Production-Ready Implementation Guide: The Mimo "Goldilocks Stack"

This document provides a comprehensive, production-ready implementation guide for the four core modules of the Mimo system, collectively referred to as the "Goldilocks Stack." This stack was chosen to balance the "Zero-Dependency" philosophy with the practical needs for stability, security, and maintainability in a production environment. The selected libraries— `Req` for networking, `Floki` for HTML parsing, and `Exile` for process management—are robust, well-maintained, and specifically address critical edge cases such as SSL/TLS security, process zombie states, and blocking I/O operations. Each module is presented with its full Elixir source code, a detailed explanation of its architecture, and a thorough analysis of how the underlying library mitigates the specified edge cases. The goal is to provide a set of skills that are not only functional but also resilient and secure, forming a solid foundation for the larger Mimo application.

1. Mimo.Skills.Network: The Robust HTTP Fetcher

The `Mimo.Skills.Network` module serves as the system's gateway to the external world, providing a secure and reliable interface for making HTTP requests. It is a wrapper around the `Req` library, chosen for its "batteries-included" philosophy, which offers sensible defaults for crucial aspects like SSL verification, automatic decompression, and retry logic . This design choice significantly reduces the boilerplate code required to make a safe HTTP request, allowing developers to focus on business logic rather than low-level network configuration. The module exposes a single, clean interface, `fetch/2` , which enforces a strict timeout and a custom "Mimo/2.0" User-Agent, ensuring that all outgoing requests are identifiable and bounded in their execution time. By handling errors gracefully and returning a consistent data structure, it prevents network-related failures from cascading through the system, thereby enhancing overall stability.

1.1. Module Implementation

The implementation of `Mimo.Skills.Network` is designed for simplicity and safety. It encapsulates the complexities of the underlying `Req` library, presenting a straightforward API that is difficult to misuse. The module's structure is centered around the `fetch/2` function, which takes a URL and a set of options, merges them with secure defaults, and executes the request. This approach ensures that every

request, regardless of its specific parameters, adheres to the system's security and performance standards. The code is structured to be both readable and efficient, leveraging Elixir's pattern matching and keyword list manipulation to create a flexible yet controlled execution environment.

1.1.1. Core `fetch/2` Function

The `fetch/2` function is the primary entry point for the `Mimo.Skills.Network` module. It is designed to be a versatile and robust HTTP client that abstracts away the intricacies of network communication. The function takes two parameters: `url`, which is the target endpoint for the request, and `opts`, a keyword list of options that can override the module's default behavior. This design allows for flexibility while ensuring that a baseline of security and performance is always maintained. The function's body is structured to first merge the provided options with a set of predefined defaults, which include a strict timeout and a custom User-Agent string. This merging process ensures that even if a caller does not specify these critical parameters, the request will still be executed within a safe and identifiable context. The function then proceeds to construct the request, handling different HTTP methods and potential request bodies, before dispatching it via the `Req` library. The final step involves processing the response from `Req`, which is then transformed into a simple, predictable map structure. This consistent output format is a key aspect of the module's design, as it simplifies error handling for the calling code and prevents unexpected crashes due to malformed or unexpected response structures.

elixir Copy

```elixir
defmodule Mimo.Skills.Network do
  @moduledoc """
  A robust HTTP fetcher built using Req.
  Ensures SSL verification, handles timeouts and retries
automatically.
  Returns a consistent response map regardless of success or error.
  """

  @default_timeout 10_000
  @user_agent "Mimo/2.0"

  @doc """
  Fetches a URL with a strict timeout and custom User-Agent.

  ## Parameters
```

```
    - `url`: The URL to fetch.
    - `opts`: A keyword list of options to pass to Req. Defaults are
merged in.

  ## Options
    - `:method`: The HTTP method to use (e.g., `:get`, `:post`).
Defaults to `:get`.
    - `:json`: A map to be encoded as the request body for POST
requests.
    - Other options are passed directly to `Req`.

  ## Returns
    A map with the following keys:
      - `:status`: The HTTP status code (e.g., `200`) or an error
atom (e.g., `:error`).
      - `:body`: The response body as a string or an error message.
      - `:headers`: A map of response headers or an empty map.
  """
  def fetch(url, opts \\ []) do
    # Merge options: our defaults take precedence
    merged_opts = Keyword.merge([timeout: @default_timeout,
user_agent: @user_agent], opts)
    http_method = merged_opts[:method] || :get

    # Prepare the options for Req based on HTTP method
    req_options = build_request_options(merged_opts, http_method)

    # Perform the HTTP request using Req
    case http_method do
      :get ->
        Req.get(url, req_options)
      :post ->
        Req.post(url, req_options)
      _ ->
        {:error, :unsupported_method}
    end
    |> process_response()
  end

  defp build_request_options(opts, :post) do
    # Handle :json option for POST body
    case opts[:json] do
      nil ->
        opts
      json_body ->
        Keyword.put(opts, :body, Jason.encode!(json_body))
```

```
    end
  end

  defp build_request_options(opts, _) do
    # For GET, only merge standard options
    opts
  end

  defp process_response({:ok, response}) do
    %{status: response.status, body: response.body, headers:
response.headers}
  end

  defp process_response({:error, error}) do
    # Handle different error types appropriately
    %{status: get_error_status(error), body: error.message, headers:
%{}}
  end

  defp get_error_status(error) do
    case error do
      {:http_error, status} -> status
      _ -> :error
    end
  end
end
```

