Mimo Native Skills: Technical Design Document

1. The Modern `:httpc` Wrapper (Network Layer)

The foundational requirement for the Mimo Native Skills suite is a robust, secure, and user-friendly HTTP client. Given the strict constraint of zero external dependencies, the Erlang/OTP module `:httpc` is the only viable choice. However, its native API is often described as clunky and non-idiomatic for Elixir developers. This section details the design and implementation of `Mimo.Network.Fetch`, a wrapper that abstracts away the complexities of `:httpc` while providing a clean, modern, and secure interface. The wrapper will handle critical aspects such as secure SSL/TLS communication, automatic decompression of responses, graceful handling of timeouts and errors, and a flexible JSON decoding strategy that adapts to the runtime environment. The goal is to create a network layer that is not only functional but also resilient and easy to integrate into the broader Mimo system, ensuring that higher-level skills can perform web requests with confidence and simplicity.

1.1. Core Wrapper Design

The design of the `Mimo.Network.Fetch` module is centered around creating a single, primary function, `fetch_url/2`, which serves as the main entry point for all HTTP requests. This function will encapsulate the entire lifecycle of an HTTP request, from initial setup and configuration to response processing and error handling. The design prioritizes an Elixir-idiomatic API, which means leveraging keyword arguments for optional parameters, returning structured data in the form of maps, and using the `{:ok, result}` and `{:error, reason}` tuple pattern for handling outcomes. This approach provides a predictable and consistent interface for developers using the skill, abstracting the lower-level details of the Erlang `:httpc` module. The wrapper will be responsible for starting the necessary OTP applications (`:inets` and `:ssl`), managing request headers, and processing the raw response into a more usable format.

1.1.1. Elixir-Idiomatic API Design

The primary function, `fetch_url/2`, will be designed for simplicity and clarity. It will take a mandatory URL string as its first argument and an optional keyword list of options as its second. This design allows for a flexible and expressive API without requiring users to construct complex request tuples or manage multiple function arities. For example, a simple GET request can be made with

`Mimo.Network.Fetch.fetch_url("https://example.com")`, while a more complex request with custom headers and a timeout can be specified as `Mimo.Network.Fetch.fetch_url("https://api.example.com/data", headers: [{"Authorization", "Bearer token"}], timeout: 5000])`. This approach is consistent with many popular Elixir libraries and provides a gentle learning curve. The function will be responsible for translating these high-level options into the specific configuration required by the `:httpc:request/4` function, including setting up the request tuple, HTTP options, and request options. This abstraction shields the user from the intricacies of the Erlang API, such as the need to handle string formats (Erlang strings are character lists, while Elixir uses binaries) and the specific structure of the request and response tuples.

1.1.2. Return Value Structure

To provide a consistent and predictable interface, the `fetch_url/2` function will standardize the return value format. On a successful request, it will return an `:ok` tuple containing a map with the following keys: `:status`, `:body`, and `:headers`. The `:status` key will hold the HTTP status code as an integer (e.g., `200`). The `:body` key will contain the response body as a binary, which will have been decompressed if necessary. The `:headers` key will map to another map, where header names (as strings) are keys and header values are strings. This structure is intuitive and aligns with the response objects found in other modern HTTP clients. For instance, a successful response would look like `{:ok, %{status: 200, body: "...", headers: %{"content-type" => "text/html"}}}`. In case of an error, the function will return an `:error` tuple with a descriptive reason. This could be a network-level error (e.g., `:nxdomain`), an HTTP-level error (e.g., `:timeout`), or a parsing error. This clear separation between success and failure cases, along with a well-defined structure for the success data, simplifies error handling and data extraction for the calling code.

1.1.3. Handling Timeouts and Errors

Robust error handling is a cornerstone of the wrapper's design. The `:httpc` module provides several timeout-related options, which will be exposed through the `fetch_url/2` options list. Specifically, the `timeout` option will set the maximum time to wait for a response, while the `connect_timeout` option will set the maximum time to wait for the initial connection to be established . If a timeout occurs, the function will return `{:error, :timeout}`. Other potential errors, such as DNS resolution failures (`:nxdomain`), connection refusals (`:econnrefused`), or premature socket closures

(`:socket_closed_remotely`) , will also be caught and returned as part of the `{:error, reason}` tuple. The wrapper will also handle HTTP status codes that indicate an error (e.g., 4xx and 5xx). While the native `:httpc` behavior is to return the response body for these codes, the wrapper will treat them as successful requests from a network perspective, leaving it to the caller to interpret the status code. This is a deliberate design choice to provide maximum flexibility. The wrapper's role is to ensure the request was successfully sent and a response was received; the interpretation of that response's meaning is left to the higher-level skill logic.

