

Shelly

TECHNICAL DOCUMENTATION



By VLAD STEFANESCU

GreenCoders®

Part of: Servicii Virtuale Media SRL®

INTELLIGENT
AI AGENT

Shelly App – Technical Documentation

Introduction

Shelly is a hybrid AI automation assistant for Windows, combining the intelligence of GPT-4 with local execution capabilities. It enables users (especially developers, power users, and early adopters) to automate complex tasks through natural language. The mission of Shelly is to **plan and perform multi-step tasks** on a Windows PC – such as running scripts, searching the web, editing files, or analyzing on-screen content – all by interpreting a user’s request in plain English. Shelly is **not targeted at enterprises** or non-technical end-users; instead, it’s designed for individual developers and collaborators who seek a powerful, extensible assistant they can tweak and trust in a local environment.

Shelly is an open-source project (released under a Creative Commons BY-NC license) and welcomes contributions. This documentation provides a comprehensive technical overview of Shelly’s architecture and components. It is structured to guide developers and advanced users through Shelly’s design, including its GPT-driven planning brain, execution core, custom function library, and user interface. We’ll discuss how Shelly plans tasks, executes them securely, handles errors, and how new capabilities (custom “tools”) can be added. Real-world examples are provided to illustrate Shelly’s use cases, like batch file editing, script generation, and screen content analysis.

System Overview

Shelly employs hybrid **GPT-agent architecture**. At a high level, Shelly works as follows: a user enters a command or question in natural language; Shelly uses GPT-4 (via OpenAI’s API) to **plan** how to fulfill the request; and then Shelly’s local **execution engine** carries out the plan step by step on the user’s machine. This approach lets Shelly handle multi-step procedures automatically (“auto-mode”) – for example, searching for information and then creating a file with the results – without further user intervention. If a request doesn’t require any external actions, Shelly can answer directly with GPT. Otherwise, Shelly will combine **AI-generated** instructions with **local actions** (PowerShell scripts or custom VB.NET functions) to complete the job.

Figure: High-level architecture of Shelly's GPT-powered agent. The user's request goes to a Planner module (CallGPTBrain) which uses GPT to produce a JSON plan. That plan is then executed step-by-step by the Executor (CallGPTCore & local functions). The executor may call custom VB.NET functions (tools) or run PowerShell scripts. Dashed lines indicate interactions with the OpenAI API (for planning, content generation, or fixes). Solid lines indicate control flow on the local machine.

Under the hood, Shelly maintains a conversation with the GPT model to accumulate context about what it's doing. It uses two main AI calls: one to a **Planner** (which formulates a series of actions in JSON format), and another to a **Core Executor** (which handles general queries or content generation using GPT). The "brain" of Shelly (the Planner) decides **which tools or scripts to use** to satisfy the request. The "hands" of Shelly (the Executor) then invoke those tools on the local system. Shelly's architecture thus balances cloud AI intelligence with local execution power. Crucially, the AI is **not given free rein on the system**; it can only perform actions through a predefined set of tools (custom functions or scripted commands) that developers have explicitly implemented. This ensures that Shelly's capabilities are extensible yet constrained to intended operations.

In practice, when the user submits a prompt, Shelly will either:

- **Answer directly** (for purely informational queries) using the GPT model (Shelly will return a simple answer via a "FreeResponse" tool), or
- **Generate a plan** of one or more steps if the request requires actions. The plan might include custom function calls (like reading a file or taking a screenshot) and/or PowerShell commands. Shelly will then execute each step in order automatically.

Shelly's system is designed for **multi-step task automation**. It loops through planning and execution until the user's request is fully satisfied. After completing a series of steps, Shelly can even summarize what it did for the user. For example, if the user asked Shelly to perform a series of file operations and web searches, once done Shelly can present a friendly summary of all actions carried out. This makes the interaction feel natural and informative.

Planner: *CallGPTBrain* (Task Planning with GPT-4)

The planning phase is handled by the *CallGPTBrain* function (located in **AIBrainiac.vb**).

This function acts as Shelly's "brain," formulating a step-by-step game plan to achieve the user's request. It leverages OpenAI's Chat Completion API (specifically GPT-4, often with the 32k token context or higher) possibly via the OpenAI *Beta Assistants* mechanism. The planner uses a system prompt that embodies Shelly's core instructions and toolset, and it sends the user's query in that context.

How *CallGPTBrain* works: When invoked, it first retrieves the assistant's preset profile (if using the Assistants API) including any built-in instructions and the recommended model. It then constructs a message list for the chat completion call:

- A **system message** with Shelly's fixed role and guidelines. For example, Shelly's system prompt tells GPT: *"You are Shelly, a powerful Windows assistant. Decide which actions to take – via PowerShell or custom VB.NET functions – to fulfill the user's request fully and sequentially. Always use custom functions when available, otherwise use PowerShell in fenced code blocks. Never reveal your internal process."* This ensures the AI knows it should output actions, not just an answer.
- The assistant's own additional instructions (if any were fetched via the Assistants API).
- Recent **conversation history** (the last N messages, to maintain context). Shelly caps this to a certain number of messages (by default 20) to fit within model limits.
- A **user message** describing the planning task. Shelly formulates a special user prompt that includes: the original user request, a list of any tools already executed so far (if in a loop), and explicit instructions on how the AI should respond. Notably, Shelly instructs the AI to output either a direct answer or a JSON array of steps. For example, the planning prompt tells GPT:

text

ORIGINAL REQUEST: <user's request here>

TOOLS ALREADY FINISHED (with full **args**): <none or list of completed steps>

- If you can answer entirely in natural language, return a single step using tool="FreeResponse" with args.text set to the answer.
- If the answer requires ****running or evaluating PowerShell****, use tool="ExecutePowerShellScript" and place the script inside **args.script** (wrapped in a fenced block).
- Otherwise list ALL remaining tool steps in order. Return only the JSON array (no markdown fences, no commentary).

(The above is a simplified representation of Shelly's planner instructions.)

When CallGPTBrain sends this prompt to the OpenAI API, GPT-4 will ideally respond with a **JSON-formatted plan**. For example, GPT-4 might return:

```
json
[
  {"tool": "ReadFileAndAnswer", "args": {"filePaths":
"C:\\\\Logs\\\\app.log", "query": "summarize errors"}},
  {"tool": "GenerateBatchAndPs1File", "args": {"outputFolder":
"D:\\\\Demo", "userQuery": "Check disk space and list largest
files"}}
]
```

Processing the plan output: Shelly's code examines the raw reply from GPT. It strips away any markdown formatting to isolate the JSON. Then it attempts to deserialize the JSON string into a list of PlanStep objects.

There are a few possibilities at this stage:

- **Valid Plan:** If parsing succeeds and yields a non-empty list of steps, Shelly proceeds with those steps.

- **Single FreeResponse:** If the plan is a one-step plan and that step is a FreeResponse, Shelly interprets it as an answer. In this case, Shelly will simply display the `args.text` from that step as the assistant's answer to the user (without running further tasks).
- **Multi-step Plan:** If the plan contains multiple steps, Shelly enters multi-task execution mode (described in the next section). It sets a flag `lastRunMultiTask = True` to remember that multiple actions were taken.
- **No JSON / Unexpected Output:** If GPT's response can't be parsed as the expected JSON (for example, the model might have returned a narrative answer or incorrectly formatted plan), Shelly falls back to a more forgiving approach. In this fallback, Shelly passes the raw AI response to a multi-step handler that will **parse any embedded code blocks or function calls from plain text** (treating the response as if it were a conversation containing instructions). This ensures that even if the planner didn't follow the format strictly, Shelly can still attempt to execute any detectable actions from the reply.

Shelly logs the raw plan for debugging purposes, so developers can inspect what the AI proposed. Any error in parsing is caught, and if no executable steps are found, Shelly will ultimately just show an error or do nothing.

Role of the Planner (CallGPTBrain): In summary, the planner's job is to translate an open-ended user request into a structured game plan. It knows about all the **available tools** (custom functions and the special `ExecutePowerShellScript` and `FreeResponse` actions) and instructs GPT-4 to use them appropriately. The planner is stateless except for the chat history it carries – importantly, it does not itself execute anything. It simply returns a plan. By centralizing decision-making in GPT-4 (which has knowledge of the task and tool descriptions), Shelly can easily be extended with new tools: update the planner's prompt to include the new function, and GPT-4 can start using it in plans. (We will discuss extensibility in a later section.)

The use of OpenAI's Assistants API means the exact model (and possibly some predefined high-level instructions) can be managed via an "assistant profile"; for example, Shelly's assistant profile could specify GPT-4-32k to allow very large plans or context, and include a list of tool definitions for GPT. The `CallGPTBrain` function makes sure to insert Shelly's core rules at runtime so that even if the assistant profile changes, Shelly's fundamental policies (use tools, don't reveal system prompt, etc.) are enforced on every plan request.

Executor: *CallGPTCore* and Task Execution Flow

Once a plan is obtained, Shelly's **Executor** takes over to carry out each step. The executor involves a few components working together:

- The **CallGPTCore** function (in **Alcall.vb**) is a general-purpose routine to call the OpenAI API for **non-planning purposes** (e.g., getting the content of a text request, or asking GPT to fix a script error). It's essentially Shelly's interface to GPT-4 for everything aside from the main planning step.
- The **ExecutorAgent** (part of Shelly's logic in **Shelly.vb** and possibly **ExecutorAgent.vb**) which iterates through the plan steps and invokes the appropriate local action for each.
- The **CustomFunctions** module (in **CustomFunctions.vb**), which implements the set of custom VB.NET functions (tools) that the plan might reference.
- A **PowerShell runner**, which executes any raw PowerShell scripts that the plan includes, with error handling and retries.

CallGPTCore (Alcall.CallGPTCore): This function is a wrapper around the OpenAI chat completion API with some important features:

- It will automatically inject Shelly's system prompt (the same one described earlier) into the message list if no system message is present. This is a safeguard to ensure even ad-hoc GPT calls adhere to Shelly's rules (for instance, if a custom function directly queries GPT, Shelly still reminds the model about being a Windows assistant, etc.).
- It implements **dynamic token budgeting**. Shelly defines a very large context window (up to 128k tokens input, 16k output in code, anticipating future model capabilities). It calculates how many tokens the input messages might consume (estimating ~4 characters per token) and then sets `max_tokens` for the completion accordingly. For example, if the messages already use 10k tokens, it might allow up to 6k for the answer (or less if hitting model limits). This prevents GPT from generating responses that exceed context or from refusing due to length.
- It sets OpenAI API parameters like temperature, `top_p`, etc. (Shelly often uses a moderate temperature ~0.7 for creative tasks, or 0.0 for deterministic tasks like code generation or extraction).
- It handles **API errors and retries**. If the HTTP request to OpenAI fails or times out, **CallGPTCore** will catch exceptions and try up to 3 times with a short backoff. Certain errors like invalid API key or model not found are caught and returned as error messages (e.g., "[ERROR] Model not found") without retry.

- On a successful API call, it extracts the assistant's response text and returns it as a trimmed string. It also stores the actual model used in a global variable for reference (since OpenAI may roll over to a compatible model, this is logged for transparency).

In essence, CallGPTCore is the reliable messenger: it sends a prompt and gets a response from GPT, handling all the low-level details (headers, JSON payload, error cases). The Executor will use this whenever it needs GPT's help during execution (for example, to answer a sub-question or to correct a PowerShell script).

Executing the Plan: After planning, Shelly's main loop (in **Shelly.HandleUserRequestAsync**) receives the list of PlanSteps. It then calls `ExecutorAgent.ExecutePlanAsync(plan)`, which runs each step in order. Each step has a Tool name and an Args dictionary. The execution flow does the following for each step:

1. **Identify the tool type:** Shelly checks the Tool field to decide how to execute it:
 - If the tool corresponds to a custom VB.NET function (one of Shelly's built-in functions in `CustomFunctions.vb`), Shelly will call that function.
 - If the tool is "ExecutePowerShellScript", Shelly will extract the script from Args and run it in PowerShell.
 - If the tool is "FreeResponse", it means this step is just a natural language answer. Shelly will output the text in `args.text` to the user interface.
 - (If the tool was "TextRequest" or similar, which could be used internally for GPT-only queries, Shelly would use CallGPTCore to fulfill it. In the JSON plan format, typically only FreeResponse is used for direct answers; other purely text operations might not appear as a tool but could arise from the fallback parser.)
2. **Ensure no duplicate executions:** Shelly uses a hash set `executedCalls` to record each function call or script it has executed in the current run. If the plan (or GPT output) happens to list the **same exact action twice**, Shelly will skip the duplicates to avoid redundant operations. For custom functions, it uses the function signature string (e.g., `"ReadFileAndAnswer(\"C:\\file.txt\", \"Find X\")"`) as the key; for PowerShell scripts, it hashes the script content as the key (since scripts might be long). This deduplication protects against scenarios where the AI might inadvertently repeat an instruction.

3. **Execute the step:** Shelly executes according to type:

- **Custom Function Call:** Shelly invokes `ExecuteAppFunctionAsync(signature, ct)` which looks up the function by name and calls it on a background thread (allowing async operations). All custom functions return a `Task(Of String)` – i.e., an asynchronous result string. Shelly awaits the result. If the function returns a non-empty string, Shelly will append that to the result output box for the user to see. Many functions return some message or result text; some functions might return an empty string to indicate they handled their own UI update or have no user-facing output. After a custom function executes, Shelly sets a flag `skipNextPlainTextSegment = True`. This flag is used because GPT sometimes includes a descriptive text after a function call; Shelly uses it to *suppress the next text segment* if it looks like a redundant summary the AI provided. (For example, the AI might have planned: `GenerateImages(...)` followed by a text segment like “Here are the images I generated.” Shelly will execute `GenerateImages`, then skip printing the follow-up text “Here are the images...” since it’s unnecessary – Shelly instead directly shows the list of image files generated.)
- **PowerShell Script Execution:** Shelly passes the script content to `ExecutePowerShellWithFixLoopAsync` (or similar internal method) which runs the script in a sandboxed PowerShell process. This process is created with user-level permissions (Shelly does not require admin rights unless the script itself needs to do admin tasks and the user runs Shelly as admin).
- Shelly typically uses PowerShell’s `-NoProfile` mode to avoid user-specific profile scripts, ensuring a clean environment, and may use `-ExecutionPolicy Bypass` so that even if the system has a restrictive policy, Shelly can run the script. The output and errors from the script are captured. Shelly then enters a loop to handle errors: if the script failed (non-zero exit or any error text), Shelly will ask GPT to **debug/fix the script** on the fly.
- It does this by calling `Alcall.CallGPTCore` with a prompt like: “I tried running this PowerShell script but got an error: <error message>. Script: powershell ... Please return only a corrected script in a powershell block.”. GPT will respond with a modified script (hopefully fixing the issue). Shelly then extracts the corrected script from the response and tries to run it again. Shelly will retry up to a few times (by default 3 attempts) for a failing script, each time feeding back the error to GPT and getting a fix.

- This **auto-debugging loop** is a key feature: it allows Shelly to handle cases where the initial command might not work due to environment specifics, missing modules, minor syntax issues, etc. If after the maximum attempts the script still fails, Shelly stops and logs a failure. On success, Shelly captures the script's output (if any) and displays it to the user. Successful or not, the script step is marked done and (as with functions) Shelly sets `skipNextPlainTextSegment = True` to avoid printing any AI-generated explanation that might have accompanied the script.
 - **FreeResponse (Natural Answer):** Shelly simply takes the provided text and appends it to the chat output as the assistant's answer. This typically happens when GPT determined no action was needed beyond explanation. Shelly ensures to log this and avoid duplicating any custom function output.
4. **Intermediate updates:** After each step execution, Shelly may insert the output as a message in the conversation history (so that the AI can reference what happened if the planning loop continues). The UI's status label is updated to reflect progress (e.g., "Executing planned tasks..."). A small delay or yield is introduced to keep the UI responsive. The user can cancel at any time using the Cancel button – Shelly checks a cancellation token `ct.IsCancellationRequested` at various points in the loop, ensuring it can abort long operations if the user requests.
 5. **Post-plan summary:** If Shelly executed multiple steps for the request, at the very end it triggers a special summary generation. It formulates a new prompt for GPT: essentially "You (Shelly) just finished running multiple tasks based on the user's prompt. Please give a short, friendly, dorky summary of what was done. If there were errors, mention them with a tone of trying your best.". It uses a GPT call (`CallGPTBrain` again in this case) to get a single response summary, which it then displays to the user in the results box. This summary is purely for the user's benefit and has no further actions. After this, Shelly resets its state for the next user prompt (clearing the multi-task flag, etc.).

While executing, Shelly logs debug information for each step (including outputs or any [ERROR] messages returned) for developers to inspect if needed. It also continuously trims the stored conversation history to avoid exceeding memory or token limits, using `TrimConversationHistoryByTokens` after significant additions.

Multitasking and concurrency: Shelly's design currently executes tasks **sequentially** (one after the other in a single thread of execution, aside from awaiting async calls). It does not run multiple plan steps in parallel – this simplifies dependency management (later steps might

depend on earlier ones, e.g., reading a file after it was created). “Multitasking” in Shelly refers to the ability to handle a **sequence of tasks automatically**, rather than simultaneous tasks. Each planning loop can handle multiple actions, and Shelly can even go through multiple plan-execute cycles if the assistant chooses to (though typically one cycle is enough per user query). During execution, the UI remains responsive and the user can cancel if needed, but they cannot issue a new query until the current one finishes (the input is disabled during run).