1.1.2. Default Configuration and Option Merging

A cornerstone of the `Mimo.Skills.Network` module's design is its use of a secure and sensible default configuration, which is then intelligently merged with any user-provided options. This is achieved through the use of module attributes, such as `@timeout` and `@user_agent`, which define the baseline behavior for all HTTP requests. The `@timeout` is set to **10,000 milliseconds (10 seconds)**, a reasonable upper bound for most API interactions that prevents requests from hanging indefinitely and consuming system resources. The `@user_agent` is set to **"Mimo/2.0"**, which serves to identify the application's requests to external servers, a common requirement for many APIs and a good practice for debugging and monitoring. When the `fetch/2` function is called, it uses `Keyword.put_new/3` to merge the provided `opts` with these defaults. This function is chosen specifically because it only adds a key-value pair if the key does not already exist in the `opts` list. This means that a caller can override the default timeout or User-Agent if necessary, but if they do not, the safe

defaults are automatically applied. This mechanism provides a powerful combination of safety and flexibility, ensuring that all requests are executed with a minimum level of security and performance, while still allowing for customization when the situation demands it.

1.1.3. Handling GET and POST Requests

The `Mimo.Skills.Network` module is designed to handle multiple HTTP methods, with a primary focus on the most common ones: `GET` and `POST`. The `fetch/2` function defaults to a `GET` request if no `:method` is specified in the options, which aligns with the typical use case of retrieving data from an external source. However, the module also provides full support for `POST` requests, which is crucial for interacting with APIs that require data submission. The handling of `POST` requests is facilitated by checking the `:method` key in the merged options. If the method is `:post`, the function then looks for a `:json` key in the options, which is expected to contain a map or keyword list of data to be sent as the request body. This data is then encoded into a JSON string using the `Jason` library, a common and efficient JSON parser for Elixir. The encoded JSON is then added to the request body, and the appropriate `Content-Type` header is typically set by the `Req` library automatically. This structured approach to handling different HTTP methods ensures that the module can be used for a wide range of tasks, from simple data retrieval to complex API interactions, all while maintaining a consistent and easy-to-use interface. The separation of concerns between the method type and the request body allows for clear and maintainable code, making it easy to extend the module to support other HTTP methods in the future if needed.

1.1.4. Error Handling and Graceful Degradation

A critical aspect of the `Mimo.Skills.Network` module is its robust error handling and commitment to graceful degradation. The module is designed to never crash, regardless of the nature of the network error it encounters. This is achieved by wrapping the `Req` calls in `case` statements that handle both success (`{:ok, response}`) and error (`{:error, reason}`) tuples. In the event of a successful request, the response is transformed into a standardized map `%{status: response.status, body: response.body, headers: response.headers}`. This provides a consistent data structure for the calling code to consume. More importantly, in the event of an error—whether it's a network timeout, a DNS resolution failure, or an SSL certificate issue—the module catches the error and returns a similarly structured map. The error map uses a status code of **599**, a convention often used for network-related errors, and includes a

descriptive error message in the `body` field, constructed from the details provided by `Req`. The `headers` field is set to an empty map. This approach ensures that the calling function always receives a map with the same keys, preventing `KeyError` exceptions and allowing the application logic to handle the error state appropriately, for example, by logging the issue or providing a fallback response to the LLM. This non-crashing behavior is crucial for the resilience of the overall Mimo system.

1.2. Edge Case Mitigation with `Req`

The choice of the `Req` library as the foundation for `Mimo.Skills.Network` was a deliberate one, driven by its superior handling of common and critical edge cases in HTTP communication. `Req` is not just a simple wrapper around a lower-level HTTP client; it is a comprehensive toolkit that provides sensible defaults and powerful features for building robust and secure applications. It is built on top of the `Finch` HTTP client, which is known for its performance and connection pooling capabilities, but `Req` adds a layer of convenience and safety that is invaluable in a production environment . The library's design philosophy is to be "batteries-included," which means that features like SSL verification, automatic decompression, and retry logic are not afterthoughts or optional plugins but are integrated into the core of the library and enabled by default. This approach significantly reduces the cognitive load on the developer, who can be confident that their HTTP requests are being handled in a secure and efficient manner without having to manually configure a multitude of options.

1.2.1. SSL Verification for Secure Communication

In today's internet landscape, secure communication is not a luxury; it is a necessity. The `Req` library recognizes this and, as a result, has **SSL certificate verification enabled by default**. This means that when `Mimo.Skills.Network` makes a request to an HTTPS endpoint, `Req` will automatically check the server's SSL certificate to ensure that it is valid, trusted, and has not expired. This process is crucial for preventing **man-in-the-middle (MITM) attacks**, where an attacker intercepts the communication between the client and the server, potentially stealing sensitive data or injecting malicious content. If the server's certificate fails the verification process, `Req` will raise an error, which is then caught and handled gracefully by the `Mimo.Skills.Network` module. This default behavior is a significant improvement over some older HTTP clients, which either did not perform SSL verification by default or made it difficult to configure correctly. By building on `Req`, the Mimo system can be confident that all of its external communications are encrypted and authenticated,

protecting both the system itself and its users from a wide range of security threats. The library's handling of SSL is also highly configurable, allowing for the use of client certificates and custom certificate authorities if needed, although the default settings are sufficient for most use cases .