1.2. Secure SSL/TLS Implementation

In today's internet landscape, secure communication over HTTPS is non-negotiable. The `:httpc` module relies on the Erlang `:ssl` application to handle TLS connections. A common and dangerous pitfall is to disable certificate verification to quickly overcome SSL errors, a practice often achieved by setting the `verify` option to `:verify_none` . This completely undermines the security of the connection, making the application vulnerable to man-in-the-middle attacks. The `Mimo.Network.Fetch` wrapper will enforce secure SSL/TLS practices by default, making it difficult for developers to inadvertently compromise security. It will provide a pre-configured, secure set of SSL options and will require explicit and deliberate action to override them, ensuring that all HTTPS connections are properly authenticated and encrypted.

1.2.1. Avoiding the `verify_none` Trap

The primary security goal is to prevent the use of `verify_none` . The wrapper will achieve this by providing a secure-by-default configuration for the `:ssl` options passed to `:httpc` . Instead of leaving SSL configuration entirely to the user, the wrapper will build a default set of options that enable peer verification. This means that by default, all HTTPS requests will validate the server's certificate against a list of trusted Certificate Authorities (CAs). If a developer encounters an SSL error, the correct approach is to diagnose the root cause (e.g., an expired certificate, a missing intermediate certificate, or a hostname mismatch) rather than simply disabling verification. The wrapper will guide users towards this best practice by making the insecure option non-obvious and requiring a conscious decision to bypass security. This design philosophy is critical for maintaining the integrity and trustworthiness of the Mimo system, especially when it may be handling sensitive data or interacting with external APIs.

1.2.2. Configuring `:ssl` Options for Peer Verification

To implement secure peer verification, the wrapper will construct a set of `:ssl` options based on established security guidelines . The core of this configuration will be the `verify: :verify_peer` option, which instructs the SSL client to perform certificate path validation. This process ensures that the server's certificate is signed by a trusted CA and that the certificate chain is valid. The wrapper will also configure the `depth` option to a reasonable value (e.g., 3) to limit the length of the certificate chain that will be accepted, preventing potential denial-of-service attacks involving excessively long chains. Furthermore, the `customize_hostname_check` option will be used with the `public_key:pkix_verify_hostname_match_fun(:https)` function to ensure that the server's certificate is valid for the hostname being requested, a crucial step in preventing attacks where a valid certificate for a different domain is presented . This comprehensive configuration ensures that all aspects of the server's identity are verified before a connection is considered secure.

1.2.3. Handling Certificate Authorities (CAs)

A critical component of peer verification is having a reliable set of trusted Certificate Authority (CA) certificates. The wrapper will leverage the `:public_key.cacerts_get/0` function, available in modern OTP versions, to obtain the system's default CA bundle . This function provides a curated and regularly updated list of trusted CAs, which is maintained as part of the Erlang/OTP distribution. By using this function, the wrapper avoids the need to manually manage a separate CA certificate file, which can be cumbersome and prone to becoming outdated. The `cacerts` option in the SSL configuration will be set to the result of `:public_key.cacerts_get()` , ensuring that the SSL client has a robust and current set of roots of trust for validating server certificates. This approach is both secure and maintainable, as it relies on the underlying OTP platform to keep the CA bundle up-to-date with the latest security standards and certificate revocations.

1.3. HTTP Request Handling

Beyond security, the wrapper must provide a convenient and flexible interface for constructing and sending HTTP requests. This includes support for different HTTP methods, the ability to set custom headers, and automatic handling of common server behaviors like redirects. The goal is to simplify the process of making HTTP requests, allowing developers to focus on the logic of their skills rather than the mechanics of the HTTP protocol. The wrapper will handle the translation of high-level Elixir data structures into the specific format required by the Erlang `:httpc` module, providing a seamless experience for the user.

1.3.1. Managing GET and POST Requests

The wrapper will primarily support the two most common HTTP methods: GET and POST. The `fetch_url/2` function will default to a GET request, which is suitable for retrieving data from a server. For sending data, a `method: :post` option will be supported, along with a `body` option to specify the request payload. The wrapper will handle the construction of the request tuple required by `:httpc`, which has the format `{Url, Headers, ContentType, Body}` for POST requests and `{Url, Headers}` for GET requests . The `ContentType` will be automatically set to a sensible default (e.g., `"application/octet-stream"`) if not provided by the user, but it can be overridden via the `content_type` option. This design provides a simple and intuitive way to perform both data retrieval and data submission operations, covering the majority of use cases for the Mimo skills.