Error handling and logging: Shelly is robust in catching exceptions:

- If a custom function throws an exception, Shelly catches it and returns a safe error message like [ERROR] <exception message> to the result box instead of crashing.
- If something unforeseen happens in the main loop, Shelly catches it and shows a status “Error: <message>” in the UI status label.
- All errors and debug info are printed to Debug.WriteLine (which developers can see in Visual Studio’s Output window if running in debug) and some are stored in an internal log or the console form for later review.
- The planning step also handles error conditions, for example if the Assistants API fails or returns nothing, CallGPTBrain returns an error string which Shelly will display to the user (so that the user knows the planning failed).

After all tasks (or on error), Shelly re-enables the UI and resets the cancel button. The overall flow then waits for the next user input.

AI Model Selection and Auto-Mode Logic

Shelly is built to work best with **GPT-4** (particularly models with large context windows). By default, it will use the model indicated by the Assistant profile or a configured global model. In practice, this means Shelly will typically call gpt-4-32k if available, to handle large inputs, but it can fall back to gpt-4 or even gpt-3.5-turbo if configured.

The variable `Globals.AiModelSelection` (and similarly `Globals.UserApiKey`) is used to choose the model for `CallGPTCore` calls. This can be set by the user in a configuration UI or defaults. Shelly’s code suggests an adaptive approach: it estimates token usage and if something is too large for the current model, it might warn or adjust. For instance, when performing web page reading or file reading, Shelly will attempt a single-call extraction if the text fits the model’s context; if not,

it automatically falls back to a chunk-by-chunk processing. This is a form of **auto-mode logic** where Shelly adapts to the model's limitations by splitting tasks.

Auto-mode in Shelly has two meanings:

1. **Automatic model/context adaptation:** Shelly will try to utilize the largest context model available to avoid unnecessary chopping of content. The `models.xlsx` file in the project likely lists model names and their token limits, guiding `AiModelSelection`. For example, if a user only has an API key for GPT-3.5, Shelly might detect that and use that model (but then certain complex tasks might not work as well or at all). On the other hand, if GPT-4 32k is available, Shelly will leverage it to handle huge texts. All of this is transparent to the user – the idea is the user just provides their API key and Shelly picks an optimal model.
2. **Automatic task execution (agent auto-mode):** Once the user issues a command and Shelly generates a plan, Shelly runs **fully automatically** through all steps. The user does not have to approve each action in a step-by-step manner (although they will see the actions/results as they happen). This design makes Shelly efficient for “one-shot” automation: the user describes an outcome, and Shelly figures out and executes the necessary procedure. Users can always break the loop by hitting “Cancel”, but otherwise Shelly assumes it has permission to carry out the plan it devised. This is in contrast to tools that might pause and ask “Do you want to execute this command?” for each step – Shelly’s philosophy is to be truly autonomous in carrying out the user’s wishes (since the user explicitly asked for it).

From a developer perspective, the model selection is handled in code by setting `Globals.AiModelSelection`. The Assistants API call (`RetrieveAssistant`) can dynamically provide a model name – for example, an assistant profile could specify `gpt-4` vs `gpt-4-32k`. Shelly uses whatever model is returned in the profile for the planning call. For the execution calls (`CallGPTCore`), Shelly uses its own `Globals.AiModelSelection` which presumably is set to the same model (or a default). There is also logic to record the actual model used from each API response, updating `Globals.LastUsedModel`, so that the UI or logs can show which model produced the output.

In summary, Shelly tries to **maximize the use of AI capabilities automatically**:

- It uses the most powerful model available for better reasoning and larger context.

- It breaks down tasks automatically and uses multi-turn interactions with the model without user intervention.
- It automatically handles when to just answer versus when to plan actions. The user does not have to explicitly toggle modes – Shelly’s planner will decide based on the request. (For instance, ask Shelly a general question and it will likely just answer; ask it to perform an operation and it will plan and execute.)

This “auto-mode” provides a seamless experience but also places responsibility on the planning logic to make correct decisions. The developer can fine-tune this by adjusting the planner’s prompt or adding new tools. If Shelly ever mis-classifies a request (e.g., tries to execute when it should just answer), that can be tweaked by modifying the instructions given to GPT in the planning step.

PowerShell Execution and Validation Loop

One of Shelly’s most powerful capabilities is executing PowerShell scripts generated by GPT. This allows Shelly to do virtually anything on the system (within the user’s permission scope) – from file system operations to system configuration – based on natural language instructions. However, running arbitrary scripts poses both reliability and security challenges. Shelly’s design addresses these with a **validation loop** and some sandboxing measures:

- **Isolated Execution:** Shelly runs PowerShell scripts in a separate process (not in the main application’s process). It likely uses `System.Diagnostics.Process` to start a hidden PowerShell instance. The script content is passed via standard input or as a `-Command` argument. Shelly uses `UseShellExecute = false` and `RedirectStandardOutput/RedirectStandardError` to capture the results. By using an external process, Shelly ensures that if a script hangs or crashes, it can still terminate it (using the `currentPowerShellProcess` handle it keeps) without bringing down the UI. Shelly also sets a **cancellation token** that, if triggered by the user pressing cancel, will attempt to kill the running PowerShell process gracefully.
- **No Persistent Changes without Intent:** Shelly launches PowerShell with “**-NoProfile**”, meaning it doesn’t execute the user’s PowerShell profile scripts. This avoids unintended side effects or malicious profile code. For the `GenerateBatchAndPs1File` function, Shelly explicitly writes files to disk and suggests running the `.bat` which uses `-ExecutionPolicy Bypass` (since that is a user-initiated action, it allows execution of the generated `.ps1` without signing). For direct script execution within Shelly, running in-memory via `-Command` typically bypasses execution policy as well, but Shelly’s use of

Bypass ensures that even if the script had to be saved and run, it would work. Essentially, Shelly assumes the user trusts the actions they've asked for and so it prioritizes executing them successfully, while containing them to the user context.

- **Safety of the Local Environment:** It's important to note that Shelly does **not** run the scripts in a constrained language mode or a VM sandbox – it runs them as the current user. This means any command the AI puts in the script will be executed with the user's privileges. Shelly does not silently run scripts from remote sources; all script content is generated on-the-fly by GPT in response to the user's query. Nonetheless, developers and users should be cautious: if Shelly is asked to do something destructive ("delete all my files"), GPT might generate exactly such a script. Shelly will execute it. Therefore, **risk is managed primarily by user intent and transparency**. Shelly ensures that the user can see what it's doing (through logs or the outcomes printed). In a future improvement, one might add a confirmation dialog for very destructive commands, but currently Shelly assumes the user's prompt is the approval.
- **Validation & Auto-Fix Loop:** The reliability of GPT-generated code is not 100%, so Shelly wraps PowerShell execution in a *validate-and-repair cycle*. After running a script, Shelly inspects the result:
 - If the script succeeded (exit code 0 and no error text), Shelly proceeds.
 - If it failed, Shelly captures the error output (for example, a PowerShell exception message). It then asks GPT (via CallGPTCore) to debug it. The prompt to GPT includes the original script and the error message, and instructs GPT to return **only a corrected script** in a markdown block. This prompt frames GPT as a "PowerShell troubleshooting assistant".
 - GPT might respond with a revised script (perhaps adding error handling, installing a missing module, correcting a command, etc.). Shelly then extracts the code from the markdown and tries running the new script.
 - This loop repeats. By default Shelly allows up to 3 attempts. Each iteration is logged (so developers can see "[PS Failure #1]: <error>... [PS Repair GPT]: <new script>" in debug logs). If after the third attempt it's still failing, Shelly stops and shows the user a failure message "❌ All attempts to run/fix PowerShell script failed.". This way, Shelly doesn't get stuck infinitely on one task.
 - If a fix attempt succeeds partway through, Shelly breaks out of the loop and continues with the next steps of the plan.

This validation loop greatly enhances safety and correctness. It means that if GPT produces a script with a syntax error or a minor mistake, Shelly won't blindly run it and move on – it will catch the error and actually attempt to correct it. From the user's perspective, this might look like Shelly "thinking" a bit longer on that step, but the end result is a higher chance of success or at least a clear error report if it couldn't be fixed.

- **Output handling:** Shelly captures whatever the script writes to standard output or error. It **cleans the output** by stripping away any Markdown code fences that might inadvertently be included (GPT might sometimes include ``` marks in responses). It then presents the output in the UI. If the script didn't produce any output but succeeded, Shelly prints a generic confirmation like "✅ Task completed. Enjoy it!" to let the user know the command ran. For errors, after exhausting retries, Shelly will show "❌ All attempts to run/fix PowerShell script failed." so the user knows that Shelly couldn't complete that step.
- **Sandboxing:** While not a full sandbox, Shelly does attempt to **limit the scope** of actions in some ways:
 - It never runs a PowerShell script unless the AI explicitly decided one was needed for the task (and thus presumably the user's query required it). If a direct answer is possible, Shelly won't run any code.
 - The available *custom functions* cover many common operations (file read/write, web search, etc.), so often GPT will choose those instead of writing a PowerShell from scratch. This is safer because those functions are coded by developers with specific, constrained behavior. For example, SearchForTextInsideFiles will only search text; it won't delete files. GPT is guided to prefer these safer, pre-defined actions.
 - The PowerShell itself runs with the same permissions as Shelly (which is typically a normal user). So it cannot do certain system-wide changes without elevation. If the user wanted to do something requiring admin rights, they'd have to run Shelly as admin explicitly. This way, a casual user running Shelly doesn't accidentally execute an admin-level destructive action unless they intended to.
- **Secure API Key Handling:** Running PowerShell also means ensuring the OpenAI API key (which Shelly holds to call GPT) is not exposed to the script. Shelly's API key is stored in memory (in Globals.UserApiKey or Config.OpenAiApiKey) and used only in HTTP calls to OpenAI. It is never passed to any shell commands or external processes. The custom functions and scripts do not need the OpenAI key (only the .NET code that calls the API uses it). Shelly does not log the API key or show it in the UI (except perhaps when the

user initially enters it in a settings panel). If the user opts to save the API key for convenience, it would be stored in a user-specific configuration file or Windows secure storage – the implementation isn't shown here, but the emphasis is that the key remains on the user's machine. In technical terms, **Shelly does not transmit the API key to any destination other than OpenAI**. Developers integrating Shelly should follow this practice: never hard-code the key, and never expose it in logs. If collaborating, use environment variables or prompt the user for their key.

In conclusion, Shelly treats PowerShell execution as a powerful tool that is used carefully:

- It is only invoked when necessary.
- It's run in a controlled way (isolated process, with error checking).
- The AI's output is validated and corrected if possible.
- The user's system is respected (no privilege escalation, no unwanted persistence).
- At any point, the user can see what's happening (and they can always look at Shelly's logs to audit the exact script that ran, if desired).

Key UI Components

Shelly's user interface is built with Windows Forms (VB.NET). It provides a simple, interactive front-end for the underlying logic. The key UI elements and components include:

- **Main Window (Shelly.vb):** This is the primary form that users interact with. It contains:
 - An **input textbox** (UserInputBox) where the user types commands or questions.
 - A "Run" button (or the user can press Enter) to submit the query.
 - A **rich text output box** (ResultBox or similar) where Shelly displays conversation history and results. This box shows the user's queries and Shelly's responses (including any output from tasks or errors). The text is color-coded (the code calls JavaScript setColorGreen() etc., likely to distinguish AI text).
 - A **Cancel button** (CancelTaskButton) that is enabled when a task is running. The user can click this to interrupt an ongoing multi-step process. Internally, this triggers a cancellation token that the executor checks, leading to aborting further steps and printing "[Canceled by user]" if caught in time.

- A status label (LabelStatusUpdate) at the bottom that shows current status messages (e.g., “Running PowerShell script... Attempt #2” or “All tasks completed.”). This gives feedback on what Shelly is doing.
- Possibly a small panel or indicator showing whether Shelly is in “auto” mode or not (in our case, auto-mode is always on by design, so no toggle UI, but there is a collapsible panel used for additional options).
- The main form also hosts an **off-screen WebView2 browser component** (WebView21 from Microsoft Edge WebView2) which is not visible to the user but used by certain functions (like web search or web page reading). This browser control is created at runtime and never shown; it’s purely for programmatic web automation. After such functions run, Shelly disposes of the control.
- Menu or buttons for settings: e.g., an option to open a settings dialog (perhaps “Default Folder” for setting where to save files or screenshots), an “About” dialog, etc. The file list suggests there is an **About form** and a **DefaultFolder form**. The DefaultFolder.vb likely lets user choose a directory for Shelly to use for file outputs (the code references C:\ShellyDefault in examples, possibly a default working directory).
- Shelly’s main window is styled with a custom title bar and can be dragged, etc., as seen in code that handles form border and mouse events.
- **Console Window (Console.vb):** Shelly includes a secondary form called “Console”. This console is aimed at developers or power users for debugging and advanced control. It likely displays the debug log or allows executing custom functions manually. The presence of CustomFunctionsEngine.vb (FOR CONSOLE HELP) in comments indicates that the console might list all available custom functions and their usage (perhaps from the <CustomFunction> attributes defined on them). The console could allow a user to type a command like ToolPlanner.ListFunctions or directly invoke a function by name for testing. In essence, the Console is a sandbox/testing UI for Shelly’s capabilities – useful during development to ensure a function works as expected.

Security and Privacy Considerations

Security: All automation tasks executed by Shelly occur on the user’s local machine under the user’s permissions. Shelly does not perform any action unless it was explicitly part of the AI-generated plan (which in turn is triggered by the user’s request). This means **Shelly won’t**

arbitrarily run code or access files unless the user asked for it in their prompt. PowerShell scripts are run in a controlled environment (a child powershell.exe process with no user profile loaded). The scope of potential changes is limited by the user's own system rights – Shelly won't escalate privileges on its own. If Shelly is run as a normal user, for instance, a plan to modify system files will likely fail (and Shelly will report the error). In general, **risk is managed by transparency and containment:**

- The user can see the outcomes of each step (and developers can log or display the actual commands if needed).
- The planning logic biases towards using predefined functions for known tasks (which are safer and more predictable).
- Arbitrary script execution is used as a fallback and is monitored through the retry/fix loop.
- There is a cancel mechanism to stop runaway processes. Also, after 3 failed attempts at a script, Shelly stops trying further fixes to avoid any infinite loop or unintended side effects.

It's still possible for Shelly to execute a harmful command if a user deliberately or accidentally requests it (for example, asking "Shelly, wipe my temp files" could lead to a script that deletes files). **Developers and users should treat Shelly with the same caution as a powerful scripting tool.** Always review what you ask it to do. In a collaborative setting, you might add additional safeguards (like confirmation prompts for dangerous operations, or a sandbox mode restricting file write/delete operations). As of now, Shelly leaves the responsibility to the user's intent – it executes faithfully what was requested.

Privacy: Shelly is designed to **not "peek" at any user data** unless it is required to fulfill a command. The AI (GPT-4) does not have inherent access to your files, clipboard, screen, or network – it only knows what Shelly tells it. Shelly only sends data to the OpenAI API that the user has explicitly asked to be processed. For example:

- If you ask Shelly "*What's on my screen?*", it will intentionally capture a screenshot and send it to the AI for analysis (because you requested that).
- If you never ask such a thing, Shelly will never arbitrarily capture or analyze your screen or files in the background.

In other words, **the AI isn't inspecting local content on its own.** It generates automation steps based on the user's query and predefined tools. Each tool has a limited, specific purpose (like

“read a given file path” or “search for a keyword in these files”). There is no tool that just says “scan the entire computer” or “upload my documents” unless the user explicitly provided such broad instructions. This ensures a level of privacy by design – Shelly’s creators deliberately constrain the AI with only certain abilities. If a task *does* involve personal data (e.g., reading a file or the screen), that data is sent to OpenAI’s servers to get the AI’s response. Users should be aware of this and avoid using Shelly on highly sensitive data unless they are comfortable with OpenAI processing that information. (For instance, don’t ask Shelly to read a confidential document if you don’t want the document’s text to leave your machine.) All communication with the OpenAI API is encrypted (HTTPS), and the API key is kept local as described earlier.

Finally, Shelly does not collect telemetry or send your prompts anywhere except to OpenAI for the completion. The code is open-source, so developers can verify there are no hidden data transmissions. Any logs remain local. In summary, **Shelly respects user privacy** by only acting on data when instructed, and even then, handling it as transparently as possible.

Extensibility and CustomFunctions

A key goal of Shelly is to be **extensible** – developers can add new capabilities (new “tools”) by writing custom VB.NET functions and integrating them into the planning/execution system. The file **CustomFunctions.vb** is central to this extensibility. It contains a library of static Public Async Function ... As Task(Of String) methods, each implementing a specific action that Shelly (via GPT) can take. These range from manipulating files, to simulating key presses, to performing web searches.

Each custom function typically returns a result string (which may be shown to the user) or an “[ERROR]...” message if something goes wrong. They often use other helper classes (like a FileHandler for clipboard operations, or WebView2 browser control for web content). To make a function available to the AI planner, the developer must do a few things:

- **Implement the function in CustomFunctions.vb:** The function should be Public Async Function <Name>(args...) As Task(Of String). Keeping the signature simple (usually string inputs, maybe optional CancellationToken) is advisable. The function should catch its own exceptions and return an “[ERROR] ...” message on failure (this prevents crashes and allows graceful error handling).