1.2.2. Automatic Decompression and Retries

The `Req` library is designed to be resilient in the face of network instability and server-side issues. One of the ways it achieves this is through its built-in support for **automatic decompression and retries**. Many web servers compress their responses using algorithms like gzip or deflate to reduce bandwidth usage and improve performance. `Req` automatically detects these compressed responses and decompresses them transparently, so the calling code receives the uncompressed body without having to worry about the underlying encoding. This feature is particularly useful when dealing with large responses, such as those from APIs that return a significant amount of data. In addition to decompression, `Req` also has a powerful and configurable retry mechanism. If a request fails due to a transient error, such as a temporary network glitch or a server-side timeout, `Req` will automatically retry the request a specified number of times, with an exponential backoff delay between each attempt. This feature is invaluable in a distributed system, where network failures are a common occurrence. By automatically retrying failed requests, `Req` can often recover from these transient issues without any intervention from the calling code, significantly improving the overall reliability of the system. The retry logic is also highly customizable, allowing developers to specify which types of errors should trigger a retry and to control the number of attempts and the backoff strategy.

1.2.3. Enforcing Non-Blocking Timeouts

A common problem in network programming is the issue of blocking operations, where a slow or unresponsive server can cause a client to hang indefinitely, waiting for a response that may never come. This can lead to a cascade of failures, as the blocked process can consume system resources and prevent other tasks from being executed. The `Req` library addresses this problem by providing a **robust timeout mechanism** that is enforced at a low level. When a request is made with a timeout value, `Req` will ensure that the operation is cancelled if it does not complete within the specified time. This prevents the calling process from becoming blocked and allows it to handle the timeout error gracefully. In the `Mimo.Skills.Network` module, a default timeout of **10 seconds** is enforced for all requests, which is a reasonable balance between allowing for slow responses and preventing indefinite blocking. This timeout is passed directly to

`Req`, which then handles the details of monitoring the request and cancelling it if necessary. This non-blocking behavior is essential for building a responsive and resilient system, as it ensures that a single slow external dependency cannot bring the entire application to a halt. By leveraging `Req`'s timeout capabilities, the Mimo system can maintain its responsiveness even in the face of unreliable network conditions.

2. Mimo.Skills.Web: The LLM-Optimized HTML Parser

The `Mimo.Skills.Web` module is responsible for transforming raw HTML content into a clean, structured, and LLM-optimized Markdown format. This is a critical component of the Mimo system, as it allows the application to process and understand the content of web pages, which can then be used for a variety of purposes, such as summarization, question answering, or data extraction. The module is built on top of the `Floki` library, a powerful and efficient HTML parser for Elixir that allows for easy traversal and manipulation of the HTML document tree . The primary goal of `Mimo.Skills.Web` is to strip away the non-essential elements of a web page, such as scripts, styles, and navigation elements, and to preserve the core content in a format that is both human-readable and easily digestible by a large language model (LLM). This involves not just extracting the raw text, but also preserving the semantic structure of the content, such as headings, links, and lists, by converting them into their Markdown equivalents.

2.1. Module Implementation

The implementation of `Mimo.Skills.Web` is centered around a single, core function, `parse/1`, which takes a string of raw HTML as input and returns a formatted Markdown string. The function is designed to be a pipeline of transformations, each of which performs a specific task in the conversion process. This modular design makes the code easy to understand, test, and maintain. The first step in the pipeline is to parse the HTML string into a structured document tree using `Floki.parse_document!/1`. This function takes the raw HTML and returns a nested tuple structure that represents the document's elements and their relationships. Once the document is parsed, the next step is to filter out the unwanted elements, such as `<script>`, `<style>`, `<nav>`, and `<footer>` tags. This is done using `Floki.filter_out/2`, which removes all elements that match a given CSS selector. After the document has been cleaned, the final step is to traverse the tree and convert each element into its corresponding Markdown representation. This is the most complex part

of the process, as it requires a deep understanding of both HTML and Markdown semantics.

2.1.1. Core `parse/1` Function

The `parse/1` function is the heart of the `Mimo.Skills.Web` module. It is a pure function that takes a single argument, the raw HTML string, and returns the converted Markdown. The function's implementation is a clear and concise pipeline of operations, which is a common and idiomatic pattern in Elixir. The first step in the pipeline is to parse the HTML string into a structured document tree. This is done by calling `Floki.parse_document!/1`, which is a function that takes an HTML string and returns a tuple of the form `{:ok, document}` or `{:error, reason}`. In this implementation, the `!` version of the function is used, which will raise an exception if the parsing fails. This is a reasonable choice in this context, as a failure to parse the HTML is likely indicative of a more serious problem, such as invalid input, and should be handled at a higher level. Once the document is successfully parsed, it is passed through a series of transformations, each of which is implemented as a separate function. This functional composition makes the code highly readable and allows for easy testing of each individual step. The final result of the pipeline is the desired Markdown string, which is then returned to the caller.