1.3.2. Setting Custom Headers (e.g., User-Agent)

HTTP headers are essential for providing context about a request. The wrapper will allow users to specify custom headers through the `headers` option, which will accept a list of key-value tuples (e.g., `[{"User-Agent", "Mimo/1.0"}, {"Accept", "application/json"}]`). The wrapper will ensure these headers are correctly formatted as Erlang character lists, as required by `:httpc`. A default `User-Agent` header will be provided to identify the Mimo system, but this can be easily overridden. This flexibility is crucial for interacting with a wide range of web services, many of which require specific headers for authentication, content negotiation, or tracking. By providing a simple and direct way to set headers, the wrapper empowers developers to tailor their requests to the specific requirements of any API or web server they need to communicate with.

1.3.3. Automatic Redirect Following

Many web servers respond to requests with a 3xx status code to indicate that the requested resource has moved to a new location. The `:httpc` module provides an `autoredirect` option that, when set to `true`, will automatically follow these redirects and return the final response . The `Mimo.Network.Fetch` wrapper will enable this option by default. This is a sensible default for most use cases, as it simplifies the logic for the calling code, which would otherwise have to manually handle redirects by extracting the `Location` header from the response and making a new request. By automatically following redirects, the wrapper provides a more seamless experience, making a request to a URL that has been moved as simple as requesting the original

URL. For advanced use cases where redirect behavior needs to be controlled, this option could be exposed in the future, but for the initial implementation, automatic redirect following will be the standard behavior.

1.4. Response Processing

Once a response is received from the server, the wrapper's job is to process it into a clean and usable format. This involves parsing the headers, handling any content encoding (such as gzip compression), and extracting the response body. The raw response from `:httpc` is a tuple with a specific structure, and the wrapper will transform this into the standardized map format described earlier. This processing step is crucial for providing a consistent interface, regardless of the specifics of the server's response.

1.4.1. Parsing HTTP Response Headers into a Map

The raw response from `:httpc` includes a list of header tuples, where each tuple is of the form `{HeaderName, HeaderValue}` . Both the name and value are provided as Erlang character lists. The wrapper will iterate through this list and convert it into a more convenient Elixir map. The header names will be converted to strings (binaries) and downcased to ensure case-insensitivity, as per the HTTP specification. The header values will also be converted to strings. This transformation results in a map like `%{"content-type" => "text/html; charset=utf-8", "content-length" => "1234"}` , which is much easier to work with in Elixir. Accessing a specific header becomes as simple as `headers["content-type"]` , which is more intuitive than searching through a list of tuples.

1.4.2. Handling Content-Encoding (e.g., Gzip)

To save bandwidth, many web servers compress their responses using algorithms like gzip or deflate. The server indicates this by setting the `Content-Encoding` header in the response. The wrapper must be able to detect this header and decompress the response body accordingly. The Erlang `:zlib` module provides the necessary functions for this, such as `:zlib.gunzip/1` for gzip-compressed data . The wrapper will check the `Content-Encoding` header and, if it indicates compression, will pass the body through the appropriate decompression function before returning it to the caller. This process is transparent to the user of the wrapper; they will always receive the decompressed, readable body. This is a critical feature for a general-purpose HTTP client, as a significant portion of the web uses compression. Failure to handle this would result in the wrapper returning unreadable binary data for many websites.

1.4.3. Extracting the Response Body

The response body is provided as a binary within the raw response tuple from `:httpc` . After handling any necessary decompression, the wrapper will perform a final conversion to ensure the body is a standard Elixir binary. This step is mostly a formality, as the body is often already in the correct format, but it ensures consistency. The final body is then placed into the `:body` field of the returned map. This binary can then be used directly by the calling code, whether it's being passed to an HTML parser, a JSON decoder, or simply saved to a file. The wrapper's responsibility ends with providing the raw, decompressed response body; any further interpretation of its content (e.g., parsing HTML or JSON) is the responsibility of the higher-level skill that initiated the request.

1.5. JSON Decoding Strategy

A common task for an HTTP client is to fetch and parse JSON data from APIs. The Elixir ecosystem has excellent JSON libraries like Jason and Poison, but the zero-dependency constraint of the Mimo project prohibits their use. This necessitates a more creative approach. The strategy will be twofold: first, leverage the new native `:json` module introduced in OTP 27 if it is available, and second, provide a minimal, pure-Elixir fallback decoder for older OTP versions. This adaptive strategy ensures that the wrapper can handle JSON responses effectively across a range of Erlang/OTP environments without relying on external packages.

1.5.1. Leveraging the Native `:json` Module (OTP 27+)

The introduction of a native JSON module in Erlang/OTP 27 presents a significant opportunity for the Mimo Native Skills project to handle JSON decoding without external dependencies. This module, presumably named `:json` , would provide a standard, performant, and officially supported way to parse JSON strings directly within the Erlang VM. For the `fetch_json` skill, this means that on systems running OTP 27 or later, the implementation can be both simple and robust. The core advantage of using a native module is the elimination of the need to write and maintain a custom JSON parser, which is a complex and error-prone task, especially when dealing with the full JSON specification. A native implementation is likely to be highly optimized, written in C or Rust, and thoroughly tested by the OTP team, ensuring both speed and correctness. This aligns perfectly with the project's goal of creating a zero-dependency, production-ready skill set. The API for such a module would likely be straightforward, offering functions like `decode/1` to convert a JSON string into Erlang terms (which are seamlessly interoperable with Elixir data structures). This approach would not only simplify the codebase but also improve reliability and performance, making it the preferred method for JSON handling in the Mimo system when available.

