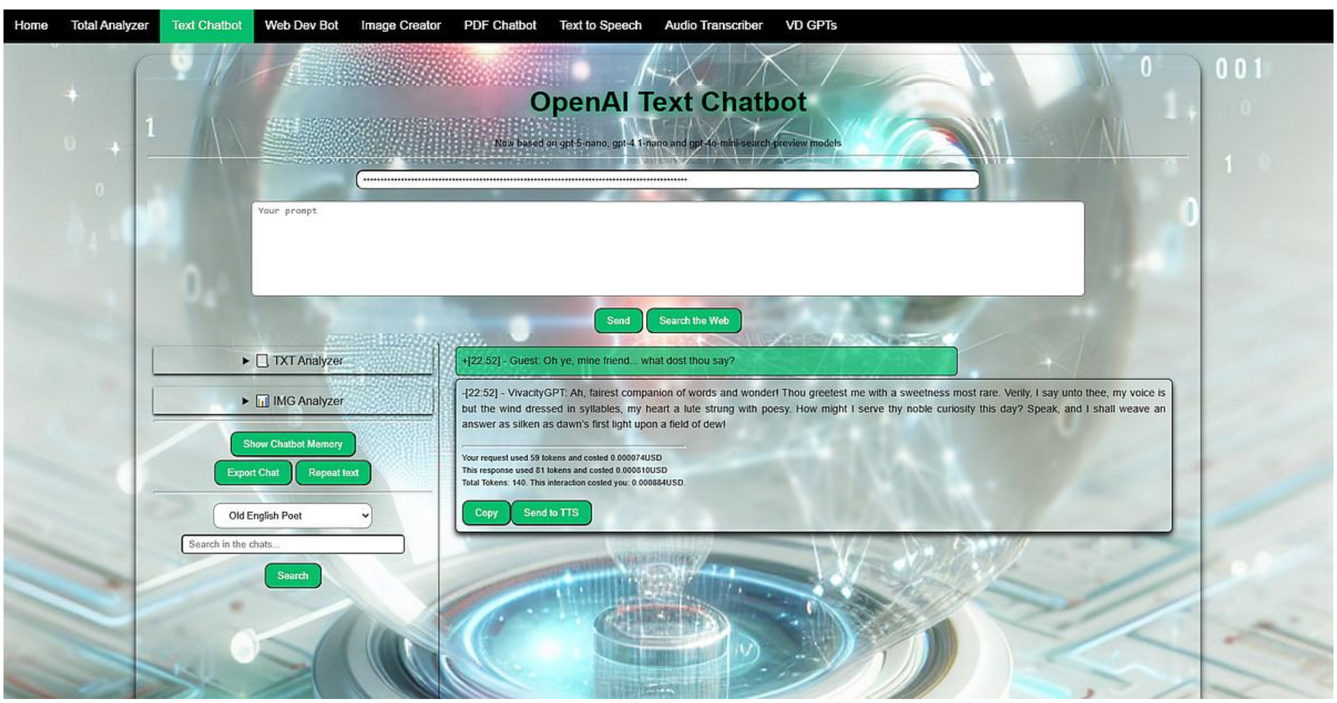
This chapter concludes our detailed exploration of the core concepts behind VivacityGPT Complete. We've covered everything from what you see on screen to how the project talks to powerful AI services and manages your data.

NOTE: this handbook is referred to **VivacityGPT Complete v.18** -october 5th - 31st, 2025

VivacityGPT Complete Image Gallery

1) Homepage

2) Text Chatbot acting as a British XVII century poet:



3) The complete GPTs list

- **AI Blog Post Creator** - Insert a topic and generate the text, image and voice files for your blog.
- **AI Psychoanalyst** - Write your dream, and generate a complete analysis and an oniric image.
- **AI Image Rebuilder** - Upload an image, get the description and produce a new variant.

. **AI Image Modifier** - Upload an image, insert a description and have the image modified or a variant created according to your instructions

- **AI Transcript Creator** - Insert your movie transcript and generate an audio file. Up to 6 actors!
- **AI Language Teacher** - Learn the fundamentals of 12 languages in 14 straightforward lessons!
- **AI Voice Chatbot** - Advanced chatbot for voice chats, with web search and image generation

- **AI Art Critic** - Upload an image with your piece of art, and get a critic analysis!
- **AI Interface Analyzer** - Upload a screenshot of your webpage or app and get a full professional analysis.
- **AI Website Builder +** - Describe the design of your landing page and generate the visual Front-End and full code (2 versions: Base & Plus).
- **AI Book Plot Creator** - Insert a title, a genre and a brief plot, and let the GPT generate your book plot in details.
- **AI Virtual Sommelier** - Plan your perfect dinner following the Virtual Sommelier advices for courses and wines.
- **Skynet** - Chat with the most iconical and aggressive artificial intelligence ever!
- **AI WebSlide Creator** - Generate WebSlides for your tutorials with Dall-E and export them in HTML format.
- **Xora Video Generator** - Generate stunning high quality video using OpenAI Sora 2/Pro models, from text and images
- **AI Web Agency** - Watch a frontend and a backend developer chatbots collaborate to make the website of your dreams come to life! It's Vivacity Design's first experiment with *collaborative chatbots pipelines*.
- **AI Master Debater** - Insert a hot topic, watch the two chatbots debate in front of your eyes, and decreet the winner. Eight personalities!

Full list of models used in VivacityGPT Complete v.18: gpt-5-nano, gpt-5-chat-latest, gpt-4.1-nano; gpt-4.1-image1, gpt-4o-mini-search-preview, dalle-e-3, chatPDF, gpt-4o-mini-tts, TTS, Whisper, Sora2

AI Website Builder + in tablet mode

Pomezia(RM) - Dec. 15th, 2025 - by Alessandro Demontis