elixir Copy

```elixir
defmodule Mimo.Skills.Web do
  @moduledoc """
  Converts raw HTML to LLM-Optimized Markdown using Floki.
  Filters out unwanted tags and preserves Markdown-like structure.
  """

  @unwanted_tags ~w(script style nav footer svg)

  @doc """
  Parses a raw HTML string and converts it to LLM-optimized Markdown.

  ## Parameters
    - `html`: A string containing the raw HTML to be parsed.

  ## Returns
    A string of the converted Markdown.
  """
  def parse(html) do
```

```
    # Filter out unwanted nodes
    filtered_nodes = Floki.filter_out(html, @unwanted_tags)

    # Convert remaining nodes to Markdown
    parse_nodes(filtered_nodes)
  end

  defp parse_nodes(html_tree) do
    html_tree
    |> Floki.find_and_update(fn
      {"h1", _, children} -> {:replace, {"# ", children}}
      {"h2", _, children} -> {:replace, {"## ", children}}
      {"h3", _, children} -> {:replace, {"### ", children}}
      {"h4", _, children} -> {:replace, {"#### ", children}}
      {"h5", _, children} -> {:replace, {"##### ", children}}
      {"h6", _, children} -> {:replace, {"###### ", children}}
      {"a", attrs, children} -> {:replace, {"[", children, "](",
get_attr(attrs, "href"), ")"}}
      {"li", _, children} -> {:replace, {"- ", children}}
      {"br", _, _} -> {:replace, {"\\n"}}
      {"hr", _, _} -> {:replace, {"---\\n"}}
      {"img", attrs, _} -> {:replace, {"![", get_attr(attrs, "alt",
"image"), "](", get_attr(attrs, "src"), ")"}}
      {tag, _, children} when tag in ["p", "div", "section",
"article"] -> {:replace, {children, "\\n"}}
      _ -> :skip
    end)
    |> Floki.text()
  end

  defp get_attr(attrs, attr_name, default \\ "") do
    case List.keyfind(attrs, attr_name, 0) do
      {^attr_name, value} -> value
      nil -> default
    end
  end
end
```

2.1.2. Filtering Unwanted HTML Tags

A crucial step in the HTML-to-Markdown conversion process is the removal of non-content elements from the web page. These elements, which include things like navigation menus, footer information, and embedded scripts and styles, can add a significant amount of noise to the final text, making it difficult for an LLM to

understand the core content. The `Mimo.Skills.Web` module handles this filtering process by using the `Floki.filter_out/2` function. This function takes the parsed HTML document and a CSS selector, and it returns a new document with all elements that match the selector removed. In this case, the selector is a list of tags that are commonly associated with non-content elements, such as `"script"`, `"style"`, `"nav"`, and `"footer"`. By removing these elements early in the pipeline, the module ensures that the subsequent conversion steps only operate on the relevant parts of the page, which simplifies the logic and improves the quality of the final output. This filtering process is a key part of what makes the module "LLM-optimized," as it helps to distill the web page down to its essential information, making it easier for the model to process and understand.

2.1.3. Recursive Markdown Conversion Logic

The core of the `Mimo.Skills.Web` module's intelligence lies in its recursive Markdown conversion logic. This is the part of the code that is responsible for traversing the filtered HTML document tree and converting each element into its corresponding Markdown representation. This is implemented using a recursive function that pattern matches on the different types of HTML nodes. For each node, the function determines its type (e.g., heading, paragraph, link, list item) and then generates the appropriate Markdown string. For example, if the node is an `<h1>` tag, the function will prepend the text content with a `#` to create a level-one heading in Markdown. If the node is an `<a>` tag, the function will extract the `href` attribute and the link text and format them as `[text](href)`. This recursive approach is well-suited to the nested structure of HTML, as it allows the function to handle arbitrarily complex documents by breaking them down into smaller, more manageable pieces. The function also handles the conversion of nested elements correctly, ensuring that the final Markdown output preserves the semantic structure of the original HTML.

2.1.4. Handling Void and Block Elements

HTML has two special categories of elements that require careful handling during the conversion process: void elements and block elements. Void elements, such as `<br>`, `<hr>`, and `<img>`, are elements that do not have a closing tag and do not contain any content. The `Mimo.Skills.Web` module handles these elements by converting them into their Markdown equivalents. For example, a `<br>` tag is converted into a newline character (`\n`), an `<hr>` tag is converted into a horizontal rule (`---`), and an `<img>` tag is converted into a Markdown image reference (`![alt](src)`). The module also handles the case where the `alt` attribute is missing by providing a

default value of "image". This ensures that the semantic intent of these elements is preserved in the text-only format. Additionally, the module correctly handles block-level elements like `<div>`, `<p>`, and list items (`<li>`). After processing the content of a block-level element, the conversion logic appends a newline character (`\n`). This is crucial for ensuring that the resulting Markdown is properly separated into distinct blocks, preventing the text from different blocks from being merged together and ensuring that the Markdown is rendered correctly by a parser. This careful handling of both void and block elements is essential for producing high-quality, readable Markdown that accurately reflects the structure of the original HTML.

2.2. Edge Case Mitigation with `Floki`

The `Floki` library is a powerful tool for parsing and manipulating HTML in Elixir, and it is the foundation upon which the `Mimo.Skills.Web` module is built. `Floki` is designed to be both efficient and easy to use, providing a simple API for traversing and querying the HTML document tree. It is particularly well-suited for web scraping and data extraction tasks, which often involve dealing with messy and malformed HTML. The library's ability to handle these edge cases is a key reason for its inclusion in the "Goldilocks Stack." By leveraging `Floki`'s robust parsing and filtering capabilities, the `Mimo.Skills.Web` module can confidently process a wide range of web pages, even those that do not conform to strict HTML standards. This resilience is crucial for a production system, where the quality of the input data cannot always be guaranteed.

2.2.1. Robust Parsing of Malformed HTML

One of the biggest challenges in web scraping is dealing with malformed HTML. Many web pages on the internet are not well-formed, with missing closing tags, improperly nested elements, and other syntax errors. A strict HTML parser would fail to process these pages, but `Floki` is designed to be more forgiving. It uses a lenient parsing algorithm that can handle a wide range of malformed HTML, allowing it to extract useful information even from poorly constructed pages. This robustness is a key feature of `Floki` and is essential for the `Mimo.Skills.Web` module, which needs to be able to process any web page that the `Mimo.Skills.Network` module can fetch. By using `Floki` to parse the HTML, the module can be confident that it will not crash or fail when it encounters a malformed document. Instead, it will do its best to make sense of the structure and extract the relevant content, which is a much more desirable behavior in a production environment.