The strategic importance of this feature cannot be overstated. JSON is the de facto standard for data interchange in modern web APIs, and the ability to parse it natively is a fundamental requirement for any tool that interacts with the web. Prior to OTP 27, the Elixir and Erlang ecosystems relied on a variety of third-party libraries, such as `Jason` and `Poison`, which, while excellent, introduce external dependencies that the Mimo project aims to avoid . The lack of a built-in solution has been a long-standing point of discussion in the community, with developers like José Valim and others debating the merits of including a JSON library in the core language . The consensus had been to keep the core lean, but the practical necessity of JSON parsing for web development has always been a strong counter-argument . The introduction of the `:json` module in OTP 27 resolves this debate at the Erlang level, providing a stable, universal foundation that Elixir can leverage. For the Mimo project, this means that the `fetch_json` skill can be implemented with a simple conditional check: if the `:json` module is available, use it; otherwise, fall back to a more basic solution. This graceful degradation ensures that the skill remains functional across different OTP versions while taking full advantage of the latest platform features when they are present.

1.5.2. Runtime Detection of the `:json` Module

To ensure the `Mimo.Network.Fetch` module remains compatible across different versions of Erlang/OTP, a runtime detection mechanism for the native `:json` module is essential. This strategy allows the code to dynamically determine the best available method for JSON decoding at the moment of execution. The implementation would involve using Elixir's `Code.ensure_loaded?/1` function to check for the existence of the `:json` module. If this function returns `true`, the code can confidently call `:json.decode/1` to parse the response body. This approach provides a clean and efficient way to leverage the new OTP 27 feature without breaking functionality on older systems. The logic would be encapsulated within a private function, perhaps named `decode_json/1`, which abstracts away the underlying implementation details.

This function would first attempt to use the native module and, if it's not available, proceed to a fallback method. This design pattern, often referred to as feature detection, is superior to version-based checks (e.g., checking the OTP version string) because it directly verifies the availability of the required functionality rather than making assumptions based on version numbers. This makes the code more resilient to future changes and potential backports of the `:json` module to earlier OTP versions.

The practical implementation of this detection mechanism would be straightforward. The `fetch_json` function, after receiving the HTTP response body, would pass it to this internal `decode_json/1` function. This function would contain the conditional logic. For example:

elixir 复制

```elixir
defp decode_json(body) do
  if Code.ensure_loaded?(:json) do
    case :json.decode(body) do
      {:ok, data} -> {:ok, data}
      {:error, reason} -> {:error, "Native JSON decode failed: #
{inspect(reason)}"}
    end
  else
    # Fallback to minimal decoder
    MinimalJson.decode(body)
  end
end
```

This structure ensures that the primary, most efficient path is taken when possible, while still providing a functional alternative for older environments. The use of `Code.ensure_loaded?/1` is a standard and reliable method for this type of dynamic module checking in Elixir. It avoids the need for complex build-time configurations or conditional compilation, keeping the codebase simple and maintainable. The error handling within the native decode branch is also crucial, as it provides clear feedback if the native decoder itself fails, which could happen with malformed JSON input. This layered approach to error handling—first checking for module availability, then handling potential decoding errors—results in a robust and user-friendly API that gracefully handles a variety of runtime conditions. This strategy is a cornerstone of building resilient, zero-dependency libraries that can operate across a wide range of deployment environments.

1.5.3. Fallback: Minimal Pure-Elixir JSON Decoder

In environments where the native `:json` module is not available (i.e., OTP versions prior to 27), the Mimo Native Skills project must provide a fallback mechanism for JSON decoding. Given the strict constraint of zero external dependencies, this fallback must be a minimal, pure-Elixir JSON decoder implemented using only the Elixir standard library and Erlang/OTP modules. The research conducted, including discussions on the Elixir Forum, confirms that there is no built-in JSON parsing capability in Elixir or older versions of Erlang, making a custom implementation necessary . However, implementing a full-featured JSON parser that adheres to the entire RFC 8259 specification is a significant undertaking and is beyond the scope of this project. Therefore, the fallback decoder will be intentionally minimal, designed to handle only the most common and simple JSON structures typically found in standard API responses. This includes basic objects (maps), arrays, strings, numbers, booleans, and null. The parser will not aim for perfect compliance or high performance but will focus on being "good enough" for the majority of use cases encountered by the LLM-driven skills.