- **Add a [CustomFunction(...)] attribute** above the function (this is a custom attribute likely defined to hold metadata). Shelly's code uses attributes to provide a human-readable description and an example usage of the function. For instance, in CustomFunctions.vb you'll see annotations like:

```
vb
<CustomFunction("Searches file(s) or folder(s) for a specific
string ...",
    "SearchForTextInsideFiles(\"C:\\Folder;C:\\File.txt\",
\"search term\")">
Public Async Function SearchForTextInsideFiles(paths As String,
searchWord As String) As Task(Of String)
```

The first string in the attribute is a description (used possibly for documentation or console help), and the second is an example of how to call it. These attributes are not directly used by GPT (since GPT is just fed text), but they can be used by the **ToolPlanner or Console to list available commands**. They serve as a form of documentation and could be included in the planner's prompt.

- **Register the function in the planner's tool list:** Shelly's planner needs to know the names of available tools (functions). In the code, there is a hardcoded array of function names used by the parse]. The developer must add the new function's name to that list (and anywhere else the project notes, e.g., a ToolPlanner module or ExecutorAgent). This array is used to detect function calls in GPT's responses. If it's not updated, GPT might output a function name that Shelly fails to recognize. Similarly, the ToolPlanner (which crafts the prompt for GPT) should include the new tool in the instructions. Shelly likely has a section (perhaps in ToolPlanner.vb or the assistant's profile) where it describes each tool's purpose to GPT. After adding a function, you'd append an entry there describing when to use it.
- **Integrate in FunctionRegistry/Executor:** Shelly uses a FunctionRegistry or CustomFunctionsEngine to dynamically invoke these functions by name. Often this is done via .NET reflection or a manually maintained map from string to Func. When adding a new function, if reflection is used, it might be picked up automatically (for example, the code could reflect all methods with the CustomFunction attribute and register them). If not, the developer would manually add a case or dictionary entry mapping the string name to the actual function delegate. In Shelly's

case, the code comments indicate updates needed in **CustomFunctionsEngine.vb** and **ExecutorAgent.vb** when new functions are added, which suggests a bit of manual wiring.

- **Testing:** After adding the function, one can use the Console window to test it (if the console provides a way to call the function directly by name) or simply run Shelly and ask for it. For example, if you added `TranslateText(enPhrase, targetLanguage)`, you might prompt Shelly: "Translate 'hello world' to Spanish." If everything is set up, GPT-4 should include `TranslateText("hello world", "Spanish")` in its plan, and Shelly will attempt to execute it.

ToolPlanner logic: To elaborate, the **planner (CallGPTBrain)** essentially needs to be aware of the tools. Typically, the assistant's system-level instructions or the user prompt for planning will include a brief on each custom tool. In the current Shelly implementation, the planner prompt we saw doesn't explicitly list tools, but the comment in `CustomFunctions.vb` suggests there is a `ToolPlanner.vb` that likely enumerates them. A possible approach (which Shelly likely uses) is that the assistant's OpenAI "Beta Assistant" profile associated with Shelly has knowledge of the toolset, or Shelly might prepend a hidden message like "Tools you can use: 1) `WriteInsideFileOrWindow(topic)`: writes text into the active window; 2) `ReadFileAndAnswer(filePaths, query)`: reads file(s) and answers question; ... etc." This way GPT knows the function names and how to use them. So when extending Shelly, updating this list is critical. If not, GPT might not realize a new function exists and thus won't use it.

Developers can thus extend Shelly's abilities by writing more VB.NET code, without needing to change the fundamental architecture. Want Shelly to send an email? One could add a `SendEmail(to, subject, body)` function that uses an SMTP library, then update the planner prompt to prefer that for email-related requests. As long as the function returns a string result (for success or error) and doesn't crash, Shelly can integrate it. The modularity of having distinct `CustomFunctions` means contributors can easily add features.

To summarize the extensibility steps:

1. Write the function (async, returns string) in `CustomFunctions.vb`.
2. Add a `<CustomFunction>` attribute with description and example.
3. Register the function name in the parser and planning prompt (`ToolPlanner`).
4. Ensure the Executor can call it (reflective or manual mapping).

5. Build and test – the GPT will start using it in plans once it “knows” about it.

Shelly’s architecture is flexible, but currently requires a few manual steps to add tools – in the future, this could be improved with reflection (auto-discover functions with the attribute to reduce manual sync). The documentation comments in the code serve as a checklist for developers performing this task.

Custom Function Library: Detailed Breakdown

Let’s delve into the major custom functions that come built-in with Shelly. Understanding these will illustrate how Shelly accomplishes specific tasks. We’ll describe what each function does and how it’s implemented internally:

WriteInsideFileOrWindow(topic, totalChunks)

Purpose: Types or inserts AI-generated text directly into the currently active application window (for example, into an open Notepad file or a text field in another program). This is useful for letting Shelly “paste” content into an app that the user is currently focused on.

How it works: When invoked, this function first captures the handle of the **foreground window** and the focused control within it. (It uses a `FileHandler.GetForegroundWindow()` and `GetFocusedControl` which likely call Win32 API to get the active window handle.) It then waits a few seconds to ensure the user has placed their cursor where text should go. Next, it generates the content to insert by leveraging GPT again:

- The function divides the task into chunks if `totalChunks > 1`. For each chunk (e.g., chunk 1 of 3), it prompts GPT-4 via `CallGPTCore` with a system instruction like *“You are a content generator focused on producing exact text for insertion.”* and a user message: **“You are writing part #i of N for: {topic}. Generate the exact content that should be inserted, preserving all spaces, line breaks, and formatting. Do not include any code block markers or extra commentary.”*. Essentially, Shelly asks the AI to produce the text content for that chunk.
- It receives the `chunkText` from GPT, then cleans it to ensure no stray markdown/code fences are present.
- Finally, it **inserts the text into the target window**. It does this by copying the text to clipboard and sending a paste command (Ctrl+V) or similar –

specifically, `FileHandler.PasteTextBulk(windowHandle, controlHandle, chunkText)` is used to simulate the past] . This likely injects the text via Windows messages to the target control.

The function repeats this for each chunk, resulting in potentially large amounts of text being entered. After completion, it clears the clipboard (to remove the leftover copied text] . It returns an empty string on success (because the result is directly visible in the target application, Shelly doesn't need to print anything in its own UI for this function). If something fails (no window, or an error generating content), it returns an "[ERROR]" message.

Use case: If a user asks Shelly "Write a paragraph about X in my open document," Shelly can plan to call `WriteInsideFileOrWindow("paragraph about X")`. The result will be that the paragraph is typed into whatever document the user currently has focused (for example, Word, Notepad, an email composer, etc.), as if the user typed or pasted it.

CheckMyScreenAndAnswer(query)

Purpose: Analyzes the user's screen (screenshot) to answer a question about what's visually present. This is like OCR + interpretation on demand. For example, the user might ask, "What error message is shown in the dialog on my screen?"

How it works: This function is essentially a wrapper that ties together screenshot capture and AI vision analysis. Internally, it calls `Alimage.AnalyzeScreenshotAsync(apiKey, query)]` .

The `Alimage`` module presumably does the following:

- If a screenshot hasn't been taken recently, it triggers `TakePrintScreenOrScreenShot` to capture the primary monitor. In Shelly's code, `TakePrintScreenOrScreenShot` is actually defined as a PowerShell script (see below), but `Alimage` might have a more direct method using a library (the project references `Dapplo.Windows` which can also capture screens). One way or another, a screenshot image is obtained (likely saved to a temporary file or kept in memory).
- `AnalyzeScreenshotAsync` then calls OpenAI's API with a special prompt, attaching the screenshot image (encoded in base64) as part of the message. Since GPT-4 (the version with vision) can accept image inputs, it will process the image and the query. Essentially, Shelly asks GPT-4: *"Here's an image (screenshot). Now answer the user's query about this image: '{query}'"*
- The AI returns an answer based on the image content.

Because this uses GPT-4's multi-modal capability, it requires the API key to have vision access (in 2025, GPT-4 with vision is typically available). The function then returns the answer string to Shelly, which will be displayed to the user.

Use case: The user can simply ask, "Shelly, what's on my screen?" or more pointedly, "Shelly, read the error on the screen and tell me what it means." Shelly (via this function) will take a screenshot and GPT-4 will output something like, "The error dialog says 'File not found: config.ini'. It means the application couldn't locate the configuration file." This is incredibly useful for assisting with on-screen information without the user having to type it out.

(If the Copilot window is open, GPT might use that content instead via ReadCopilotConversation, but that's a different function. CheckMyScreenAndAnswer is purely about the graphical screen content.)

ReadFileAndAnswer(filePaths, query)

Purpose: Reads the content of one or multiple files and answers a question about that content. Essentially, it lets the user ask something like "Summarize the following file for me" or "Search these files for XYZ and explain the context."

How it works: This function takes a semicolon-separated list of file paths and a natural language query. It will:

- Open and read each file's content. It uses a helper `FileHandler.GetFileContent(path)` which likely handles text encoding, etc. It concatenates all file contents into one big string, but with clear delimiters marking where each file begin]. For example, it might produce:

```
' ==== File: C:\Folder\file1.txt ====  
(contents of file1)  
' ==== File: D:\Docs\file2.log ====  
(contents of file2)
```

Including the file name helps GPT understand context and reference it in the answer.

- It then splits the combined text into manageable chunks to avoid token limit]. It does this by lines: it accumulates lines until a max char count ($\sim \text{Globals.maxInputTokensPerChunk} * 4$ characters) is reached, then starts a new chunk. This ensures no chunk is too large for GPT-4. Often `maxInputTokensPerChunk` is something like 8192 or similar (the developer can override it), meaning each chunk might be ~8000 tokens or less.
- Next, the function builds a **single conversation** for GPT with these chunks. This is a clever approach: Instead of calling the AI for each chunk separately (which could lose cross-chunk context), Shelly sends *all chunks in one conversation*. It does so by constructing multiple user messages: the first user message contains "<CODE CHUNK 1/N>\n{content_of_chunk1}", the second user message "<CONTINUATION 2/N>\n{content_of_chunk2}", etc., and finally one message that asks the actual question about the conten]. Additionally, a system message is added if needed (the code example shows a system message about being a "VB.NET syntax checker" in the snippe], which might be from a specific use case; generally, it could be tailored to the query).
- Then it calls `CallGPTCore` with this assembled message lis]. GPT-4 will receive the conversation consisting of potentially many messages (the concatenated file content split across messages) and then the query. Thanks to its large context, GPT-4 can consider all of it and produce a single answer.
- The answer (which might be a summary, or search result, or explanation, depending on the query) is then returned by the function and shown to the user.

This approach is effectively performing a **multi-part prompt** to GPT, feeding it large file contents in chunks. By labeling them <CODE CHUNK i/n>, the prompt helps GPT understand they are sequential parts of a whole. The function itself doesn't do the answering – GPT does – but the function orchestrates feeding the data to GPT properly.

Use case: If a user says, "Look at *report1.txt* and *report2.txt* and tell me if either mentions John Doe," Shelly's planner would choose `ReadFileAndAnswer(["report1.txt;report2.txt"], "Do they mention John Doe?")`. The function will read both, pass them to GPT with the question. GPT might answer: "Yes, John Doe is mentioned in *report2.txt* in the context of ..., but not in *report1.txt*." This saves the user from manually opening and searching each file.

GenerateLargeFileWithTextOrCode(topic, outputPath, totalChunks)

Purpose: Creates a large text file (or code file) by generating it piece by piece with the AI. This is used when the content requested is too big to generate in one go. For example, “Generate a 1000-line CSV of sample data” or “Create a long example config file.”

How it works: This function will create an empty file at outputPath and then append to it in a loop:

- It ensures the directory exists and initializes an empty file] .
- It clears any cached content for that file in Shelly’s memory (Shelly caches file texts in Globals.FileContents sometimes] .
- It determines a “tail context” length: basically how many characters from the end of the file to include as context for each next chunk generation. It uses Globals.maxInputTokensPerChunk (which might be something like 2048 tokens) and multiplies by 4 to get char coun] . If the file is large, it will only feed the last segment of it to GPT when asking for the next part, to maintain continuity.
- Then for each i from 1 to totalChunks:
 - It reads the current content of the file (especially the tail part – e.g., last 8000 characters] .
 - It builds a prompt asking GPT to continue the file. The prompt might look like:

```
File: example.txt
Topic: {topic}
Chunks: i of N
[If there is existing content:] Already generated (last
context):
<last few lines of the existing file content>
```

Now generate ONLY chunk #i, continuing immediately after the above content. Do NOT repeat any existing lines.

If i == N (the final chunk), then finish and close the file.

Respond with only the fully updated content for this chunk (no extra explanations or fences).

This prompt is placed in a user message, possibly with a system message framing the task.

- It calls `CallGPTCore` with that prompt, temperature 0 (for deterministic output] . GPT returns a block of text which represents the next segment of the file.
- The function then cleans that response by stripping any `` that might have snuck in or unnecessary whitespace] .
- It appends this chunk text to the file (using `AppendContentToFileUniversal`] . This actually writes to disk.
- It logs that chunk's completion and moves to the next iteration.
- After generating all chunks, it reloads the full file content into memory and returns the path or a success message] . (In the code, it returns the final content as a string as well, but typically we just need the file written.)

This function essentially coaxes GPT to produce a very large output by doing it in parts, each part knowing what came before. By including the tail of the file in each successive prompt, GPT maintains continuity (for example, not to duplicate lines, or to ensure the code compiles across chunks, etc.). It also explicitly instructs GPT not to repeat content and to finalize properly on the last chunk.

Use case: Suppose a user asks, "Generate a dummy log file with 10,000 lines of varied log entries." The planner might plan something like `GenerateLargeFileWithTextOrCode("dummy log entries", "C:\\ShellyDefault\\biglog.txt", totalChunks:=5)`.

Shelly will then call GPT five times, each time getting ~2000 lines (for example) of log entries, appending to `biglog.txt`. The end result is a large file on disk. Shelly can then say "File generated successfully at: `C:\\ShellyDefault\\biglog.txt`" as confirmation. The user can open that file to see the content. Without chunking, GPT might not be able to generate such a large body of text in one response due to token limits; this function circumvents that.

`UpdateFileByChunks(filePath, updateInstruction, chunkTokenOverride)`

Purpose: Edits or refactors a large file in chunks based on a single instruction, while preserving overall context. In simpler terms, it allows Shelly to **take a big file and make consistent modifications throughout it** using AI (for example, "remove all comments from this code file" or "change the formatting of this JSON file").

How it works: This is one of the more complex functions:

- It loads the entire file content into memory (caching it in `Globals.FileContents`).
- It estimates total tokens of the file and decides a safe chunk size. Typically, it tries to use ~65% of the context window for input (to leave room for output). For example, if using a 32k model, 65% ~ 20k tokens for input. It converts that to a char count (token * 4).
- It then splits the file into overlapping chunks by line. It includes a small overlap (e.g., 1 line overlap) between consecutive chunks so that changes at chunk boundaries stay consistent. Essentially, it will produce chunks of text from the file, each chunk possibly a few thousand tokens long.
- It then iterates through each chunk and processes it with GPT:
 - For the first chunk, it sends a prompt containing the **instruction** and the chunk text. Specifically, it might say:
“Instruction: <updateInstruction>\n---\nChunk 1/N:\n\n<chunk text>\n\n---\nRespond with only the fully updated content for this chunk; do not include fences.” (Notice they use “” as a fence delimiter in the prompt to clearly mark the chunk content.)
 - GPT returns some updated text (supposedly the chunk with the instruction applied, for example if instruction was “remove comments”, GPT returns the chunk without comments).
 - Shelly appends this result to an output string builder.
 - If the GPT output was cut off (there’s logic to detect if the response likely hit max tokens), the function will send a follow-up user message “Continue updating the rest of this chunk, appending only new content without repeating prior content.” and call GPT again to get the continuation. It keeps doing that until the chunk is fully processed (this handles cases where even one chunk’s output didn’t fit in one response).
 - It then moves to the next chunk. For chunk 2 and onwards, it likely has context from previous chunk if needed (though in implementation, they might treat each chunk separately with the same initial instruction – the overlap ensures continuity).

- After processing all chunks, the function has a list of updated chunks. It then **reassembles** them into one final content string. The overlapping lines mean there will be duplication at chunk boundaries; Shelly handles this by skipping the first line of each subsequent chunk's output (because it's the overlap from the previous chunk).
- It writes the final assembled text back to the file (overwriting it) and updates the cache.
- It returns the updated content or a success note.

Use case: A concrete example: "Shelly, in the file LargeReport.md, replace every occurrence of 'ACME Corp' with 'Acme Corporation' and reformat all bullet points to numbered lists." This is a complex, repetitive edit. Shelly's planner would instruct a multi-step, but more elegantly, it might do it in one go with `UpdateFileByChunks("LargeReport.md", "Replace 'ACME Corp' with 'Acme Corporation' and convert bullet lists to numbered lists.")`.

The function will feed the file to GPT in segments with that single instruction, ensuring GPT's changes are applied consistently throughout the file. In the end, LargeReport.md is modified on disk with all occurrences replaced and lists renumbered. This is extremely powerful for bulk editing or code refactoring tasks.

GenerateImages(imagePrompt, numImages, style, folderPath)

Purpose: Uses OpenAI's image generation API (DALL·E or similar) to create one or multiple images based on a prompt, and saves them to disk. Shelly can then reference these images or provide their file paths to the user.