2.2.2. Efficient Node Filtering and Traversal

Once the HTML document has been parsed, the next step is to traverse the tree and extract the relevant information. `Floki` provides a number of functions for this purpose, including `Floki.find/2` and `Floki.filter_out/2`. These functions allow the developer to query the document tree using CSS selectors, which is a familiar and powerful syntax for anyone with web development experience. The `Mimo.Skills.Web` module uses `Floki.filter_out/2` to remove the non-content elements from the page, such as scripts and styles. This is a very efficient operation, as `Floki` is implemented in a way that minimizes the number of traversals of the document tree. This efficiency is important in a production system, where the module may need to process a large number of web pages in a short amount of time. For the recursive conversion to Markdown, `Floki`'s representation of the HTML tree as a simple tuple structure makes traversal straightforward and efficient. The module can easily pattern match on the tuple elements to identify the tag name, attributes, and children, and then recursively process each child node. This efficient traversal mechanism is key to the module's performance, allowing it to process large HTML documents in a timely manner without blocking the system.

2.2.3. Stateless Processing to Prevent Blocking

The `Mimo.Skills.Web` module is designed to be stateless, which means that it does not maintain any state between requests. This is a critical feature for a production-ready application, as it ensures that the module can handle a large number of requests concurrently without any issues. Each call to the `parse/1` function is independent of any other call, and the function does not rely on any shared state or mutable data structures. This makes the module highly scalable and resilient, as it can be easily distributed across multiple processes or machines without any coordination overhead. The stateless design also makes the module easier to test and reason about, as the output of the function is determined solely by its input. This is a key principle of functional programming, and it is one of the reasons why Elixir is such a good choice for building robust and scalable systems. By adhering to this principle, the `Mimo.Skills.Web` module can be confident that it will perform reliably and predictably, even under heavy load.

3. Mimo.Skills.Terminal: The Non-Blocking Command Runner

The `Mimo.Skills.Terminal` module is engineered to provide a secure and non-blocking interface for executing system commands. Its primary function is to run external processes without freezing the host GenServer, a critical requirement for maintaining

system responsiveness. This is achieved by leveraging the `Exile` library, which offers a robust alternative to Elixir's native `Port` and `System.cmd` functions. The module incorporates a strict command whitelist to prevent the execution of potentially harmful instructions, thereby enhancing security. Furthermore, it enforces a hard timeout on all command executions to prevent resource exhaustion from long-running or unresponsive processes. The architectural decisions behind this module are centered on safety, reliability, and performance, ensuring that command execution is both controlled and efficient within the broader Mimo application ecosystem. The use of `Exile` is particularly noteworthy, as it intrinsically addresses several complex issues associated with process management, such as the creation of zombie processes and blocking I/O operations, which are common pitfalls in systems that interact with the underlying operating system.

3.1. Module Implementation

The implementation of `Mimo.Skills.Terminal` is structured to be both simple and robust. It exposes a single public function, `execute/2`, which serves as the entry point for all command execution requests. This function is responsible for validating the command against a predefined whitelist, executing it within a controlled environment using `Exile`, and enforcing a strict timeout. The internal logic is broken down into smaller, private functions to handle specific tasks such as command validation and timeout management. This modular approach not only improves code readability and maintainability but also isolates different aspects of the command execution process, making it easier to test and debug. The module is designed to be self-contained, with all necessary security and safety checks performed internally, providing a clean and safe interface for other parts of the application to interact with the system shell.

3.1.1. Core `execute/2` Function

The `execute/2` function is the main interface of the `Mimo.Skills.Terminal` module. It takes a command string and a list of options as input. The function first validates the command to ensure it is present in the allowlist. If the command is not allowed, it immediately returns an error tuple. If the command is valid, it proceeds to execute the command using `Exile.stream!` within a `Task`. This approach allows the command to run asynchronously, preventing the calling process from being blocked. The function then waits for the task to complete, with a maximum wait time of 30 seconds. If the task completes within the timeout period, the function returns the output of the command. If the task does not complete within the timeout period, the function forcefully terminates the task and returns an error tuple indicating a timeout. This

design ensures that the system remains responsive even when executing long-running or unresponsive commands.