The implementation of this minimal decoder will rely heavily on Elixir's powerful pattern matching and recursive functions. The process would involve several stages. First, a tokenizer would break the input JSON string into a list of tokens (e.g., `{` , `}` , `[` , `]` , `:` , `,` , string literals, number literals, `true` , `false` , `null`). This can be achieved using a combination of string manipulation and regular expressions. Second, a parser would consume this list of tokens and build the corresponding Elixir data structure. For example, upon encountering a `{` token, the parser would recursively parse key-value pairs until a matching `}` token is found, constructing a map in the process. Similarly, a `[` token would trigger the parsing of a list of values until the corresponding `]` is reached. This recursive descent approach is well-suited for the hierarchical nature of JSON. While this method is not as efficient as a parser generated by a tool like Leex or Yecc, it is entirely feasible to implement within the project's constraints and will provide the necessary functionality for simple JSON responses. The key is to clearly document the limitations of this minimal parser, such as its inability to handle complex number formats or deeply nested structures beyond a certain depth, setting appropriate user expectations.

2. Zero-Dependency HTML-to-Markdown (The "Readability" Layer)

The conversion of raw HTML content into a clean, readable Markdown format is a critical component of the Mimo Native Skills suite, designed to provide Large Language Models (LLMs) with structured, text-based data from web sources. This process must be achieved without relying on external Hex packages such as `Floki` or `Meeseeks`, adhering strictly to the project's zero-dependency constraint. The primary challenge lies in creating a robust yet lightweight parser that can effectively handle the complexities and inconsistencies of real-world HTML. The research conducted reveals two primary viable strategies for this task: a heuristic, regex-based approach and a more structured approach leveraging Erlang's built-in `:xmerl` XML parsing library. Each strategy presents a distinct set of trade-offs in terms of implementation complexity, performance, and robustness. The final implementation must prioritize producing "LLM-readable" output, meaning the Markdown should be well-structured and free of HTML artifacts, even if it does not achieve perfect 100% fidelity to every HTML specification nuance. This section details the design philosophy, core logic, and advanced considerations for both parsing strategies, providing a comprehensive roadmap for building a production-ready HTML-to-Markdown converter.

2.1. Parser Design Philosophy

The design of the HTML-to-Markdown parser is guided by the project's core constraints and the specific use case of preparing web content for LLM consumption. The fundamental goal is not to create a universal, standards-compliant HTML parser, which is a notoriously difficult task, but rather to build a pragmatic and efficient tool that extracts the primary textual content and represents it in a clean Markdown format. This pragmatic approach allows for certain simplifications and assumptions that would be unacceptable in a general-purpose library but are perfectly suited for this application. The parser's design must balance the need for robustness against malformed HTML with the requirement for high performance and minimal resource consumption, all while operating within the strict confines of the Elixir standard library and Erlang/OTP modules. The following subsections explore the key philosophical tenets that will shape the implementation, including the choice between regex-based and structured parsing, the definition of "LLM-readable" output, and the strategies for gracefully handling the wide variety of HTML structures found on the web.

2.1.1. Heuristic and Regex-Based Approach

A heuristic, regex-based approach offers a direct and often surprisingly effective method for converting HTML to Markdown, especially when the goal is "LLM-readable" output rather than perfect semantic fidelity. This strategy involves a series of sequential string replacements using Elixir's powerful `Regex` module to identify HTML tags and replace them with their corresponding Markdown syntax. For example, a pattern like `~r/<h1>(.*?)<\/h1>/i` could be used to find level-1 headers and replace them with `# \1`, or `~r/<a\s+href="(.*?)">(.*?)<\/a>/i` could convert hyperlinks to the `[text](url)` format. This method's primary advantage is its simplicity and speed; regex operations are highly optimized and can process large chunks of text very quickly without the overhead of building a complex document tree. This approach is well-suited for handling common, well-formed HTML structures and can be implemented with minimal code. However, its major drawback is its fragility when faced with nested tags, malformed HTML, or complex attributes. For instance, a regex for `<strong>` tags might incorrectly match content within a `<script>` or `<style>` block, or fail if the tag attributes are in an unexpected order. The research into existing solutions, such as the JavaScript-based converter on Scito.ch, confirms that while a regex-based system can be built, it requires careful ordering of operations and extensive testing to handle a reasonable subset of HTML . For the Mimo project, this approach would be implemented as a series of well-defined, ordered `String.replace/4` calls within the `Mimo.Web.HtmlParser` module, with each regex pattern carefully crafted and tested to minimize false positives and handle common variations in HTML syntax.