How it works: This function constructs an image request and handles the resulting files:

- It normalizes the target folder path (creating it if it doesn't exist).
- It creates a **filename slug** from the prompt – a lowercase, alphanumeric version of the prompt (truncated to 40 chars) to use in image filename. For example, prompt "Sunset over mountains" → slug "sunset-over-mountains".
- It appends a timestamp to ensure uniqueness (like 20250502-1315 for date-time) and an index, to form names like sunset-over-mountains-20250502-1315-1.jpg, ...-2.jpg, etc. It also checks for collisions (if file exists, add a suffix) just in case.

- It then prepares a list of ImageRequestData objects (each containing the prompt, desired size, and output name). Here, it likely concatenates the main prompt and the style text (if style is provided] . For instance, prompt “sunset over mountains”, style “in watercolor style” becomes full prompt “sunset over mountains in watercolor style”.
- It calls Alimage.CallImageGeneration(apiKey, reqs) to actually call the OpenAI Image API for generatio] . This returns a list of image URLs (hosted by OpenAI) if successful.
- The function then downloads each image from its URL to the specified folder with the filenames chose] . (It likely uses an async HTTP client to fetch the binary and save it.)
- It adds all saved file paths to a global list Globals.GeneratedImages and also to `Globals.TaskData("ImagePaths")` – possibly for later reference or for the UI to easily find them.
- Finally, it returns a string listing the saved images, e.g.:

Images saved:

C:\ShellyDefault\sunset-over-mountains-20250502-1315-1.jpg

C:\ShellyDefault\sunset-over-mountains-20250502-1315-2.jpg

If the API call fails or returns no images, it returns an error message.

Use case: The user might say, “Shelly, create 3 images of a cat riding a bicycle, in a sketch style.”

Shelly will use GenerateImages("a cat riding a bicycle", 3, "sketch style", "C:\\ShellyDefault").

After execution, Shelly will output something like:

“Images saved:

C:\ShellyDefault\cat-riding-a-bicycle-20250502-131501.jpg ...” for each image. The user can then open these files to view the generated images.

WebSearchAndRespondBasedOnPageContent(promptQuery, siteName, question)

Purpose: Performs a live web search and then extracts information from the first result page to answer a user's question. In essence, this function gives Shelly a mini web browser capability to fetch info not contained in the local context or training data.

How it works: Given a search query (promptQuery), an optional site/domain filter (siteName), and a question to answer from the page, the function will:

- Build a Google search URL. If siteName is provided and not "google", it uses the site:siteName promptQuery Google query to constrain results to that sit] . Otherwise, it searches the general web for the promptQuer] .
- It launches an invisible WebView2 browser instance (a headless browser embedded] . It navigates to the Google search URL and waits for the page to loa] .
- It then executes JavaScript in the context of the page to grab all hyperlink URLs on the search results pag] . It parses these URLs and picks the first result that is **not** a Google internal link (skips things like google.com/url? etc.] . This presumably gives the URL of the first real search result.
- It navigates the WebView2 to that first result URL and waits for it to loa] . Then it scrolls a bit to trigger any lazy content loading (the code scrolls down twice with delays] .
- It retrieves the entire visible text of the page by running document.body.innerText in the browser and getting the resul] . This yields all the textual content of the page (sans HTML tags).
- Now with pageText (which might be very large), the function tries to extract the answer to the user's question:
 - If the page text is within a size that GPT-4 can handle in one shot (they define constants, e.g., ContextWindow=128000 chars and MaxCompletion=16384 tokens] , it will do a single call. It sends a system message "You are an information extractor." and a user message: *"Extract all info to answer: '{question}' from the text below. List each item on its own line. Text:<<<{pageText}>>>"] . GPT then returns an answer, which the function prepends with the source URL and return] .

- If the page text is **too large**, the function falls back to a chunked approach . It splits the page text into slices (e.g., 8000 characters each) and for each chunk, asks GPT: *`“Extract info to answer: '{question}' from this text chunk. List items each on own line:<<<{chunk}>>>”` . It collects the partial answers from each chunk . Then it filters out irrelevant lines (like if GPT says “no info found in this chunk”) and merges all the distinct items .
- After that, it may send the merged list back to GPT to **format it nicely** (for example, to remove duplicates or put it in a specific output format) .
- It then returns the URL and the final formatted answer .
- Finally, the WebView2 browser is disposed to free memory.

Use case: If a user asks, “Shelly, find me the list of all HP laptop models and their prices on emag.ro,” Shelly will:

- Search Google for site:emag.ro HP laptops.
- Click the first result (likely a page on emag.ro listing HP laptops).
- Scrape that page’s text.
- Ask GPT to extract “all HP laptops and prices” from the text.
- Possibly chunk it if needed, then output a list like:

```
URL: https://www.amazon.com/... (the page it found)
HP Model 15 – $500
HP Spectre x360 – $1200
...
```

This single function encapsulates a multi-step web browsing and reading task that GPT alone could not do (because GPT’s training data might be outdated and it can’t browse by itself in real-time). It’s a great example of extending Shelly’s knowledge by giving it controlled internet access.

TakePrintScreenOrScreenShot(OutputPath)

Purpose: Captures a screenshot of the primary monitor and saves it to a file. This is used by Shelly when an action requires an image of the screen (like CheckMyScreenAndAnswer or potentially a user explicitly asking for a screenshot).

How it works: This is actually implemented as a small PowerShell script rather than a VB function:

powershell

```
Function TakePrintScreenOrScreenShot {  
    param([string]$OutputPath) if (-not (Test-Path $OutputPath)) {  
        New-Item -ItemType Directory -Path $OutputPath | Out-Null }  
    $fileName = 'Screenshot_' + (Get-Date -Format 'yyyyMMdd_HH:mm:ss')  
    + '.png' $filePath = Join-Path -Path $OutputPath -ChildPath  
    $fileName Add-Type -AssemblyName System.Windows.Forms Add-Type -  
    AssemblyName System.Drawing $bounds =  
    [System.Windows.Forms.Screen]::PrimaryScreen.Bounds $bitmap =  
    New-Object System.Drawing.Bitmap $bounds.Width, $bounds.Height  
    $graphics = [System.Drawing.Graphics]::FromImage($bitmap)  
    $graphics.CopyFromScreen($bounds.Location,  
    [System.Drawing.Point]::Empty, $bounds.Size)  
    $bitmap.Save($filePath,  
    [System.Drawing.Imaging.ImageFormat]::Png) $graphics.Dispose()  
    $bitmap.Dispose() Write-Output 'Screenshot saved to: ' +  
    $filePath }
```

When Shelly's planner wants to take a screenshot, it can either directly invoke this via an `ExecutePowerShellScript` step (embedding this script), or via a function call if one wrapped it. The above script:

- Ensures the output directory exists,
- Generates a filename with a timestamp,
- Uses .NET System.Drawing to capture the screen, - Saves the PNG to the directory, - Outputs the file path. In Shelly's context, likely `Alimage.AnalyzeScreenshotAsync` uses this under the hood (or a similar

method) to get the screenshot file path, then loads that image for analysis.

For the user, this isn't directly called by name typically (there's no direct user-facing command "TakePrintScreen"), but it's an important part of enabling screen analysis.

ImageAnswer(imagePaths, query)

Purpose: Answers a question about one or more image files provided by path.

This is similar to CheckMyScreenAndAnswer, but for arbitrary images, not necessarily the screenshot.

How it works: The function takes a string of one or multiple file paths (comma-separated). It first validates that each path exists on disk].

If any file is missing, it returns an error for that invalid path. Assuming all images exist, it calls ``Alimage.AnalyzeImagesContentAsync(apiKey, imagePaths, query)]``.

This likely packages each image (converting to Base64) and sends them along with the question to the OpenAI GPT-4 model with vision. The model then sees the images and the query and produces an answer. For multiple images, the prompt might enumerate them ("Image1 is ..., Image2 is ..., now answer the query using both."). The function returns whatever answer GPT gives.

Use case: If the user has images on disk, say two charts, and asks "Shelly, do these two images show the same trend?", Shelly could use ``ImageAnswer("C:\imgs\chart1.png, C:\imgs\chart2.png", "Are the trends similar?")``.

GPT-4 would analyze both images and answer the question (e.g., "Yes, both charts show an upward trend over time."). This function turns Shelly into a mini-image analyst for local images.

ReadCopilotConversation(query)

Purpose: Integrates content from Shelly's ****Copilot chat window**** to answer a question. Shelly has a secondary interface ("Copilot" form) where perhaps the user or an AI has been generating some conversational or code context (for example, maybe the user loaded a large text or had a prior conversation with GPT in a different mode).

This function lets Shelly pull that conversation and summarize or answer questions about it.

How it works: It first checks that the Copilot form is open and visible. If not, it returns an error asking the user to open it (since there's no conversation to read otherwise). - It then grabs the

text from ``Copilot.Instance.AiOnePrompt.Text`` (which is likely a `RichTextBox` containing the chat history in the Copilot window]. If it's empty, returns an error.

- It splits this conversation text into chunks of ~800 word].
- It then iteratively sends each chunk to GPT with the user's query, presumably to summarize or extract relevant info chunk by chunk: - It likely calls a helper (maybe ``FileProcessChunk(chunk, query)``] that returns a tuple of (answer, reasoning) per chunk. In the code snippet, they then take `result.Item1` (the answer portion) and label it as "[Chunk i]: answer" and collect thes].
- After processing all chunks, it compiles an ``overallSummary`` which is basically a set of partial answers from each chunk].
- It then builds a final prompt: "Based on the following Copilot conversation summaries: [the compiled chunk summaries] answer the following question concisely, ensuring you include all relevant details: {query}]].
- It sends that to GPT via ``CallGPTCore`` and gets a final integrated answer]
- Returns that answer.

So essentially, it's a **two-pass approach**: first summarize each chunk of the conversation relevant to the query, then combine those to answer comprehensively. It's similar to how `ReadFileAndAnswer` works for text, but specifically tailored to the content in the Copilot chat UI.

Use case: Imagine the user had a lengthy brainstorming chat in the Copilot window (maybe outside of the structured Shelly Q&A flow), and now wants Shelly to analyze it: "Given our discussion in the Copilot panel, what were the main action items?" Shelly will use ``ReadCopilotConversation("Provide a list of action items discussed")``. It will fetch the whole chat text from the Copilot UI, break it down, summarize, and produce a list of action items. This is useful to bridge that separate UI's content into Shelly's response.

SearchForTextInsideFiles(paths, searchWord)

Purpose: Searches through one or multiple files and folders for a given text substring (case-insensitive), and returns the list of files that contain it. It's basically a file grep utility for Shelly.

How it works: This function wraps a PowerShell script that performs the search: - It builds a PowerShell script (as a string) that: - Splits the ``paths`` input (which can contain multiple file or directory paths separated by ``;``). - Defines a helper ``Get-Files($p)`` that, if `$p` is a directory, lists all files inside recursively, or if `$p` is a single file, just returns that. This handles both file and folder input] . - Defines a helper ``Read-FileContent($filePath)`` to read file content safely for different file types (text, docx, xlsx, etc.].

For text, it uses `Get-Content`. For .docx, it uses the Word COM object to extract text. For .xlsx, it uses Excel COM to read cells. This is quite advanced:

- it attempts to get text out of Office files too, not just plain text. - Iterates through each file found and checks if the content (as string) matches the search word (using `-imatch` for case-insensitive regex match).

- Collects matching file paths in `\$foundFiles` and outputs them at the end.

- The VB function then executes this PowerShell script by calling
`Shelly.Instance.ExecutePowerShellScriptAsync(script)` and waiting for the result (synchronously, since it's within an Async function).

- If execution succeeds, it takes the output (which will be the list of file paths, one per line).

If the output is empty, it returns "No files matched." If there are matches, it returns "Matched files:\n<file1>\n<file2>...".

- If the PowerShell script itself failed (perhaps an invalid path, etc.), it returns an error with the error text.

Use case: The user could ask, "Shelly, search in `D:\Projects` and `C:\temp\notes.txt` for the phrase 'hello world'." Shelly would call
`SearchForTextInsideFiles("D:\\Projects;C:\\temp\\notes.txt", "hello world")`. The function returns:

```
Matched files:
D:\Projects\demo\readme.md
C:\temp\notes.txt
```

Matched files:

D:\Projects\demo\readme.md

C:\temp\notes.txt

- indicating those files contain "hello world". Shelly would present that as the answer. The user can then follow up asking Shelly to open or summarize those files if needed. This function basically gives Shelly a way to locate information across the filesystem in a safe read-only manner.

ReadWebPageAndRespondBasedOnPageContent(url, query)

Purpose: Similar to the Google search function, but directly loads a specified URL and then answers a query based on that page's content. This is useful if the user already has a URL in mind and wants Shelly to extract or summarize information from it.

How it works:

- It first validates that the `url` is well-formed]. If not, returns an error.
- It creates a headless WebView2 browser (just like the search function did], navigates to the given URL, and waits for it to fully load (with some scrolling and delays to ensure dynamic content loads].
- It then grabs the `document.body.innerText` as before to get all visible text on the page].
- If no text was retrieved (page might be empty or an error), it returns an error].
- If text is retrieved, it proceeds similar to the earlier function:
- If the text is within a single-call size, it asks GPT in one go: `*"Extract information to answer: '{query}' from the text below."*` (with system role as extractor]. GPT's answer is returned prefixed with the source UR].
- If the text is too large, it does chunking:
- Splits `pageText` into chunks (they use a slightly smaller chunk size here, 800 tokens, to be safe] .
- For each chunk, ask GPT: `*"Extract information to answer: '{query}' from this text chunk."*` , collect results.
- Filter and deduplicate the result] . If no info found, return error stating no relevant item] .
- Optionally, format the consolidated list with another GPT call to make it clea] .
- Returns the final answer prefixed by the UR] .
- Disposes the web browser.

Use case: The user says, "Shelly, open `https://www.example.com/page.html` and list all product names and prices mentioned." Shelly will not do a Google search; it will directly load that URL, scrape it, and extract product names and prices.

The result might be:

```
URL: https://www.example.com/page.html
Product A - $10
Product B - $20
...
```

This function allows targeted retrieval when the user already knows the page to go to. It's more direct than the Google search tool and avoids extraneous steps.

GenerateBatchAndPs1File(outputFolder, userQuery)

Purpose: Creates two files: a Windows batch script (`.bat`) and a PowerShell script (`.ps1`) in the specified folder, based on a single task description. The batch file when run will launch the PowerShell script. This is useful for packaging an automation so it can be run standalone outside Shelly (or shared with someone who can just double-click a .bat).

How it works:

- It ensures the output directory exists (creates it if not] .
- It builds a multi-part ****prompt for GPT-4**** that essentially instructs it to produce two files:
- It sets the system role: "You are an expert shell-and-powershell scripter."
- The user content of the prompt includes detailed instructions (written by Shelly's developer) on how to format the response. It says:
 - "Create two files in the target folder:
 - 1) A .ps1 script that performs the following task: {userQuery}
 - Wrap the entire logic in Try/Catch.
 - In the Catch block, write the error to host.
 - At the very end (in both try and catch), display a prompt: `Write-Host 'Press any key to exit...` then `[void][System.Console]::ReadKey()`.

2) A .bat file that calls the .ps1 with: ``powershell -ExecutionPolicy Bypass -File script.ps1`` .