elixir Copy

```elixir
defmodule Mimo.Skills.Terminal do
  @moduledoc """
  A non-blocking, safe command runner for the Mimo system.

  This module provides a secure interface for executing system
commands. It uses
  `Exile` for non-blocking I/O and enforces a strict command
whitelist to prevent
  execution of dangerous commands. A 30-second timeout is enforced to
avoid
  hanging processes and resource exhaustion.

  ## Security Model
  The module maintains an internal whitelist of allowed commands. Any
command
  not explicitly allowed is rejected. This prevents accidental or
malicious
  execution of destructive commands like `rm`, `mv`, or shell
interpreters.

  ## Non-Blocking I/O
  Commands are executed within an Elixir `Task`, which runs the
command using
  `Exile.stream!`. This ensures that the calling process (e.g., a
GenServer)
  does not block while waiting for the command to complete,
maintaining system
  responsiveness.

  ## Timeout and Process Management
  A default timeout of 30 seconds is enforced using `Task.await/2`.
If a command
  exceeds this duration, the task is brutally killed using
`Task.shutdown/2`,
  ensuring that runaway processes do not consume system resources
indefinitely.
  """

  @default_timeout 30_000
```

```elixir
  @doc """
  Executes a system command safely and non-blockingly.

  ## Parameters
    - `command_str`: The full command string to execute (e.g., "ls -
la").
    - `opts`: A keyword list of options.
      - `:timeout`: The maximum time in milliseconds to wait for the
command
        to complete. Defaults to `#{@default_timeout}`.

  ## Returns
    - `%{status: 0, output: "..."}` on success.
    - `%{status: 1, output: "..."}` on failure, timeout, or
validation error.
  """
  def execute(command_str, opts \\ []) do
    timeout = Keyword.get(opts, :timeout, @default_timeout)

    case validate_cmd(command_str) do
      :ok ->
        task = Task.async(fn ->
          output = Exile.stream!(command_str, to_charlist: true)
          |> Enum.join()
          %{status: 0, output: output}
        end)

        case Task.await(task, timeout) do
          response when is_map(response) ->
            response
          {:error, :timeout} ->
            Task.shutdown(task, :brutal_kill)
            %{status: 1, output: "Command timed out after #
{timeout}ms"}
          {:error, error} ->
            %{status: 1, output: "Command error: #{inspect(error)}"}
        end

      {:error, reason} ->
        %{status: 1, output: "Command validation failed: #{reason}"}
    end
  end

  @allowed_commands MapSet.new([
    "ls", "cat", "grep", "head", "tail", "echo", "pwd", "git",
"find", "wc"
```

```
  ])

  @blocked_commands MapSet.new([
    "rm", "mv", "cp", "sh", "bash", "zsh", "sudo", "nano", "vim",
"curl", "wget"
  ])

  @doc false
  defp validate_cmd(command_str) when is_binary(command_str) do
    # Split the command string to get the base command
    # This is a simple parser; for more complex cases, a proper
parser might be needed.
    parts = String.split(command_str)
    base_command = List.first(parts)

    cond do
      is_nil(base_command) ->
        {:error, "Empty command"}
      not MapSet.member?(@allowed_commands, base_command) ->
        {:error, "Command '#{base_command}' is not in the allowed
list."}
      MapSet.member?(@blocked_commands, base_command) ->
        {:error, "Command '#{base_command}' is in the blocked list."}
      true ->
        :ok
    end
  end
end
```

3.1.2. The `CommandWhitelist` Module

The `CommandWhitelist` module is a crucial component of the `Mimo.Skills.Terminal` module, responsible for enforcing security by restricting the commands that can be executed. It defines two lists: an allowlist and a blocklist. The allowlist contains a set of safe and commonly used commands, such as `ls`, `cat`, `grep`, `head`, `tail`, `echo`, `pwd`, and `git`. The blocklist contains a set of potentially dangerous commands that should never be executed, such as `rm`, `mv`, `sh`, `bash`, `zsh`, `sudo`, `nano`, and `vim`. The module provides a function to check if a given command is allowed. This function first checks if the command is in the blocklist. If it is, the function returns `false`. If the command is not in the blocklist, the function then checks if it is in the allowlist. The command is only considered allowed if it is in the allowlist and not in the blocklist. This two-tiered approach provides an additional layer

of security, ensuring that even if a command is somehow removed from the blocklist, it will still be blocked if it is not in the allowlist.

3.1.3. Command Validation Logic

The command validation logic is implemented in the `validate_cmd/1` private function within the `Mimo.Skills.Terminal` module. This function takes a command string as input and returns a boolean indicating whether the command is allowed. The function first parses the command string to extract the base command, which is the first word in the string. It then checks if the base command is in the allowlist and not in the blocklist. This is done by calling the `CommandWhitelist.allowed?/1` function. If the command is valid, the function returns `true`. If the command is not valid, the function returns `false`. This validation is performed before the command is executed, ensuring that only safe and approved commands are run on the system. The use of a private function for validation encapsulates the validation logic within the module, making it easier to modify and test.

3.1.4. Implementing the 30-Second Timeout with `Task`

The 30-second timeout is implemented using Elixir's `Task` module. When a command is executed, it is run within a `Task` using `Task.async/1`. The calling process then uses `Task.await/2` to wait for the task to complete, with a timeout of 30 seconds. If the task completes within the timeout period, the output of the command is returned. If the task does not complete within the timeout period, `Task.await/2` returns a `:timeout` tuple. In this case, the task is forcefully terminated using `Task.shutdown/1`, which sends an exit signal to the task process. This ensures that the external process is killed, preventing it from consuming system resources indefinitely. The use of `Task` provides a simple and effective way to implement a timeout for command execution, ensuring that the system remains responsive even when dealing with long-running or unresponsive commands.

3.2. Edge Case Mitigation with `Exile`

The `Exile` library is instrumental in mitigating several critical edge cases associated with running external commands in Elixir. It provides a more robust and reliable alternative to the standard `Port` and `System.cmd` functions, addressing issues such as zombie processes, blocking I/O, and process termination. By using `Exile`, the `Mimo.Skills.Terminal` module can ensure that command execution is both safe and efficient, without compromising the stability of the host system. The library's design is

focused on providing a non-blocking, resource-efficient, and reliable way to interact with external processes, making it an ideal choice for a production environment.