2.1.2. Target Output: LLM-Readable Markdown

The ultimate success metric for the `Mimo.Web.HtmlParser` is not its adherence to the CommonMark or GitHub Flavored Markdown specifications, but rather its ability to produce "LLM-readable" output. This concept defines a specific set of characteristics that make the converted text most suitable for consumption by a Large Language Model. First and foremost, the output must be clean and free of any HTML artifacts, such as stray tags or attributes, which could confuse the model or be interpreted as part of the text. The structure of the content should be preserved and clearly represented using standard Markdown conventions. This means that headings should be denoted with `#`, lists with `-` or `1.`, and code blocks with triple backticks. Hyperlinks should be converted to the `[text](url)` format, ensuring that both the anchor text and the destination URL are retained, as this information is often crucial for the LLM's understanding of the content's context and sources. Whitespace should be normalized to be consistent and readable, collapsing excessive newlines or spaces that are often artifacts of HTML formatting. The parser should also prioritize the extraction of the main content of a page, potentially stripping out non-content elements like navigation menus, headers, footers, and advertisements, which are irrelevant to the LLM's task. This focus on readability over strict specification compliance allows for pragmatic decisions in the conversion logic. For example, if a perfect conversion of a complex HTML table is overly burdensome, a simpler text-based representation that preserves the tabular data might be sufficient for the LLM to understand the information.

2.1.3. Handling Malformed HTML

A critical requirement for any practical HTML parser is its ability to handle malformed or non-standard HTML, which is ubiquitous on the web. The parser must be resilient and fail gracefully, rather than crashing or producing garbled output when it encounters unexpected syntax. The two primary parsing strategies offer different levels of resilience. A regex-based approach is inherently brittle; a single unexpected character or malformed tag can cause a regex pattern to fail, leading to unprocessed HTML in the output. To mitigate this, the regex-based implementation would need to include a "cleanup" phase at the end, using a broad pattern like `~r/<[^>]*>/` to strip any remaining HTML tags that were not caught by the specific conversion patterns. This ensures that even if the primary logic fails, the output is at least clean text. The `:xmerl` -based approach, on the other hand, is generally more robust. As a proper parser, it is designed to handle a wider range of syntax errors and can often parse documents that are not perfectly well-formed. However, it is not infallible, and severely broken HTML can still cause parsing errors. In a production implementation, the `:xmerl` parsing call would be wrapped in a `try-rescue` block to catch any potential `:error` exceptions. If parsing fails, the system could fall back to a simpler text-extraction method, such as the regex-based cleanup, to ensure that the system remains operational and still provides some usable output to the LLM. This layered approach, combining a primary robust parser with a secondary fallback, provides the best strategy for handling the unpredictable nature of web content.

2.2. Core Conversion Logic

The core of the `Mimo.Web.HtmlParser` module lies in its conversion logic, which translates the parsed HTML structure into valid Markdown syntax. The implementation of this logic will differ significantly depending on whether a regex-based or an `:xmerl` -based approach is chosen. The regex-based strategy relies on a carefully orchestrated sequence of string replacements, while the `:xmerl` strategy involves recursively traversing a document tree. Both methods must address the same fundamental tasks: identifying specific HTML elements and transforming them into their Markdown counterparts. This includes handling inline elements like links and text formatting, as well as block-level elements such as headers, lists, and paragraphs. The following subsections detail the specific algorithms and considerations for implementing this conversion logic in both the regex and structured parsing paradigms, providing concrete examples of how to handle the most common HTML elements.

2.2.1. Sequential Regex Substitutions

In a regex-based implementation, the conversion logic is a linear pipeline of `String.replace/4` functions, each targeting a specific HTML tag or pattern. The order of these substitutions is critically important to ensure correct nesting and avoid conflicts. For example, converting `<strong>` tags to `**` should generally happen before converting `<a>` tags to `[text](url)`, as the link text itself might contain bold formatting. A typical sequence might start by handling block-level elements like headers (`<h1>` to `<h6>`), paragraphs (`<p>`), and horizontal rules (`<hr>`). This is followed by list elements (`<ul>` , `<ol>` , `<li>`), and then inline elements such as emphasis (`<em>` , `<i>`), strong text (`<strong>` , `<b>`), and code (`<code>`). Finally, more complex elements like links (`<a>`) and images (`<img>`) are processed. Each step uses a distinct regex pattern. For instance, to convert `<h1>` tags, the pattern `~r/<h1\b[^>]*>(.*?)<\/h1>/is` could be used, with the replacement string `# \1\n\n`. The `s` flag allows the dot (`.`) to match newlines, which is essential for handling multi-line content within a tag. The `i` flag makes the match case-insensitive. This sequential approach is straightforward to implement and debug, as each step can be tested independently. However, as noted, its main weakness is its inability to handle complex nesting or malformed HTML gracefully, making it a pragmatic but imperfect solution.