- It then explicitly tells GPT how to respond: `**"Respond with exactly two fenced code blocks: one with ```bat ...``` for the .bat content, and one with ```powershell ...``` for the .ps1 content."*`] .
- Essentially, Shelly is asking GPT to output the contents of both files in one response, clearly separated.
- Shelly calls ``CallGPTCore`` with this prompt (temperature 0 for accuracy)] .
- It gets ``aiResponse`` which should contain something like:

```
```bat
@echo off

powershell -ExecutionPolicy Bypass -File "script.ps1"

powershell
try { # ... do the user requested task ... } catch { Write-Host $Exception.Message } finally {
Write-Host 'Press any key to exit...' [void][System.Console]::ReadKey() }
```

- Shelly then uses regex to extract the content inside the bat block and the powershell block separatel] . If it fails to find these blocks (meaning GPT didn't format as expected), it returns an erro] .
- If successful, it writes the extracted text to script.bat and script.ps1 files in the outputFolde] .
- Finally, it returns a message listing the paths of the two files create] .

**Use case:** The user asks, "Shelly, in D:\Deploy, create a script to check my last 20 Outlook emails and print the sender and subject." This is a complex automation. Shelly might not have a custom function for Outlook, so GPT would need to write a PowerShell that uses Outlook COM objects. The planner could directly attempt `ExecutePowerShellScript`, but maybe the user specifically said "create a script in folder" which hints at using this function.

Shelly uses `GenerateBatchAndPs1File("D:\\Deploy", "Check my last 20 Outlook emails and print sender address and subject")`. GPT (with its training knowledge of PowerShell and Outlook automation) will generate a PS1 that opens Outlook, retrieves mails, prints sender/subject, and the BAT to call it. Shelly saves those. The user can then go to `D:\\Deploy` and run the batch file to execute that task whenever they want, without needing Shelly running. This is an example of **Shelly generating artifacts (scripts) for later use**, effectively handing off automation in a portable form.

This function emphasizes using GPT for code generation with a very specific expected output format. It showcases how Shelly can leverage GPT to produce multi-file output, not just a single answer.

## Real-World Use Cases

With the architecture and functions described, we can illustrate a few realistic scenarios of Shelly in action:

- **Use Case 1: Bulk File Editing (Log Cleanup)** – Imagine you have a large log file and you want to remove all lines that contain a certain debug phrase and then save the cleaned file. You can instruct Shelly in natural language: “Open `C:\\Logs\\app.log` and remove any lines that say `DEBUG`, then save it.” Shelly’s planner would likely choose the `UpdateFileByChunks` tool, because it’s a large file operation with a clear instruction. Under the hood, Shelly will split the log file and ask GPT-4 to remove lines containing “`DEBUG`” in each chunk, then stitch it back together. Within seconds, Shelly will output *“File updated successfully at: `C:\\Logs\\app.log`”*. The user’s log file is now cleaned of `DEBUG` lines. This shows Shelly handling a tedious editing task that would be error-prone to do manually. Importantly, the actual content of the log (which could be thousands of lines) was processed by the AI in a controlled way, chunk by chunk, without the user having to open or scroll through it.
- **Use Case 2: Automated Script Generation for Task Scheduling** – Suppose a user needs to run a complex series of commands every day and wants to create a reusable script. The user can ask: “Shelly, generate a script in `D:\\Tasks\\Backup` to archive all `.txt` files in `D:\\Data` older than 30 days into a zip, then delete the originals. Make it so I can run it by double-click.” This is a multi-step operation: find files by date, compress them, remove originals. Shelly will use the `GenerateBatchAndPs1File` function. GPT-4 will produce a PowerShell script (maybe using `Compress-Archive` and file date logic in `Try/Catch`) and a batch file to invoke it. Shelly saves these in `D:\\Tasks\\Backup\\script.ps1` and `script.bat` and tells the user. Now the user has a ready-

made automation script they can run anytime, even if Shelly isn't open. This showcases Shelly acting as a coding assistant that not only writes code but also packages it for use. For a developer collaborator, it's impressive to see GPT-4 generate two coordinated files with proper error handling as per Shelly's prompt constraints.

- **Use Case 3: Screen Information and Troubleshooting** – A user encounters an error dialog while installing software and doesn't understand it. They can simply ask Shelly: "What does the error on my screen mean?" Shelly will trigger `CheckMyScreenAndAnswer`. It takes a screenshot of the desktop, and sends it along with the query to GPT-4. GPT might see the error message "Error 1722: There is a problem with this Windows Installer package..." on the image. Shelly then replies to the user with something like: *"The error on your screen is Error 1722, which usually means there's a problem with the installer. This often indicates a custom action in the MSI failed. In practice, it means the installation didn't complete. You might need to run the installer as admin or check if another installer is conflicting."* The user gets an immediate explanation without typing the error manually into Google. This demonstrates Shelly's capability to bridge the gap between visual data and solutions, effectively acting like a support technician. It's also a good example of privacy in action: Shelly only took that screenshot because the user asked, and the analysis of the screenshot was done on the fly, not stored or sent elsewhere beyond the AI for that moment.

These examples barely scratch the surface: Shelly can also do things like **web research** ("Shelly, find the top 5 StackOverflow answers about error X and summarize them" – it would use the web search function), **code analysis** ("Check these two code files for differences" – using `ReadFileAndAnswer` or `SearchForText`), **image generation** ("Create an image of a sunset over the ocean" – Shelly will call `GenerateImages` and save a file for the user to view). All of this is done through a unified conversational interface.

**Shelly's audience** (developers and tech enthusiasts) can leverage these capabilities to speed up workflows: Instead of manually writing scripts or performing repetitive searches, they can delegate to Shelly. And if Shelly doesn't have a built-in tool for something, a developer can add it, thanks to the extensible CustomFunctions design. This makes Shelly not just a single-use app, but a platform for AI-assisted automation on Windows.

# Shelly App – Technical Documentation

## Introduction

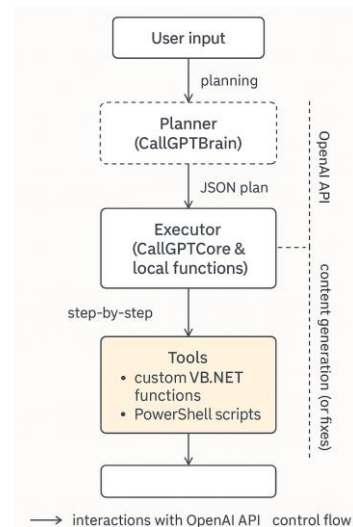
Shelly is a hybrid AI automation assistant for Windows, combining the intelligence of GPT-4 with local execution capabilities. It enables users (especially developers, power users, and early adopters) to automate complex tasks through natural language. The mission of Shelly is to **plan and perform multi-step tasks** on a Windows PC – such as running scripts, searching the web, editing files, or analyzing on-screen content – all by interpreting a user’s request in plain English. Shelly is **not targeted at enterprises** or non-technical end-users; instead, it’s designed for individual developers and collaborators who seek a powerful, extensible assistant they can tweak and trust in a local environment.

Shelly is an open-source project (released under a Creative Commons BY-NC license) and welcomes contributions. This documentation provides a comprehensive technical overview of Shelly’s architecture and components. It is structured to guide developers and advanced users through Shelly’s design, including its GPT-driven planning brain, execution core, custom function library, and user interface. We’ll discuss how Shelly plans tasks, executes them securely, handles errors, and how new capabilities (custom “tools”) can be added. Real-world examples are provided to illustrate Shelly’s use cases, like batch file editing, script generation, and screen content analysis.

## System Overview

Shelly employs a **hybrid GPT-agent architecture**. At a high level, Shelly works as follows: a user enters a command or question in natural language; Shelly uses GPT-4 (via OpenAI’s API) to **plan** how to fulfill the request; and then Shelly’s local **execution engine** carries out the plan step by step on the user’s machine. This approach lets Shelly handle multi-step procedures automatically (“auto-mode”) – for example, searching for information and then creating a file with the results – without further user intervention. If a request doesn’t require any external actions, Shelly can answer directly with GPT. Otherwise, Shelly will combine **AI-generated** instructions with **local actions** (PowerShell scripts or custom VB.NET functions) to complete the job.

*Figure: High-level architecture of Shelly's GPT-powered agent. The user's request goes to a Planner module (CallGPTBrain) which uses GPT to produce a JSON plan. That plan is then executed step-by-step by the Executor (CallGPTCore & local functions). The executor may call custom VB.NET functions (tools) or run PowerShell scripts. Dashed lines indicate interactions with the OpenAI API (for planning, content generation, or fixes). Solid lines indicate control flow on the local machine.*



Under the hood, Shelly maintains a conversation with the GPT model to accumulate context about what it's doing. It uses two main AI calls: one to a **Planner** (which formulates a series of actions in JSON format), and another to a **Core Executor** (which handles general queries or content generation using GPT). The “brain” of Shelly (the Planner) decides **which tools or scripts to use** to satisfy the request. The “hands” of Shelly (the Executor) then invoke those tools on the local system. Shelly's architecture thus balances cloud AI intelligence with local execution power. Crucially, the AI is **not given free rein on the system**; it can only perform actions through a predefined set of tools (custom functions or scripted commands) that developers have explicitly implemented. This ensures that Shelly's capabilities are extensible yet constrained to intended operations.

In practice, when the user submits a prompt, Shelly will either:

- **Answer directly** (for purely informational queries) using the GPT model (Shelly will return a simple answer via a “FreeResponse” tool), - or -
- **Generate a plan** of one or more steps if the request requires actions. The plan might include custom function calls (like reading a file or taking a screenshot) and/or PowerShell commands. Shelly will then execute each step in order automatically.

Shelly's system is designed for **multi-step task automation**. It loops through planning and execution until the user's request is fully satisfied. After completing a series of steps, Shelly can even summarize what it did for the user. For example, if the user asked Shelly to perform a series of file operations and web searches, once done Shelly can present a friendly summary of all actions carried out. This makes the interaction feel natural and informative.

## Planner: *CallGPTBrain* (Task Planning with GPT-4)

The planning phase is handled by the *CallGPTBrain* function (located in **AIBrainiac.vb**). This function acts as Shelly's "brain," formulating a step-by-step game plan to achieve the user's request. It leverages OpenAI's Chat Completion API (specifically GPT-4, often with the 32k token context or higher) possibly via the OpenAI *Beta Assistants* mechanism. The planner uses a system prompt that embodies Shelly's core instructions and toolset, and it sends the user's query in that context.

**How *CallGPTBrain* works:** When invoked, it first retrieves the assistant's preset profile (if using the Assistants API) including any built-in instructions and the recommended model. It then constructs a message list for the chat completion call:

- A **system message** with Shelly's fixed role and guidelines. For example, Shelly's system prompt tells GPT: *"You are Shelly, a powerful Windows assistant. Decide which actions to take – via PowerShell or custom VB.NET functions – to fulfill the user's request fully and sequentially. Always use custom functions when available, otherwise use PowerShell in fenced code blocks. Never reveal your internal process."* This ensures the AI knows it should output actions, not just an answer.
- The assistant's own additional instructions (if any were fetched via the Assistants API).
- Recent **conversation history** (the last N messages, to maintain context). Shelly caps this to a certain number of messages (by default 20) to fit within model limits.
- A **user message** describing the planning task. Shelly formulates a special user prompt that includes: the original user request, a list of any tools already executed so far (if in a loop), and explicit instructions on how the AI should respond. Notably, Shelly instructs the AI to output either a direct answer or a JSON array of steps. For example, the planning prompt tells GPT:

```
ORIGINAL REQUEST: <user's request here> TOOLS ALREADY FINISHED
(with full args): <none or list of completed steps>
```

- If you can answer entirely in natural language, return a single step using tool="FreeResponse" with **args**.text set to the answer.
- If the answer requires running or evaluating PowerShell, use tool="ExecutePowerShellScript" and place the script inside **args**.script (wrapped in a fenced block).
- Otherwise list ALL remaining tool steps in order. Return only the JSON array (no markdown fences, no commentary).



*(The above is a simplified representation of Shelly's planner instructions.)*

When CallGPTBrain sends this prompt to the OpenAI API, GPT-4 will ideally respond with a **JSON-formatted plan**. For example, GPT-4 might return:

```
[
 {
 "tool": "ReadFileAndAnswer", "args": {
 "filePaths": "C:\\\\Logs\\\\app.log",
 "query": "summarize errors"
 },
 "tool": "GenerateBatchAndPs1File", "args": {
 "outputFolder": "D:\\\\Demo",
 "userQuery": "Check disk space and list largest files"
 }
 }
]
```

This plan (as a JSON array) enumerates the tools to execute in sequence, with the necessary arguments for each. If the request was simple enough, the plan may contain just one step with the FreeResponse tool (meaning GPT-4 handled it conversationally without needing any real action).

**Processing the plan output:** Shelly's code examines the raw reply from GPT. It strips away any markdown formatting to isolate the JSON. Then it attempts to deserialize the JSON string into a list of PlanStep objects. There are a few possibilities at this stage:

- **Valid Plan:** If parsing succeeds and yields a non-empty list of steps, Shelly proceeds with those steps.
- **Single FreeResponse:** If the plan is a one-step plan and that step is a FreeResponse, Shelly interprets it as an answer. In this case, Shelly will simply display the args.text from that step as the assistant's answer to the user (without running further tasks).
- **Multi-step Plan:** If the plan contains multiple steps, Shelly enters multi-task execution mode (described in the next section). It sets a flag lastRunMultiTask = True to remember that multiple actions were taken.
- **No JSON / Unexpected Output:** If GPT's response can't be parsed as the expected JSON (for example, the model might have returned a narrative answer or incorrectly formatted plan), Shelly falls back to a more forgiving approach. In this fallback, Shelly passes the raw AI response to a multi-step handler that will **parse any embedded code blocks or**

**function calls from plain text** (treating the response as if it were a conversation containing instructions). This ensures that even if the planner didn't follow the format strictly, Shelly can still attempt to execute any detectable actions from the reply.

Shelly logs the raw plan for debugging purposes, so developers can inspect what the AI proposed. Any error in parsing is caught, and if no executable steps are found, Shelly will ultimately just show an error or do nothing.

**Role of the Planner (CallGPTBrain):** In summary, the planner's job is to translate an open-ended user request into a structured game plan. It knows about all the **available tools** (custom functions and the special ExecutePowerShellScript and FreeResponse actions) and instructs GPT-4 to use them appropriately. The planner is stateless except for the chat history it carries – importantly, it does not itself execute anything. It simply returns a plan. By centralizing decision-making in GPT-4 (which has knowledge of the task and tool descriptions), Shelly can easily be extended with new tools: update the planner's prompt to include the new function, and GPT-4 can start using it in plans. (We will discuss extensibility in a later section.) The use of OpenAI's Assistants API means the exact model (and possibly some predefined high-level instructions) can be managed via an "assistant profile"; for example, Shelly's assistant profile could specify GPT-4-32k to allow very large plans or context, and include a list of tool definitions for GPT. The CallGPTBrain function makes sure to insert Shelly's core rules at runtime so that even if the assistant profile changes, Shelly's fundamental policies (use tools, don't reveal system prompt, etc.) are enforced on every plan request.

## Executor: *CallGPTCore* and Task Execution Flow

Once a plan is obtained, Shelly's **Executor** takes over to carry out each step. The executor involves a few components working together:

- The CallGPTCore function (in **Alcall.vb**) is a general-purpose routine to call the OpenAI API for **non-planning purposes** (e.g., getting the content of a text request, or asking GPT to fix a script error). It's essentially Shelly's interface to GPT-4 for everything aside from the main planning step.
- The **ExecutorAgent** (part of Shelly's logic in **Shelly.vb** and possibly **ExecutorAgent.vb**) which iterates through the plan steps and invokes the appropriate local action for each.
- The **CustomFunctions** module (in **CustomFunctions.vb**), which implements the set of custom VB.NET functions (tools) that the plan might reference.

- A **PowerShell runner**, which executes any raw PowerShell scripts that the plan includes, with error handling and retries.

**CallGPTCore (Alcall.CallGPTCore):** This function is a wrapper around the OpenAI chat completion API with some important features:

- It will automatically inject Shelly's system prompt (the same one described earlier) into the message list if no system message is present. This is a safeguard to ensure even ad-hoc GPT calls adhere to Shelly's rules (for instance, if a custom function directly queries GPT, Shelly still reminds the model about being a Windows assistant, etc.).
- It implements **dynamic token budgeting**. Shelly defines a very large context window (up to 128k tokens input, 16k output in code, anticipating future model capabilities). It calculates how many tokens the input messages might consume (estimating ~4 characters per token) and then sets `max_tokens` for the completion accordingly. For example, if the messages already use 10k tokens, it might allow up to 6k for the answer (or less if hitting model limits). This prevents GPT from generating responses that exceed context or from refusing due to length.
- It sets OpenAI API parameters like temperature, `top_p`, etc. (Shelly often uses a moderate temperature ~0.7 for creative tasks, or 0.0 for deterministic tasks like code generation or extraction).
- It handles **API errors and retries**. If the HTTP request to OpenAI fails or times out, CallGPTCore will catch exceptions and try up to 3 times with a short backoff. Certain errors like invalid API key or model not found are caught and returned as error messages (e.g., "[ERROR] Model not found") without retry.
- On a successful API call, it extracts the assistant's response text and returns it as a trimmed string. It also stores the actual model used in a global variable for reference (since OpenAI may roll over to a compatible model, this is logged for transparency).

In essence, CallGPTCore is the reliable messenger: it sends a prompt and gets a response from GPT, handling all the low-level details (headers, JSON payload, error cases). The Executor will use this whenever it needs GPT's help during execution (for example, to answer a sub-question or to correct a PowerShell script).

**Executing the Plan:** After planning, Shelly's main loop (in **Shelly.HandleUserRequestAsync**) receives the list of PlanSteps. It then calls `ExecutorAgent.ExecutePlanAsync(plan)`, which runs each step in order. Each step has a Tool name and an Args dictionary. The execution flow does the following for each step:

1. **Identify the tool type:** Shelly checks the Tool field to decide how to execute it:
  - If the tool corresponds to a custom VB.NET function (one of Shelly's built-in functions in CustomFunctions.vb), Shelly will call that function.
  - If the tool is "ExecutePowerShellScript", Shelly will extract the script from Args and run it in PowerShell.
  - If the tool is "FreeResponse", it means this step is just a natural language answer. Shelly will output the text in args.text to the user interface.
  - (If the tool was "TextRequest" or similar, which could be used internally for GPT-only queries, Shelly would use CallGPTCore to fulfill it. In the JSON plan format, typically only FreeResponse is used for direct answers; other purely text operations might not appear as a tool but could arise from the fallback parser.)
2. **Ensure no duplicate executions:** Shelly uses a hash set executedCalls to record each function call or script it has executed in the current run. If the plan (or GPT output) happens to list the **same exact action twice**, Shelly will skip the duplicates to avoid redundant operations. For custom functions, it uses the function signature string (e.g., "ReadFileAndAnswer(\"C:\\file.txt\", \"Find X\")") as the key; for PowerShell scripts, it hashes the script content as the key (since scripts might be long). This deduplication protects against scenarios where the AI might inadvertently repeat an instruction.
3. **Execute the step:** Shelly executes according to type:
  - **Custom Function Call:** Shelly invokes ExecuteAppFunctionAsync(signature, ct) which looks up the function by name and calls it on a background thread (allowing async operations). All custom functions return a Task(Of String) – i.e., an asynchronous result string. Shelly awaits the result. If the function returns a non-empty string, Shelly will append that to the result output box for the user to see. Many functions return some message or result text; some functions might return an empty string to indicate they handled their own UI update or have no user-facing output. After a custom function executes, Shelly sets a flag skipNextPlainTextSegment = True. This flag is used because GPT sometimes includes a descriptive text after a function call; Shelly uses it to *suppress the next text segment* if it looks like a redundant summary the AI provided. (For example, the AI might have planned: GenerateImages(...) followed by a text segment like "Here are the images I generated." Shelly will execute GenerateImages, then skip printing the follow-up text "Here are the images..." since it's unnecessary – Shelly instead directly shows the list of image files generated.)

- **PowerShell Script Execution:** Shelly passes the script content to `ExecutePowerShellWithFixLoopAsync` (or similar internal method) which runs the script in a sandboxed PowerShell process. This process is created with user-level permissions (Shelly does not require admin rights unless the script itself needs to do admin tasks and the user runs Shelly as admin). Shelly typically uses PowerShell's `-NoProfile` mode to avoid user-specific profile scripts, ensuring a clean environment, and may use `-ExecutionPolicy Bypass` so that even if the system has a restrictive policy, Shelly can run the script. The output and errors from the script are captured. Shelly then enters a loop to handle errors: if the script failed (non-zero exit or any error text), Shelly will ask GPT to **debug/fix the script** on the fly. It does this by calling `Alcall.CallGPTCore` with a prompt like: "I tried running this PowerShell script but got an error: <error message>. Script: powershell ... Please return only a corrected script in a powershell block.". GPT will respond with a modified script (hopefully fixing the issue). Shelly then extracts the corrected script from the response and tries to run it again. Shelly will retry up to a few times (by default 3 attempts) for a failing script, each time feeding back the error to GPT and getting a fix. This **auto-debugging loop** is a key feature: it allows Shelly to handle cases where the initial command might not work due to environment specifics, missing modules, minor syntax issues, etc. If after the maximum attempts the script still fails, Shelly stops and logs a failure. On success, Shelly captures the script's output (if any) and displays it to the user. Successful or not, the script step is marked done and (as with functions) Shelly sets `skipNextPlainTextSegment = True` to avoid printing any AI-generated explanation that might have accompanied the script.
  - **FreeResponse (Natural Answer):** Shelly simply takes the provided text and appends it to the chat output as the assistant's answer. This typically happens when GPT determined no action was needed beyond explanation. Shelly ensures to log this and avoid duplicating any custom function output.
4. **Intermediate updates:** After each step execution, Shelly may insert the output as a message in the conversation history (so that the AI can reference what happened if the planning loop continues). The UI's status label is updated to reflect progress (e.g., "Executing planned tasks..."). A small delay or yield is introduced to keep the UI responsive. The user can cancel at any time using the Cancel button – Shelly checks a cancellation token `ct.IsCancellationRequested` at various points in the loop, ensuring it can abort long operations if the user requests.

5. **Post-plan summary:** If Shelly executed multiple steps for the request, at the very end it triggers a special summary generation. It formulates a new prompt for GPT: essentially “You (Shelly) just finished running multiple tasks based on the user’s prompt. Please give a short, friendly, dorky summary of what was done. If there were errors, mention them with a tone of trying your best.”. It uses a GPT call (CallGPTBrain again in this case) to get a single response summary, which it then displays to the user in the results box. This summary is purely for the user’s benefit and has no further actions. After this, Shelly resets its state for the next user prompt (clearing the multi-task flag, etc.).

While executing, Shelly logs debug information for each step (including outputs or any [ERROR] messages returned) for developers to inspect if needed. It also continuously trims the stored conversation history to avoid exceeding memory or token limits, using TrimConversationHistoryByTokens after significant additions.

**Multitasking and concurrency:** Shelly’s design currently executes tasks **sequentially** (one after the other in a single thread of execution, aside from awaiting async calls). It does not run multiple plan steps in parallel – this simplifies dependency management (later steps might depend on earlier ones, e.g., reading a file after it was created). “Multitasking” in Shelly refers to the ability to handle a **sequence of tasks automatically**, rather than simultaneous tasks. Each planning loop can handle multiple actions, and Shelly can even go through multiple plan-execute cycles if the assistant chooses to (though typically one cycle is enough per user query). During execution, the UI remains responsive and the user can cancel if needed, but they cannot issue a new query until the current one finishes (the input is disabled during run).

**Error handling and logging:** Shelly is robust in catching exceptions:

- If a custom function throws an exception, Shelly catches it and returns a safe error message like [ERROR] <exception message> to the result box instead of crashing.
- If something unforeseen happens in the main loop, Shelly catches it and shows a status “Error: <message>” in the UI status label.
- All errors and debug info are printed to Debug.WriteLine (which developers can see in Visual Studio’s Output window if running in debug) and some are stored in an internal log or the console form for later review.
- The planning step also handles error conditions, for example if the Assistants API fails or returns nothing, CallGPTBrain returns an error string which Shelly will display to the user (so that the user knows the planning failed).

After all tasks (or on error), Shelly re-enables the UI and resets the cancel button. The overall flow then waits for the next user input.

## AI Model Selection and Auto-Mode Logic

Shelly is built to work best with **GPT-4** (particularly models with large context windows). By default, it will use the model indicated by the Assistant profile or a configured global model. In practice, this means Shelly will typically call gpt-4-32k if available, to handle large inputs, but it can fall back to gpt-4 or even gpt-3.5-turbo if configured. The variable `Globals.AiModelSelection` (and similarly `Globals.UserApiKey`) is used to choose the model for `CallGPTCore` calls. This can be set by the user in a configuration UI or defaults. Shelly's code suggests an adaptive approach: it estimates token usage and if something is too large for the current model, it might warn or adjust. For instance, when performing web page reading or file reading, Shelly will attempt a single-call extraction if the text fits the model's context; if not, it automatically falls back to a chunk-by-chunk processing. This is a form of **auto-mode logic** where Shelly adapts to the model's limitations by splitting tasks.

Auto-mode in Shelly has two meanings:

1. **Automatic model/context adaptation:** Shelly will try to utilize the largest context model available to avoid unnecessary chopping of content. The `models.xlsx` file in the project likely lists model names and their token limits, guiding `AiModelSelection`. For example, if a user only has an API key for GPT-3.5, Shelly might detect that and use that model (but then certain complex tasks might not work as well or at all). On the other hand, if GPT-4 32k is available, Shelly will leverage it to handle huge texts. All of this is transparent to the user – the idea is the user just provides their API key and Shelly picks an optimal model.
2. **Automatic task execution (agent auto-mode):** Once the user issues a command and Shelly generates a plan, Shelly runs **fully automatically** through all steps. The user does not have to approve each action in a step-by-step manner (although they will see the actions/results as they happen). This design makes Shelly efficient for “one-shot” automation: the user describes an outcome, and Shelly figures out and executes the necessary procedure. Users can always break the loop by hitting “Cancel”, but otherwise Shelly assumes it has permission to carry out the plan it devised. This is in contrast to tools that might pause and ask “Do you want to execute this command?” for each step – Shelly's philosophy is to be truly autonomous in carrying out the user's wishes (since the user explicitly asked for it).

From a developer perspective, the model selection is handled in code by setting `Globals.AiModelSelection`. The Assistants API call (`RetrieveAssistant`) can dynamically provide a model name – for example, an assistant profile could specify `gpt-4` vs `gpt-4-32k`. Shelly uses whatever model is returned in the profile for the planning call. For the execution calls (`CallGPTCore`), Shelly uses its own `Globals.AiModelSelection` which presumably is set to the same model (or a default). There is also logic to record the actual model used from each API response, updating `Globals.LastUsedModel`, so that the UI or logs can show which model produced the output.

In summary, Shelly tries to **maximize the use of AI capabilities automatically**:

- It uses the most powerful model available for better reasoning and larger context.
- It breaks down tasks automatically and uses multi-turn interactions with the model without user intervention.
- It automatically handles when to just answer versus when to plan actions. The user does not have to explicitly toggle modes – Shelly’s planner will decide based on the request. (For instance, ask Shelly a general question and it will likely just answer; ask it to perform an operation and it will plan and execute.)

This “auto-mode” provides a seamless experience but also places responsibility on the planning logic to make correct decisions. The developer can fine-tune this by adjusting the planner’s prompt or adding new tools. If Shelly ever mis-classifies a request (e.g., tries to execute when it should just answer), that can be tweaked by modifying the instructions given to GPT in the planning step.

## PowerShell Execution and Validation Loop


One of Shelly’s most powerful capabilities is executing PowerShell scripts generated by GPT. This allows Shelly to do virtually anything on the system (within the user’s permission scope) – from file system operations to system configuration – based on natural language instructions. However, running arbitrary scripts poses both reliability and security challenges. Shelly’s design addresses these with a **validation loop** and some sandboxing measures:

- **Isolated Execution:** Shelly runs PowerShell scripts in a separate process (not in the main application’s process). It likely uses `System.Diagnostics.Process` to start a hidden PowerShell instance. The script content is passed via standard input or as a - Command argument. Shelly uses `UseShellExecute =`





false and RedirectStandardOutput/RedirectStandardError to capture the results. By using an external process, Shelly ensures that if a script hangs or crashes, it can still terminate it (using the currentPowerShellProcess handle it keeps) without bringing down the UI. Shelly also sets a **cancellation token** that, if triggered by the user pressing cancel, will attempt to kill the running PowerShell process gracefully.

- **No Persistent Changes without Intent:** Shelly launches PowerShell with “-NoProfile”, meaning it doesn’t execute the user’s PowerShell profile scripts. This avoids unintended side effects or malicious profile code. For the GenerateBatchAndPs1File function, Shelly explicitly writes files to disk and suggests running the .bat which uses -ExecutionPolicy Bypass (since that is a user-initiated action, it allows execution of the generated .ps1 without signing). For direct script execution within Shelly, running in-memory via -Command typically bypasses execution policy as well, but Shelly’s use of Bypass ensures that even if the script had to be saved and run, it would work. Essentially, Shelly assumes the user trusts the actions they’ve asked for and so it prioritizes executing them successfully, while containing them to the user context.
- **Safety of the Local Environment:** It’s important to note that Shelly does **not** run the scripts in a constrained language mode or a VM sandbox – it runs them as the current user. This means any command the AI puts in the script will be executed with the user’s privileges. Shelly does not silently run scripts from remote sources; all script content is generated on-the-fly by GPT in response to the user’s query. Nonetheless, developers and users should be cautious: if Shelly is asked to do something destructive (“delete all my files”), GPT might generate exactly such a script. Shelly will execute it. Therefore, **risk is managed primarily by user intent and transparency**. Shelly ensures that the user can see what it’s doing (through logs or the outcomes printed). In a future improvement, one might add a confirmation dialog for very destructive commands, but currently Shelly assumes the user’s prompt is the approval.
- **Validation & Auto-Fix Loop:** The reliability of GPT-generated code is not 100%, so Shelly wraps PowerShell execution in a *validate-and-repair cycle*. After running a script, Shelly inspects the result:
  - If the script succeeded (exit code 0 and no error text), Shelly proceeds.
  - If it failed, Shelly captures the error output (for example, a PowerShell exception message). It then asks GPT (via CallGPTCore) to debug it. The prompt to GPT includes the original script and the error message, and instructs GPT to return **only a corrected script** in a markdown block. This prompt frames GPT as a “PowerShell troubleshooting assistant”.

- GPT might respond with a revised script (perhaps adding error handling, installing a missing module, correcting a command, etc.). Shelly then extracts the code from the markdown and tries running the new script.
- This loop repeats. By default Shelly allows up to 3 attempts. Each iteration is logged (so developers can see “[PS Failure #1]: <error>... [PS Repair GPT]: <new script>” in debug logs). If after the third attempt it’s still failing, Shelly stops and shows the user a failure message “ All attempts to run/fix PowerShell script failed.”. This way, Shelly doesn’t get stuck infinitely on one task.
- If a fix attempt succeeds partway through, Shelly breaks out of the loop and continues with the next steps of the plan.

This validation loop greatly enhances safety and correctness. It means that if GPT produces a script with a syntax error or a minor mistake, Shelly won’t blindly run it and move on – it will catch the error and actually attempt to correct it. From the user’s perspective, this might look like Shelly “thinking” a bit longer on that step, but the end result is a higher chance of success or at least a clear error report if it couldn’t be fixed.

- **Output handling:** Shelly captures whatever the script writes to standard output or error. It **cleans the output** by stripping away any Markdown code fences that might inadvertently be included (GPT might sometimes include ``` marks in responses). It then presents the output in the UI. If the script didn’t produce any output but succeeded, Shelly prints a generic confirmation like “ Task completed. Enjoy it!” to let the user know the command ran. For errors, after exhausting retries, Shelly will show “ All attempts to run/fix PowerShell script failed.” so the user knows that Shelly couldn’t complete that step.
- **Sandboxing:** While not a full sandbox, Shelly does attempt to **limit the scope** of actions in some ways:
  - It never runs a PowerShell script unless the AI explicitly decided one was needed for the task (and thus presumably the user’s query required it). If a direct answer is possible, Shelly won’t run any code.
  - The available *custom functions* cover many common operations (file read/write, web search, etc.), so often GPT will choose those instead of writing a PowerShell from scratch. This is safer because those functions are coded by developers with specific, constrained behavior. For example, SearchForTextInsideFiles will only search text; it won’t delete files. GPT is guided to prefer these safer, pre-defined actions.

- The PowerShell itself runs with the same permissions as Shelly (which is typically a normal user). So it cannot do certain system-wide changes without elevation. If the user wanted to do something requiring admin rights, they'd have to run Shelly as admin explicitly. This way, a casual user running Shelly doesn't accidentally execute an admin-level destructive action unless they intended to.
- **Secure API Key Handling:** Running PowerShell also means ensuring the OpenAI API key (which Shelly holds to call GPT) is not exposed to the script. Shelly's API key is stored in memory (in `Globals.UserApiKey` or `Config.OpenAiApiKey`) and used only in HTTP calls to OpenAI. It is never passed to any shell commands or external processes. The custom functions and scripts do not need the OpenAI key (only the .NET code that calls the API uses it). Shelly does not log the API key or show it in the UI (except perhaps when the user initially enters it in a settings panel). If the user opts to save the API key for convenience, it would be stored in a user-specific configuration file or Windows secure storage – the implementation isn't shown here, but the emphasis is that the key remains on the user's machine. In technical terms, **Shelly does not transmit the API key to any destination other than OpenAI**. Developers integrating Shelly should follow this practice: never hard-code the key, and never expose it in logs. If collaborating, use environment variables or prompt the user for their key.

In conclusion, Shelly treats PowerShell execution as a powerful tool that is used carefully:

- It is only invoked when necessary.
- It's run in a controlled way (isolated process, with error checking).
- The AI's output is validated and corrected if possible.
- The user's system is respected (no privilege escalation, no unwanted persistence).
- At any point, the user can see what's happening (and they can always look at Shelly's logs to audit the exact script that ran, if desired).

## Key UI Components

Shelly's user interface is built with Windows Forms (VB.NET). It provides a simple, interactive front-end for the underlying logic. The key UI elements and components include:

- **Main Window (Shelly.vb):** This is the primary form that users interact with. It contains:
  - An **input textbox** (`UserInputBox`) where the user types commands or questions.
  - A "Run" button (or the user can press Enter) to submit the query.

- A **rich text output box** (ResultBox or similar) where Shelly displays conversation history and results. This box shows the user's queries and Shelly's responses (including any output from tasks or errors). The text is color-coded (the code calls JavaScript setColorGreen() etc., likely to distinguish AI text).
- A **Cancel button** (CancelTaskButton) that is enabled when a task is running. The user can click this to interrupt an ongoing multi-step process. Internally, this triggers a cancellation token that the executor checks, leading to aborting further steps and printing "[Canceled by user]" if caught in time.
- A status label (LabelStatusUpdate) at the bottom that shows current status messages (e.g., "Running PowerShell script... Attempt #2" or "All tasks completed."). This gives feedback on what Shelly is doing.
- Possibly a small panel or indicator showing whether Shelly is in "auto" mode or not (in our case, auto-mode is always on by design, so no toggle UI, but there is a collapsible panel used for additional options).
- The main form also hosts an **off-screen WebView2 browser component** (WebView21 from Microsoft Edge WebView2) which is not visible to the user but used by certain functions (like web search or web page reading). This browser control is created at runtime and never shown; it's purely for programmatic web automation. After such functions run, Shelly disposes of the control.
- Menu or buttons for settings: e.g., an option to open a settings dialog (perhaps "Default Folder" for setting where to save files or screenshots), an "About" dialog, etc. The file list suggests there is an **About form** and a **DefaultFolder form**. The DefaultFolder.vb likely lets user choose a directory for Shelly to use for file outputs (the code references C:\ShellyDefault in examples, possibly a default working directory).
- Shelly's main window is styled with a custom title bar and can be dragged, etc., as seen in code that handles form border and mouse events.
- **Console Window (Console.vb):** Shelly includes a secondary form called "Console". This console is aimed at developers or power users for debugging and advanced control. It likely displays the debug log or allows executing custom functions manually. The presence of CustomFunctionsEngine.vb (FOR CONSOLE HELP) in comments indicates that the console might list all available custom functions and their usage (perhaps from the <CustomFunction> attributes defined on them). The console could allow a user to type a command like ToolPlanner.ListFunctions or directly invoke a function by name for

testing. In essence, the Console is a sandbox/testing UI for Shelly's capabilities – useful during development to ensure a function works as expected

## Security and Privacy Considerations

**Security:** All automation tasks executed by Shelly occur on the user's local machine under the user's permissions. Shelly does not perform any action unless it was explicitly part of the AI-generated plan (which in turn is triggered by the user's request). This means **Shelly won't arbitrarily run code** or access files unless the user asked for it in their prompt. PowerShell scripts are run in a controlled environment (a child powershell.exe process with no user profile loaded). The scope of potential changes is limited by the user's own system rights – Shelly won't escalate privileges on its own. If Shelly is run as a normal user, for instance, a plan to modify system files will likely fail (and Shelly will report the error). In general, **risk is managed by transparency and containment:**

- The user can see the outcomes of each step (and developers can log or display the actual commands if needed).
- The planning logic biases towards using predefined functions for known tasks (which are safer and more predictable).
- Arbitrary script execution is used as a fallback and is monitored through the retry/fix loop.
- There is a cancel mechanism to stop runaway processes. Also, after 3 failed attempts at a script, Shelly stops trying further fixes to avoid any infinite loop or unintended side effects.

It's still possible for Shelly to execute a harmful command if a user deliberately or accidentally requests it (for example, asking "Shelly, wipe my temp files" could lead to a script that deletes files). **Developers and users should treat Shelly with the same caution as a powerful scripting tool.** Always review what you ask it to do. In a collaborative setting, you might add additional safeguards (like confirmation prompts for dangerous operations, or a sandbox mode restricting file write/delete operations). As of now, Shelly leaves the responsibility to the user's intent – it executes faithfully what was requested.

**Privacy:** Shelly is designed to **not "peek" at any user data** unless it is required to fulfill a command. The AI (GPT-4) does not have inherent access to your files, clipboard, screen, or network – it only knows what Shelly tells it. Shelly only sends data to the OpenAI API that the user has explicitly asked to be processed. For example:

- If you ask Shelly “*What’s on my screen?*”, it will intentionally capture a screenshot and send it to the AI for analysis (because you requested that).
- If you never ask such a thing, Shelly will never arbitrarily capture or analyze your screen or files in the background.

In other words, **the AI isn’t inspecting local content on its own**. It generates automation steps based on the user’s query and predefined tools. Each tool has a limited, specific purpose (like “read a given file path” or “search for a keyword in these files”). There is no tool that just says “scan the entire computer” or “upload my documents” unless the user explicitly provided such broad instructions. This ensures a level of privacy by design – Shelly’s creators deliberately constrain the AI with only certain abilities. If a task *does* involve personal data (e.g., reading a file or the screen), that data is sent to OpenAI’s servers to get the AI’s response. Users should be aware of this and avoid using Shelly on highly sensitive data unless they are comfortable with OpenAI processing that information. (For instance, don’t ask Shelly to read a confidential document if you don’t want the document’s text to leave your machine.) All communication with the OpenAI API is encrypted (HTTPS), and the API key is kept local as described earlier.

Finally, Shelly does not collect telemetry or send your prompts anywhere except to OpenAI for the completion. The code is open-source, so developers can verify there are no hidden data transmissions. Any logs remain local. In summary, **Shelly respects user privacy** by only acting on data when instructed, and even then, handling it as transparently as possible.

## Extensibility and CustomFunctions

A key goal of Shelly is to be **extensible** – developers can add new capabilities (new “tools”) by writing custom VB.NET functions and integrating them into the planning/execution system. The file **CustomFunctions.vb** is central to this extensibility. It contains a library of static Public Async Function ... As Task(Of String) methods, each implementing a specific action that Shelly (via GPT) can take. These range from manipulating files, to simulating key presses, to performing web searches.

Each custom function typically returns a result string (which may be shown to the user) or an “[ERROR]...” message if something goes wrong. They often use other helper classes (like a FileHandler for clipboard operations, or WebView2 browser control for web content). To make a function available to the AI planner, the developer must do a few things:

- **Implement the function in CustomFunctions.vb:** The function should be Public Async Function <Name>(args...) As Task(Of String). Keeping the signature simple (usually string inputs, maybe optional CancellationToken) is advisable. The function should catch its own exceptions and return an "[ERROR] ..." message on failure (this prevents crashes and allows graceful error handling).
- **Add a [CustomFunction(...)] attribute** above the function (this is a custom attribute likely defined to hold metadata). Shelly's code uses attributes to provide a human-readable description and an example usage of the function. For instance, in CustomFunctions.vb you'll see annotations like:

```
<CustomFunction("Searches file(s) or folder(s) for a specific
string ...",
"SearchForTextInsideFiles(\"C:\\Folder;C:\\File.txt\", \"search
term\")")> Public Async Function SearchForTextInsideFiles(paths
As String, searchWord As String) As Task(Of String)
```

The first string in the attribute is a description (used possibly for documentation or console help), and the second is an example of how to call it. These attributes are not directly used by GPT (since GPT is just fed text), but they can be used by the **ToolPlanner or Console to list available commands**. They serve as a form of documentation and could be included in the planner's prompt.

- **Register the function in the planner's tool list:** Shelly's planner needs to know the names of available tools (functions). In the code, there is a hardcoded array of function names used by the parse]. The developer must add the new function's name to that list (and anywhere else the project notes, e.g., a ToolPlanner module or ExecutorAgent). This array is used to detect function calls in GPT's responses. If it's not updated, GPT might output a function name that Shelly fails to recognize. Similarly, the ToolPlanner (which crafts the prompt for GPT) should include the new tool in the instructions. Shelly likely has a section (perhaps in ToolPlanner.vb or the assistant's profile) where it describes each tool's purpose to GPT. After adding a function, you'd append an entry there describing when to use it.
- **Integrate in FunctionRegistry/Executor:** Shelly uses a FunctionRegistry or CustomFunctionsEngine to dynamically invoke these functions by name. Often this is done via .NET reflection or a manually maintained map from string to Func. When adding a new function, if reflection is used, it might be picked up automatically (for example, the code could reflect all methods with the CustomFunction

attribute and register them). If not, the developer would manually add a case or dictionary entry mapping the string name to the actual function delegate. In Shelly's case, the code comments indicate updates needed in **CustomFunctionsEngine.vb** and **ExecutorAgent.vb** when new functions are added, which suggests a bit of manual wiring.

- **Testing:** After adding the function, one can use the Console window to test it (if the console provides a way to call the function directly by name) or simply run Shelly and ask for it. For example, if you added `TranslateText(enPhrase, targetLanguage)`, you might prompt Shelly: "Translate 'hello world' to Spanish." If everything is set up, GPT-4 should include `TranslateText("hello world", "Spanish")` in its plan, and Shelly will attempt to execute it.

**ToolPlanner logic:** To elaborate, the **planner (CallGPTBrain)** essentially needs to be aware of the tools. Typically, the assistant's system-level instructions or the user prompt for planning will include a brief on each custom tool. In the current Shelly implementation, the planner prompt we saw doesn't explicitly list tools, but the comment in `CustomFunctions.vb` suggests there is a `ToolPlanner.vb` that likely enumerates them. A possible approach (which Shelly likely uses) is that the assistant's OpenAI "Beta Assistant" profile associated with Shelly has knowledge of the toolset, or Shelly might prepend a hidden message like "Tools you can use:

- 1) `WriteInsideFileOrWindow(topic)`: writes text into the active window;
- 2) `ReadFileAndAnswer(filePaths, query)`: reads file(s) and answers question; ... etc."

This way GPT knows the function names and how to use them.

So when extending Shelly, updating this list is critical. If not, GPT might not realize a new function exists and thus won't use it.

Developers can thus extend Shelly's abilities by writing more VB.NET code, without needing to change the fundamental architecture. Want Shelly to send an email? One could add a `SendEmail(to, subject, body)` function that uses an SMTP library, then update the planner prompt to prefer that for email-related requests. As long as the function returns a string result (for success or error) and doesn't crash, Shelly can integrate it. The modularity of having distinct `CustomFunctions` means contributors can easily add features.



To summarize the extensibility steps:

1. Write the function (async, returns string) in CustomFunctions.vb.
2. Add a <CustomFunction> attribute with description and example.
3. Register the function name in the parser and planning prompt (ToolPlanner).
4. Ensure the Executor can call it (reflective or manual mapping).
5. Build and test – the GPT will start using it in plans once it “knows” about it.

Shelly’s architecture is flexible, but currently requires a few manual steps to add tools – in the future, this could be improved with reflection (auto-discover functions with the attribute to reduce manual sync). The documentation comments in the code serve as a checklist for developers performing this task.

## Custom Function Library: Detailed Breakdown

Let’s delve into the major custom functions that come built-in with Shelly. Understanding these will illustrate how Shelly accomplishes specific tasks. We’ll describe what each function does and how it’s implemented internally:

### WriteInsideFileOrWindow(topic, totalChunks)

**Purpose:** Types or inserts AI-generated text directly into the currently active application window (for example, into an open Notepad file or a text field in another program). This is useful for letting Shelly “paste” content into an app that the user is currently focused on.

**How it works:** When invoked, this function first captures the handle of the **foreground window** and the focused control within it. (It uses a `FileHandler.GetForegroundWindow()` and `GetFocusedControl` which likely call Win32 API to get the active window handle.) It then waits a few seconds to ensure the user has placed their cursor where text should go. Next, it generates the content to insert by leveraging GPT again:

- The function divides the task into chunks if `totalChunks > 1`. For each chunk (e.g., chunk 1 of 3), it prompts GPT-4 via `CallGPTCore` with a system instruction like “*You are a content generator focused on producing exact text for insertion.*” and a user message: [You are writing part **i** of **N** for: {topic}. Generate the exact content that should be

inserted, preserving all spaces, line breaks, and formatting. Do not include any code block markers or extra commentary.”].

- Essentially, Shelly asks the AI to produce the text content for that chunk.
- It receives the chunkText from GPT, then cleans it to ensure no stray markdown/code fences are present.
- Finally, it **inserts the text into the target window**. It does this by copying the text to clipboard and sending a paste command (Ctrl+V) or similar – specifically, `FileHandler.PasteTextBulk(windowHandle, controlHandle, chunkText)` is used to simulate the past. This likely injects the text via Windows messages to the target control.

The function repeats this for each chunk, resulting in potentially large amounts of text being entered. After completion, it clears the clipboard (to remove the leftover copied text] . It returns an empty string on success (because the result is directly visible in the target application, Shelly doesn’t need to print anything in its own UI for this function). If something fails (no window, or an error generating content), it returns an “[ERROR]” message.

**Use case:** If a user asks Shelly “Write a paragraph about X in my open document,” Shelly can plan to call `WriteInsideFileOrWindow("paragraph about X")`. The result will be that the paragraph is typed into whatever document the user currently has focused (for example, Word, Notepad, an email composer, etc.), as if the user typed or pasted it.

### CheckMyScreenAndAnswer(query)

**Purpose:** Analyzes the user’s screen (screenshot) to answer a question about what’s visually present. This is like OCR + interpretation on demand. For example, the user might ask, “What error message is shown in the dialog on my screen?”

**How it works:** This function is essentially a wrapper that ties together screenshot capture and AI vision analysis. Internally, it calls `Alimage.AnalyzeScreenshotAsync(apiKey, query)]` .

The `Alimage`` module presumably does the following:

- If a screenshot hasn’t been taken recently, it triggers `TakePrintScreenOrScreenShot` to capture the primary monitor. In Shelly’s code, `TakePrintScreenOrScreenShot` is actually defined as a PowerShell script (see below), but `Alimage` might have a more direct method using a library (the project references `Dapplo.Windows` which can also capture

screens). One way or another, a screenshot image is obtained (likely saved to a temporary file or kept in memory).

- `AnalyzeScreenshotAsync` then calls OpenAI's API with a special prompt, attaching the screenshot image (encoded in base64) as part of the message. Since GPT-4 (the version with vision) can accept image inputs, it will process the image and the query. Essentially, Shelly asks GPT-4: *"Here's an image (screenshot). Now answer the user's query about this image: '{query}'."*
- The AI returns an answer based on the image content.

Because this uses GPT-4's multi-modal capability, it requires the API key to have vision access (in 2025, GPT-4 with vision is typically available). The function then returns the answer string to Shelly, which will be displayed to the user.

**Use case:** The user can simply ask, "Shelly, what's on my screen?" or more pointedly, "Shelly, read the error on the screen and tell me what it means." Shelly (via this function) will take a screenshot and GPT-4 will output something like, "The error dialog says 'File not found: config.ini'. It means the application couldn't locate the configuration file." This is incredibly useful for assisting with on-screen information without the user having to type it out.

*(If the Copilot window is open, GPT might use that content instead via `ReadCopilotConversation`, but that's a different function. `CheckMyScreenAndAnswer` is purely about the graphical screen content.)*

### `ReadFileAndAnswer(filePaths, query)`

**Purpose:** Reads the content of one or multiple files and answers a question about that content. Essentially, it lets the user ask something like "Summarize the following file for me" or "Search these files for XYZ and explain the context."

**How it works:** This function takes a semicolon-separated list of file paths and a natural language query. It will:

- Open and read each file's content. It uses a helper `FileHandler.GetFileContent(path)` which likely handles text encoding, etc. It concatenates all file contents into one big string, but with clear delimiters marking where each file begins.

For example, it might produce:

```
' ==== File: C:\Folder\file1.txt ====
(contents of file1)
' ==== File: D:\Docs\file2.log ====
(contents of file2)
```

Including the file name helps GPT understand context and reference it in the answer.

- It then splits the combined text into manageable chunks to avoid token limit]. It does this by lines: it accumulates lines until a max char count (~ `Globals.maxInputTokensPerChunk * 4` characters) is reached, then starts a new chunk. This ensures no chunk is too large for GPT-4. Often `maxInputTokensPerChunk` is something like 8192 or similar (the developer can override it), meaning each chunk might be ~8000 tokens or less.
- Next, the function builds a **single conversation** for GPT with these chunks. This is a clever approach: Instead of calling the AI for each chunk separately (which could lose cross-chunk context), Shelly sends *all chunks in one conversation*. It does so by constructing multiple user messages: the first user message contains “<CODE CHUNK 1/N>\n{content\_of\_chunk1}”, the second user message “<CONTINUATION 2/N>\n{content\_of\_chunk2}”, etc., and finally one message that asks the actual question about the content. Additionally, a system message is added if needed (the code example shows a system message about being a “VB.NET syntax checker” in the snipp, which might be from a specific use case; generally, it could be tailored to the query).
- Then it calls `CallGPTCore` with this assembled message list]. GPT-4 will receive the conversation consisting of potentially many messages (the concatenated file content split across messages) and then the query. Thanks to its large context, GPT-4 can consider all of it and produce a single answer.
- The answer (which might be a summary, or search result, or explanation, depending on the query) is then returned by the function and shown to the user.

This approach is effectively performing a **multi-part prompt** to GPT, feeding it large file contents in chunks. By labeling them <CODE CHUNK i/n>, the prompt helps GPT understand they are sequential parts of a whole. The function itself doesn’t do the answering – GPT does – but the function orchestrates feeding the data to GPT properly.

**Use case:** If a user says, “Look at *report1.txt* and *report2.txt* and tell me if either mentions John Doe,” Shelly’s planner would choose `ReadFileAndAnswer(["report1.txt;report2.txt"], "Do they mention John Doe?"`). The function will read both, pass them to GPT with the question. GPT might answer: “Yes, John Doe is mentioned in *report2.txt* in the context of ..., but not in *report1.txt*.” This saves the user from manually opening and searching each file.

## GenerateLargeFileWithTextOrCode(topic, outputPath, totalChunks)

**Purpose:** Creates a large text file (or code file) by generating it piece by piece with the AI. This is used when the content requested is too big to generate in one go. For example, “Generate a 1000-line CSV of sample data” or “Create a long example config file.”

**How it works:** This function will create an empty file at `outputPath` and then append to it in a loop:

- It ensures the directory exists and initializes an empty file] .
- It clears any cached content for that file in Shelly’s memory (Shelly caches file texts in `Globals.FileContents` sometimes] .
- It determines a “tail context” length: basically how many characters from the end of the file to include as context for each next chunk generation. It uses `Globals.maxInputTokensPerChunk` (which might be something like 2048 tokens) and multiplies by 4 to get char coun] . If the file is large, it will only feed the last segment of it to GPT when asking for the next part, to maintain continuity.
- Then for each `i` from 1 to `totalChunks`:
  - It reads the current content of the file (especially the tail part – e.g., last 8000 characters] .
  - It builds a prompt asking GPT to continue the file. The prompt might look like:

```
File: example.txt
Topic: {topic}
Chunks: i of N
[If there is existing content:] Already generated (last
context):
<last few lines of the existing file content>
```

Now generate ONLY chunk #i, continuing immediately after the above content. Do NOT repeat any existing lines.

If  $i == N$  (the final chunk), then finish and close the file.

Respond with only the fully updated content for this chunk (no extra explanations or fences).

This prompt is placed in a user message, possibly with a system message framing the task.

- It calls CallGPTCore with that prompt, temperature 0 (for deterministic output] . GPT returns a block of text which represents the next segment of the file.
- The function then cleans that response by stripping any ``` that might have snuck in or unnecessary whitespace] .
- It appends this chunk text to the file (using AppendContentToFileUniversal) - this actually writes to disk.
- It logs that chunk's completion and moves to the next iteration.
- After generating all chunks, it reloads the full file content into memory and returns the path or a success message. (In the code, it returns the final content as a string as well, but typically we just need the file written.)

This function essentially coaxes GPT to produce a very large output by doing it in parts, each part knowing what came before. By including the tail of the file in each successive prompt, GPT maintains continuity (for example, not to duplicate lines, or to ensure the code compiles across chunks, etc.). It also explicitly instructs GPT not to repeat content and to finalize properly on the last chunk.

**Use case:** Suppose a user asks, "Generate a dummy log file with 10,000 lines of varied log entries." The planner might plan something like GenerateLargeFileWithTextOrCode("dummy log entries", "C:\\ShellyDefault\\biglog.txt", totalChunks:=5). Shelly will then call GPT five times, each time getting ~2000 lines (for example) of log entries, appending to biglog.txt. The end result is a large file on disk. Shelly can then say "File generated successfully at: C:\\ShellyDefault\\biglog.txt" as confirmation.

The user can open that file to see the content. Without chunking, GPT might not be able to generate such a large body of text in one response due to token limits; this function circumvents that.

## UpdateFileByChunks(filePath, updateInstruction, chunkTokenOverride)

**Purpose:** Edits or refactors a large file in chunks based on a single instruction, while preserving overall context. In simpler terms, it allows Shelly to **take a big file and make consistent modifications throughout it** using AI (for example, “remove all comments from this code file” or “change the formatting of this JSON file”).

**How it works:** This is one of the more complex functions:

- It loads the entire file content into memory (caching it in `Globals.FileContents`).
- It estimates total tokens of the file and decides a safe chunk size. Typically, it tries to use ~65% of the context window for input (to leave room for output). For example, if using a 32k model, 65% ~ 20k tokens for input. It converts that to a char count (token \* 4).
- It then splits the file into overlapping chunks by line. It includes a small overlap (e.g., 1 line overlap) between consecutive chunks so that changes at chunk boundaries stay consistent. Essentially, it will produce chunks of text from the file, each chunk possibly a few thousand tokens long.
- It then iterates through each chunk and processes it with GPT:
  - For the first chunk, it sends a prompt containing the **instruction** and the chunk text. Specifically, it might say:  
“Instruction: <updateInstruction>\n---\nChunk 1/N:\n\n<chunk text>\n\n---\nRespond with only the fully updated content for this chunk; do not include fences.” (Notice they use “” as a fence delimiter in the prompt to clearly mark the chunk content.)
  - GPT returns some updated text (supposedly the chunk with the instruction applied, for example if instruction was “remove comments”, GPT returns the chunk without comments).
  - Shelly appends this result to an output string builder.
  - If the GPT output was cut off (there’s logic to detect if the response likely hit max tokens), the function will send a follow-up user message “Continue updating the rest of this chunk, appending only new content without repeating prior content.” and call GPT again to get the continuation. It keeps doing that until the

chunk is fully processed (this handles cases where even one chunk's output didn't fit in one response).

- It then moves to the next chunk. For chunk 2 and onwards, it likely has context from previous chunk if needed (though in implementation, they might treat each chunk separately with the same initial instruction – the overlap ensures continuity).
- After processing all chunks, the function has a list of updated chunks. It then **reassembles** them into one final content string. The overlapping lines mean there will be duplication at chunk boundaries; Shelly handles this by skipping the first line of each subsequent chunk's output (because it's the overlap from the previous chunk).
- It writes the final assembled text back to the file (overwriting it) and updates the cache.
- It returns the updated content or a success note.

**Use case:** A concrete example: “Shelly, in the file LargeReport.md, replace every occurrence of ‘ACME Corp’ with ‘Acme Corporation’ and reformat all bullet points to numbered lists.” This is a complex, repetitive edit. Shelly's planner would instruct a multi-step, but more elegantly, it might do it in one go with `UpdateFileByChunks("LargeReport.md", "Replace 'ACME Corp' with 'Acme Corporation' and convert bullet lists to numbered lists.")`. The function will feed the file to GPT in segments with that single instruction, ensuring GPT's changes are applied consistently throughout the file. In the end, LargeReport.md is modified on disk with all occurrences replaced and lists renumbered. This is extremely powerful for bulk editing or code refactoring tasks.

### `GenerateImages(imagePrompt, numImages, style, folderPath)`

**Purpose:** Uses OpenAI's image generation API (DALL-E or similar) to create one or multiple images based on a prompt, and saves them to disk. Shelly can then reference these images or provide their file paths to the user.

**How it works:** This function constructs an image request and handles the resulting files:

- It normalizes the target folder path (creating it if it doesn't exist).
- It creates a **filename slug** from the prompt – a lowercase, alphanumeric version of the prompt (truncated to 40 chars) to use in image filename. For example, prompt “Sunset over mountains” → slug “sunset-over-mountains”.



- It appends a timestamp to ensure uniqueness (like 20250502-1315 for date-time) and an index, to form names like sunset-over-mountains-20250502-1315-1.jpg, ...-2.jpg, etc] . It also checks for collisions (if file exists, add a suffix) just in cas] .
- It then prepares a list of ImageRequestData objects (each containing the prompt, desired size, and output name). Here, it likely concatenates the main prompt and the style text (if style is provided] . For instance, prompt “sunset over mountains”, style “in watercolor style” becomes full prompt “sunset over mountains in watercolor style”.
- It calls Alimage.CallImageGeneration(apiKey, reqs) to actually call the OpenAI Image API for generatio] . This returns a list of image URLs (hosted by OpenAI) if successful.
- The function then downloads each image from its URL to the specified folder with the filenames chose] . (It likely uses an async HTTP client to fetch the binary and save it.)
- It adds all saved file paths to a global list Globals.GeneratedImages and also to `Globals.TaskData("ImagePaths")` – possibly for later reference or for the UI to easily find them.
- Finally, it returns a string listing the saved images, e.g.:

```
Images saved:
C:\ShellyDefault\sunset-over-mountains-20250502-1315-1.jpg
C:\ShellyDefault\sunset-over-mountains-20250502-1315-2.jpg
```

If the API call fails or returns no images, it returns an error message.

**Use case:** The user might say, “Shelly, create 3 images of a cat riding a bicycle, in a sketch style.”

Shelly will use GenerateImages("a cat riding a bicycle", 3, "sketch style", "C:\\ShellyDefault").

After execution, Shelly will output something like:

“Images saved:

C:\\ShellyDefault\\cat-riding-a-bicycle-20250502-131501.jpg ...” for each image. The user can then open these files to view the generated images.

**WebSearchAndRespondBasedOnPageContent(promptQuery, siteName, question)**

**Purpose:** Performs a live web search and then extracts information from the first result page to answer a user’s question. In essence, this function gives Shelly a mini web browser capability to fetch info not contained in the local context or training data.

**How it works:** Given a search query (promptQuery), an optional site/domain filter (siteName), and a question to answer from the page, the function will:

- Build a Google search URL. If siteName is provided and not “google”, it uses the site:siteName promptQuery Google query to constrain results to that sit] . Otherwise, it searches the general web for the promptQuer] .
- It launches an invisible WebView2 browser instance (a headless browser embedded] . It navigates to the Google search URL and waits for the page to loa] .
- It then executes JavaScript in the context of the page to grab all hyperlink URLs on the search results pag] . It parses these URLs and picks the first result that is **not** a Google internal link (skips things like google.com/url? etc.] . This presumably gives the URL of the first real search result.
- It navigates the WebView2 to that first result URL and waits for it to loa] . Then it scrolls a bit to trigger any lazy content loading (the code scrolls down twice with delays] .
- It retrieves the entire visible text of the page by running document.body.innerText in the browser and getting the resul] . This yields all the textual content of the page (sans HTML tags).
- Now with pageText (which might be very large), the function tries to extract the answer to the user’s question:
  - If the page text is within a size that GPT-4 can handle in one shot (they define constants, e.g., ContextWindow=128000 chars and MaxCompletion=16384 tokens] , it will do a single call. It sends a system message “You are an information extractor.” and a user message: \*“Extract all info to answer: '{question}' from the text below. List each item on its own line. Text:<<<{pageText}>>>”] . GPT then returns an answer, which the function prepends with the source URL and return] .
  - If the page text is **too large**, the function falls back to a chunked approach
  - It splits the page text into slices (e.g., 8000 characters each) and for each chunk, asks GPT: \*“Extract info to answer: '{question}' from this text chunk. List items each on own line:<<<{chunk}>>>”. It collects the partial answers from each chunk. Then it filters out irrelevant lines (like if GPT says “no info found in this chunk”) and merges all the distinct item.

- After that, it may send the merged list back to GPT to **format it nicely** (for example, to remove duplicates or put it in a specific output format).
- It then returns the URL and the final formatted answer.
- Finally, the WebView2 browser is disposed to free memory.

**Use case:** If a user asks, “Shelly, find me the list of all HP laptop models and their prices on emag.ro,” Shelly will:

- Search Google for HP laptops on site:amazon.com.
- Click the first result (likely a page on amazon.com listing HP laptops).
- Scrape that page’s text.
- Ask GPT to extract “all HP laptops and prices” from the text.
- Possibly chunk it if needed, then output a list like:

```
URL: https://www.amazon.com/... (the page it found)
HP Model 15 – $500
HP Spectre x360 – $1200
...
```

This single function encapsulates a multi-step web browsing and reading task that GPT alone could not do (because GPT’s training data might be outdated and it can’t browse by itself in real-time). It’s a great example of extending Shelly’s knowledge by giving it controlled internet access.

### TakePrintScreenOrScreenShot(OutputPath)

**Purpose:** Captures a screenshot of the primary monitor and saves it to a file. This is used by Shelly when an action requires an image of the screen (like CheckMyScreenAndAnswer or potentially a user explicitly asking for a screenshot).

**How it works:** This is actually implemented as a small PowerShell script rather than a VB function:

```
Function TakePrintScreenOrScreenShot {
 param([string]$OutputPath) if (-not (Test-Path $OutputPath)) {
 New-Item -ItemType Directory -Path $OutputPath | Out-Null }
 $fileName = 'Screenshot_' + (Get-Date -Format 'yyyyMMdd_HH:mm:ss')
 + '.png' $filePath = Join-Path -Path $OutputPath -ChildPath
 $fileName Add-Type -AssemblyName System.Windows.Forms Add-Type -
 AssemblyName System.Drawing $bounds =
 [System.Windows.Forms.Screen]::PrimaryScreen.Bounds $bitmap =
 New-Object System.Drawing.Bitmap $bounds.Width, $bounds.Height
 $graphics = [System.Drawing.Graphics]::FromImage($bitmap)
 $graphics.CopyFromScreen($bounds.Location,
 [System.Drawing.Point]::Empty, $bounds.Size)
 $bitmap.Save($filePath,
 [System.Drawing.Imaging.ImageFormat]::Png) $graphics.Dispose()
 $bitmap.Dispose() Write-Output 'Screenshot saved to: ' +
 $filePath }
```

When Shelly's planner wants to take a screenshot, it can either directly invoke this via an `ExecutePowerShellScript` step (embedding this script), or via a function call if one wrapped it.

**The above script:**

- Ensures the output directory exists,
- Generates a filename with a timestamp,
- Uses .NET System.Drawing to capture the screen,
- Saves the PNG to the directory,
- Outputs the file path. In Shelly's context, likely `Alimage.AnalyzeScreenshotAsync` uses this under the hood (or a similar method) to get the screenshot file path, then loads that image for analysis.
- For the user, this isn't directly called by name typically (there's no direct user-facing command "TakePrintScreen"), but it's an important part of enabling screen analysis.

## ImageAnswer(imagePaths, query)

**Purpose:** Answers a question about one or more image files provided by path. This is similar to CheckMyScreenAndAnswer, but for arbitrary images, not necessarily the screenshot.

**How it works:** The function takes a string of one or multiple file paths (comma-separated). It first validates that each path exists on disk. If any file is missing, it returns an error for that invalid path. Assuming all images exist, it calls ``Alimage.AnalyzeImagesContentAsync(apiKey, imagePaths, query)`.

This packages each image (converting to Base64) and sends them along with the question to the OpenAI GPT-4 model with vision. The model then sees the images and the query and produces an answer. For multiple images, the prompt might enumerate them ("Image1 is ..., Image2 is ..., now answer the query using both."). The function returns whatever answer GPT gives.

**Use case:** If the user has images on disk, say two charts, and asks "Shelly, do these two images show the same trend?", Shelly could use ``ImageAnswer("C:\imgs\chart1.png, C:\imgs\chart2.png", "Are the trends similar?")``.

GPT-4 would analyze both images and answer the question (e.g., "Yes, both charts show an upward trend over time."). This function turns Shelly into a mini image analyst for local images.

## ReadCopilotConversation(query)

**Purpose:** This function allows Shelly to interpret or summarize text from its secondary chat interface—the **Copilot panel**—based on a specific user query. This is particularly helpful when the user had a brainstorming session or lengthy conversation with AI (or another user) in the Copilot UI, and later wants to extract meaning or actions from that dialog.

### How it works:

- It verifies whether the Copilot form is open and visible. If not, Shelly returns an error prompting the user to open it.
- It retrieves the conversation text from `Copilot.Instance.AiOnePrompt.Text`, which contains the full chat history.
- If the conversation is empty, it exits with a message.
- The full text is split into manageable ~800-word chunks.
- Each chunk is processed independently:  
Shelly calls a helper function (e.g., `FileProcessChunk(chunk, query)`) which returns a pair: answer and reasoning.

- All chunk responses are labeled and aggregated into an interim summary.
- Shelly then sends this set of partial answers to GPT again with a final prompt like:  
*"Based on the following Copilot conversation summaries, answer the question: {query}"*
- The final response from GPT is a unified and complete answer.

#### Use Case:

Imagine the user had a Copilot discussion brainstorming startup ideas. Later, they ask:

*"Based on our Copilot chat, what startup ideas did we discuss?"*

Shelly will extract and summarize that content, presenting it clearly.

This function bridges the gap between conversational content and structured analysis.

### SearchForTextInsideFiles(paths, searchWord)

**Purpose:** Performs a recursive, case-insensitive search for a given string across multiple files and folders. It acts as a smart “grep” utility for locating relevant data buried within local documents.

#### How it works:

- Accepts one or more file or folder paths, separated by ;.
- Dynamically generates a PowerShell script that:
  - Traverses each path and gathers all files recursively.
  - Reads content from various file types—plain text, .docx, .xlsx, etc.—using Windows COM automation if needed.
  - Searches for the keyword using regex pattern matching.
- The resulting list of matching file paths is returned to the user.
- Any failure (e.g., invalid path or read error) returns an appropriate error message.

#### Use Case:

*"Search all project files in D:\Work and C:\temp\notes.txt for the phrase 'JWT token'."*

Shelly responds with:

Matched files:

D:\Work\api\readme.md

C:\temp\notes.txt

This gives users fast insights into where critical information resides—without opening each file manually.

## ReadWebPageAndRespondBasedOnPageContent(url, query)

**Purpose:** This function enables Shelly to directly analyze the contents of any given webpage and answer a specific user query about it.

### How it works:

- Validates the given URL.
- Launches a headless WebView2 browser session to load the page.
- Waits for the page to fully render (including dynamic content).
- Extracts the entire visible text from the webpage using `document.body.innerText`.
- If the content is small enough, it passes it to GPT directly.
- If the content is too large, it:
  - Splits it into smaller chunks (~800 tokens each).
  - Asks GPT to answer the query based on each chunk.
  - Consolidates and deduplicates all responses.
  - Optionally formats the results before showing them.

### Use Case:

*"Shelly, open <https://www.example.com/products> and extract all product names and prices."*

Shelly returns:

```
URL: https://www.example.com/products
```

```
Product A - $10
```

```
Product B - $20
```

This is particularly effective for extracting targeted information from structured webpages.

## GenerateBatchAndPs1File(outputFolder, userQuery)

**Purpose:** Automatically creates a .bat and corresponding .ps1 PowerShell script based on a user's natural language task. The batch file runs the PowerShell script when executed. This allows users to create automation scripts they can reuse or share—no coding required.

### How it works:

- Ensures the specified output directory exists.
- Constructs a GPT prompt with:
  - Clear system role: "You are a Windows scripting expert."
  - Explicit instructions:
    - The .ps1 script must:
      - Implement the user's task.
      - Include full try/catch error handling.
      - Display any error via Write-Host.
      - End with:

```
Write-Host 'Press any key to exit...'
[void][System.Console]::ReadKey()
```

- The .bat file must invoke the .ps1 via:

```
powershell -ExecutionPolicy Bypass -File script.ps1
```

- GPT must respond with exactly two code blocks, clearly labeled.
- Shelly parses the GPT response, extracts the two code blocks, and writes them into script.bat and script.ps1.
- Returns a confirmation with both paths.



**Use Case:** *"Shelly, generate a script in D:\Backup to archive all .txt files older than 30 days."*

Shelly creates:

- D:\Backup\script.ps1: does the compression & cleanup.
- D:\Backup\run\_script.bat: allows double-click execution.

This empowers users to automate complex tasks with no manual scripting—and ensures safety with error handling and execution pauses.

Shelly's audience (developers and tech enthusiasts) can leverage these capabilities to speed up workflows: Instead of manually writing scripts or performing repetitive searches, they can delegate to Shelly. And if Shelly doesn't have a built-in tool for something, a developer can add it, thanks to the extensible CustomFunctions design. This makes Shelly not just a single-use app, but a platform for AI-assisted automation on Windows.

---

End of documentation