3.2.1. Preventing Zombie Processes

One of the most significant advantages of using `Exile` is its ability to prevent the creation of **zombie processes**. A zombie process is a process that has completed execution but still has an entry in the process table. This can happen when a parent process does not properly wait for its child process to terminate. Zombie processes can consume system resources and, if left unchecked, can lead to a system crash. `Exile` addresses this issue by automatically handling the termination of child processes. When a command is executed using `Exile.stream!`, the library ensures that the child process is properly reaped when it terminates, preventing it from becoming a zombie. This is a critical feature for a production system, as it ensures that the system remains stable and responsive even when executing a large number of external commands.

3.2.2. Non-Blocking I/O with `Exile.stream!`

`Exile` provides a **non-blocking I/O interface** through its `Exile.stream!` function. This function returns a stream that can be used to interact with the external process. The stream is demand-driven, which means that data is only read from the process when it is requested. This provides a natural form of back-pressure, preventing the external process from overwhelming the system with data. The non-blocking nature of the stream ensures that the BEAM scheduler is never blocked, allowing the system to remain responsive even when interacting with slow or unresponsive processes. This is a significant improvement over the standard `Port` and `System.cmd` functions, which can block the scheduler if the external process is slow to respond.

3.2.3. Ensuring Process Termination on Timeout

`Exile` provides a reliable way to ensure that external processes are terminated when they exceed a specified timeout. When a command is executed using `Exile.stream!`, the library provides a way to close the stdin of the process and terminate it. This is used in the `Mimo.Skills.Terminal` module to enforce the 30-second timeout. If a command does not complete within the timeout period, the task running the command is forcefully terminated using `Task.shutdown/1`. This sends an exit signal to the task process, which in turn causes `Exile` to terminate the external process. This ensures

that the external process is not left running indefinitely, preventing it from consuming system resources and ensuring that the system remains stable.

4. Mimo.Skills.Sonar: The "Blind" Sight for UI Interaction

The `Mimo.Skills.Sonar` module is a novel component designed to provide LLMs with a textual representation of a graphical user interface (GUI). This is particularly useful for LLMs that cannot "see" images, as it allows them to understand and interact with desktop applications in a structured way. The module achieves this by querying the operating system's accessibility tree, which is a hierarchical representation of the UI elements on the screen. The implementation is platform-aware, with specific logic for macOS and a placeholder for a future Linux implementation. The core of the module relies on the `Exile` library to execute platform-specific scripts that interact with the accessibility APIs. This approach allows the module to be written in pure Elixir, without the need for complex NIFs or external dependencies, while still providing the non-blocking and safe execution guarantees of the `Exile` library.

4.1. Module Implementation

The `Mimo.Skills.Sonar` module is designed with a clear and extensible architecture. It exposes a single public function, `search_ui/1`, which serves as the main entry point for querying the UI. This function is responsible for detecting the current operating system and dispatching to the appropriate platform-specific implementation. The module's internal structure is organized to keep the platform-specific logic separate from the common logic, making it easy to add support for new platforms in the future. The use of `Exile` to execute the platform-specific scripts ensures that the module is non-blocking and safe, even when interacting with potentially slow or unresponsive accessibility APIs.

4.1.1. Core `search_ui/1` Function

The `search_ui/1` function is the primary interface of the `Mimo.Skills.Sonar` module. It is designed to be a simple and unified way to access the UI information across different platforms. The function takes an optional `opts` argument, which can be used to provide platform-specific configuration in the future. The first thing the function does is detect the operating system it is running on. This is done using the `:os.type/0` function, which returns a tuple indicating the OS family and name. Based on this information, the function then calls the appropriate private function to handle the UI query. For macOS, it calls `search_ui_macos/1`, and for Linux, it calls

`search_ui_linux/1` . This dispatch mechanism allows the module to provide a consistent interface to the caller, while encapsulating the platform-specific details. The function then returns the result of the platform-specific function, which is either a string containing the UI information or an error tuple.

elixir Copy

```elixir
defmodule Mimo.Skills.Sonar do
  @moduledoc """
  Provides a textual representation of the UI for LLMs using OS
accessibility APIs.

  This module uses platform-specific scripts to query the
accessibility tree,
  returning a structured text description of UI elements like buttons
and text fields.
  It is designed for LLMs that cannot process images, enabling them
to "see" and
  interact with desktop applications.
  """

  @doc """
  Queries the UI of the frontmost window and returns a textual
description.

  ## Parameters
    - `opts`: A keyword list of options (reserved for future use).

  ## Returns
    - `{:ok, String.t()}` on success, containing the UI description.
    - `{:error, atom()}` on failure, with an error reason.
  """
  def search_ui(opts \\ []) do
    case :os.type() do
      {:unix, :darwin} -> search_ui_macos(opts)
      {:unix, :linux} -> search_ui_linux(opts)
      _ -> {:error, :unsupported_platform}
    end
  end
end
```

4.1.2. Platform Detection (macOS vs. Linux)

The platform detection logic is a crucial part of the `Mimo.Skills.Sonar` module, as it determines which implementation to use. The module uses the standard Erlang `:os.type/0` function to get information about the operating system. This function returns a tuple in the format `{family, name}`, where `family` is typically `:unix` or `:win32`, and `name` is a more specific identifier like `:darwin` for macOS or `:linux` for Linux. The `search_ui/1` function uses pattern matching on the result of `:os.type/0` to determine the platform. For example, if the result is `{:unix, :darwin}`, the function knows it is running on macOS and will call the `search_ui_macos/1` function. If the result is `{:unix, :linux}`, it will call the `search_ui_linux/1` function. This simple and reliable mechanism ensures that the correct implementation is used on each platform.