2.2.2. Preserving Links and Link Text

A crucial function of the parser is to correctly identify and convert HTML hyperlinks (`<a>` tags) into Markdown format (`[text](url)`), as links are a primary source of context and information on the web. In a regex-based approach, this is achieved with a pattern designed to capture both the URL from the `href` attribute and the anchor text. A suitable pattern would be `~r/<a\s+href="([^"]*)"(?:\s+title="[^"]*")?[^>]*>(.*?)<\/a>/is`. This pattern captures the URL into the first capture group (`\1`) and the link text into the second capture group (`\2`). The replacement string would then be `[\2](\1)`. The pattern also includes an optional non-capturing group `(?:\s+title="[^"]*")?` to account for the `title` attribute, which should be ignored in the conversion. In the `:xmerl` approach, the process is more structured. The `handle_link/1` function would receive an `:xmlElement` record representing the `<a>` tag. It would first extract the attributes list and find the tuple corresponding to the `href` attribute to get the URL. Then, it would process the element's content (its child nodes) by recursively calling the traversal function, which will return the plain text of the anchor. Finally, it would concatenate these two pieces of information into the Markdown format. This structured method is less prone to errors with complex anchor text or attributes in different orders, making it the more reliable of the two approaches for this critical task.

2.2.3. Converting Headers (H1-H6)

HTML headers, from `<h1>` to `<h6>`, are fundamental to the structure of a document and must be accurately converted to their Markdown equivalents (`#` to `######`). In a regex-based system, this requires six distinct patterns, one for each header level. For example, the pattern for an `<h1>` tag would be `~r/<h1\b[^>]*>(.*?)<\/h1>/is`, with the replacement `# \1\n\n`. The pattern for `<h2>` would be `~r/<h2\b[^>]*>(.*?)<\/h2>/is`, with the replacement `## \1\n\n`, and so on. The `\b` in the pattern ensures that tags like `<h11>` are not mistakenly matched when searching for `<h1>`. The order of these replacements is not critical as they target distinct tags. In the `:xmerl` implementation, a single handler function, `handle_header/2`, could be used. This function would take the `:xmlElement` record and the header level (an integer from 1 to 6) as arguments. It would generate the corresponding number of `#` characters and then recursively process the header's child nodes to get the text content. For example, upon encountering an `:xmlElement` with the name `:h3`, the main traversal function would call `handle_header(element, 3)`. This approach is more DRY (Don't Repeat Yourself) and extensible, as the logic for all header levels is consolidated into a single function. Both methods effectively achieve the conversion, but the `:xmerl` approach again demonstrates superior structure and maintainability.

2.3. Handling Common HTML Elements

Beyond the core conversion logic for headers and links, a comprehensive HTML-to-Markdown parser must be able to handle a wide variety of common HTML elements that contribute to the structure and formatting of web content. These include elements for text formatting (bold, italic, code), lists (ordered and unordered), paragraphs, line breaks, and blockquotes. The successful conversion of these elements is essential for producing Markdown that is not only readable by an LLM but also accurately reflects the intended formatting and structure of the original HTML. The implementation details for handling these elements will vary between the regex-based and `:xmerl`-based approaches, but the fundamental goal remains the same: to map each HTML tag to its corresponding Markdown syntax in a reliable and efficient manner. The following subsections provide specific implementation strategies for these common elements, offering concrete examples for both parsing methodologies.

2.3.1. Text Formatting (Bold, Italic, Code)

Inline text formatting elements like `<strong>` / `<b>` (bold), `<em>` / `<i>` (italic), and `<code>` (inline code) are common and must be converted to their Markdown equivalents (`**text**` , `*text*` , and `` `text` ``). In a regex-based approach, this involves three separate substitution patterns. For bold text, a pattern like `~r/<(?:strong|b)\b[^>]*>(.*?)<\/(?:strong|b)>/is` can be used, with the replacement `**\1**` . This pattern uses a non-capturing group `(?:strong|b)` to match either tag. Similarly, for italic text, `~r/<(?:em|i)\b[^>]*>(.*?)<\/(?:em|i)>/is` would be replaced with `*\1*` . For inline code, `~r/<code\b[^>]*>(.*?)<\/code>/is` would be replaced with `` `\1` `` . The order of these operations is important; for instance, code conversion should happen before bold and italic to avoid conflicts if a code snippet contains asterisks. In the `:xmerl` approach, dedicated handler functions like `handle_strong/1` , `handle_em/1` , and `handle_code/1` would be created. Each function would receive the element's child nodes, process them to get the plain text, and then wrap the result in the appropriate Markdown delimiters. This structured approach is cleaner and avoids the potential for regex conflicts, making it easier to manage and extend with additional formatting tags.