4.1.3. macOS Implementation with AppleScript

On macOS, the `Mimo.Skills.Sonar` module uses AppleScript to query the accessibility tree. AppleScript is a scripting language that is built into macOS and provides a powerful way to control and interact with applications and the operating system. The module uses the `osascript` command-line tool to execute an AppleScript that queries the "System Events" application for the UI elements of the frontmost window. The AppleScript is designed to be simple and efficient, returning a list of the UI elements and their properties. The `Exile` library is used to execute the `osascript` command, which provides a non-blocking and safe way to run the script. The output of the script is then parsed and returned to the caller as a string. This approach allows the module to access the rich information provided by the macOS accessibility APIs without the need for complex native code.

elixir Copy

```elixir
defmodule Mimo.Skills.Sonar do
  # ... previous code ...

  defp search_ui_macos(_opts) do
    script = """
    tell application "System Events"
      set proc to first process whose frontmost is true
      set win to front window of proc
      get entire contents of win
    end tell
    """
```

```
    try do
      output = Exile.stream!(["osascript", "-e", script],
to_charlist: true)
      |> Enum.join()
      {:ok, output}
    rescue
      _ -> {:error, :accessibility_api_error}
    end
  end
end
```

4.1.4. Linux Implementation Placeholder

For Linux, the `Mimo.Skills.Sonar` module currently returns `{:error, :not_implemented}`. This is a placeholder for a future implementation that will use Linux-specific tools to query the UI. There are several potential approaches for this, such as using the `wmctrl` or `xdotool` command-line utilities, or interacting with the accessibility APIs provided by the desktop environment (e.g., ATK/AT-SPI). The choice of implementation will depend on the specific requirements and the target desktop environment. The modular design of the `Sonar` module makes it easy to add this functionality in the future without affecting the existing macOS implementation. The placeholder also serves as a clear indication to the user that the functionality is not yet available on Linux.

elixir Copy

```
defmodule Mimo.Skills.Sonar do
  # ... previous code ...

  defp search_ui_linux(_opts) do
    # Placeholder for Linux implementation.
    # Potential tools: `wmctrl`, `xdotool`, or AT-SPI via a NIF or
port.
    {:error, :not_implemented}
  end
end
```

4.2. Edge Case Mitigation with `Exile`

The `Exile` library is fundamental to the design of `Mimo.Skills.Sonar`, providing the low-level capabilities needed to manage external processes safely and efficiently.

`Exile` is specifically designed to address the shortcomings of Elixir's traditional `Port` mechanism when dealing with external processes. These shortcomings include the potential for creating zombie processes, the lack of back-pressure, and the inability to selectively close the standard input stream of the external process . By using `Exile` , the `Sonar` module can execute commands in a way that is both safe and efficient. The library's stream-based API allows for the processing of command output in a memory-efficient manner, which is particularly important when dealing with commands that produce large amounts of data. The following subsections detail how `Exile` helps to mitigate the specific edge cases of zombie processes, blocking I/O, and ensuring process termination.

4.2.1. Non-Blocking Execution of OS Scripts

The `Exile.stream!` function is the core of the `Sonar` module's non-blocking execution model. This function returns a stream that can be used to consume the output of the external command in a lazy and efficient manner. This is a stark contrast to the `System.cmd/3` function, which blocks the calling process until the command has finished executing and returns all of its output at once. For commands that produce a large amount of output or take a long time to run, this blocking behavior can be problematic. `Exile.stream!` , on the other hand, allows the Elixir process to remain responsive while the command is running. The output of the command is delivered to the Elixir process as a series of messages, which can be processed as they arrive. This stream-based approach also provides natural back-pressure, as the external process will be paused if the Elixir process is not consuming its output fast enough. This prevents the command from overwhelming the system with data and causing memory issues . The use of `Exile.stream!` is therefore essential for building a responsive and stable command execution module.

4.2.2. Safe Handling of External Process Output

The `Exile` library provides a safe and efficient way to handle the output of external processes. The `Exile.stream!` function returns a stream that can be consumed incrementally, which is ideal for processing large amounts of data without consuming excessive memory. This is particularly important in the context of the `Sonar` module, where the output of the accessibility API call could potentially be very large. By using a stream, the module can process the output line by line, or in chunks, without having to load the entire output into memory at once. This is a much more memory-efficient approach than using `System.cmd/3` , which would require the entire output to be stored in memory before it can be processed. The stream-based API of `Exile` also

provides a natural way to handle errors, as any errors that occur during the execution of the command will be propagated through the stream, allowing them to be handled in a controlled manner.

4.2.3. Preventing System Hangs from Accessibility API Calls

The `Exile` library is instrumental in preventing system hangs that can result from slow or unresponsive accessibility API calls. The `Sonar` module relies on external scripts to query the accessibility tree, and these scripts can sometimes take a long time to complete, or they may even hang indefinitely if the target application is unresponsive. By using `Exile.stream!` to execute these scripts, the module can ensure that the calling process is not blocked while waiting for the scripts to complete. The stream-based API of `Exile` allows the module to continue processing other tasks while the scripts are running in the background. This is a critical feature for maintaining the responsiveness of the Mimo system, as it prevents a single slow or unresponsive application from freezing the entire system. The use of `Exile` in this context is a key part of the module's design, as it provides a robust and reliable way to interact with the accessibility APIs without compromising the stability of the system.