2.3.2. Lists (Ordered and Unordered)

Converting HTML lists (`<ul>` , `<ol>` , `<li>`) to Markdown is more complex due to the need to handle nesting and list item markers. In a regex-based system, this is one of the most challenging aspects. A simple pattern for `<li>` tags is insufficient, as it doesn't account for the parent list type (`<ul>` or `<ol>`) or the nesting level. A more sophisticated approach would be required, potentially involving multiple passes: one to identify and mark list blocks, and another to process the items within each block, keeping track of the nesting depth to add the correct number of leading spaces. This can become quite intricate and prone to error. The `:xmerl` approach handles this much more elegantly. A `handle_list/1` function would process a `<ul>` or `<ol>` element. It would iterate through its children, finding the `<li>` items. For each item, it would recursively process its content. The function would keep track of the nesting level and whether the list is ordered or unordered, and prepend the correct marker (`-` or a number followed by a `.`) with the appropriate indentation. This recursive, state-aware processing is a natural fit for the tree-traversal model and can accurately reproduce the structure of nested lists in Markdown.

2.3.3. Paragraphs and Line Breaks

HTML paragraphs (`<p>`) and line breaks (`<br>`) are fundamental to text layout. In Markdown, paragraphs are separated by blank lines, and line breaks within a paragraph are created by ending a line with two or more spaces. In a regex-based system, `~r/<p\b[^>]*>(.*?)<\/p>/is` can be replaced with `\1\n\n` to handle paragraphs. Line breaks, `~r/<br\s*\/?>/i` , can be replaced with ` \n` (a space followed by a newline) to create a hard line break in Markdown. The challenge is to ensure these operations are performed in the correct order and do not interfere with other block-level elements. For example, the content of a header should not be wrapped in paragraph tags. The `:xmerl` approach handles this by treating `<p>` and `<br>` as distinct elements. A `handle_paragraph/1` function would process the paragraph's content and ensure it is separated from subsequent content by a blank line. A `handle_br/0` function would simply return the Markdown for a line break. This separation of concerns makes the logic clearer and reduces the chance of unintended interactions between different elements.

2.3.4. Blockquotes and Horizontal Rules

Blockquotes (`<blockquote>`) and horizontal rules (`<hr>`) are other common block-level elements. In Markdown, blockquotes are denoted by a `>` at the beginning of each line, and horizontal rules are typically `---` or `***` . In a regex-based system, a pattern like `~r/<blockquote\b[^>]*>(.*?)<\/blockquote>/is` would need to not only capture the content but also prepend `>` to every line of the captured text. This requires a more complex replacement function, possibly using `String.replace/3` with a function as the replacement to process each line. Horizontal rules are simpler: `~r/<hr\s*\/?>/i` can be replaced with `\n\n---\n\n` . The `:xmerl` approach is again more structured. A `handle_blockquote/1` function would process the content of the blockquote and prepend the `>` character to each line of the resulting Markdown. A `handle_hr/0` function would simply return the string for a horizontal rule. This methodical, element-by-element processing is a key advantage of the structured parsing approach, leading to more maintainable and reliable code.

2.4. Advanced Parsing Considerations

To elevate the HTML-to-Markdown converter from a basic tool to a production-ready component, several advanced parsing considerations must be addressed. These go beyond the simple conversion of standard HTML tags and involve making intelligent decisions about the content to include or exclude, handling special characters and encodings, and dealing with more complex structures like tables. These considerations are crucial for ensuring the final Markdown output is not only syntactically correct but also clean, relevant, and truly "LLM-readable." The ability to filter out non-content elements, decode HTML entities, and provide at least a basic representation of tables will significantly enhance the utility and robustness of the parser. The following subsections delve into these advanced topics, outlining strategies for their implementation within the zero-dependency constraint.

2.4.1. Stripping Non-Content Elements (Script, Style, Nav)

A critical step in producing clean, readable Markdown is the removal of non-content elements that are part of the web page's structure but irrelevant to the main textual content. These elements include `<script>` blocks, `<style>` sections, navigation menus (`<nav>`), headers, footers, and advertisements. Including the text from these sections would introduce noise and clutter into the Markdown, potentially confusing the LLM. In a regex-based approach, this can be achieved with a "strip" phase before the main conversion. A broad pattern like `~r/<(script|style|nav|header|footer|aside)\b[^>]*>.*?<\/\1>/is` can be used to find and remove these blocks. The `.*?` uses a non-greedy quantifier to match everything up to the corresponding closing tag. This pre-processing step cleans the HTML before the conversion logic is applied. In the `:xmerl` approach, this is handled more elegantly during the tree traversal. The main `convert_node/1` function can be configured to simply ignore nodes with specific tag names. When it encounters an `:xmlElement` with a name like `:script` or `:nav` , it can return an empty string, effectively pruning that entire branch from the output. This is a more efficient and targeted approach, as it avoids the need for a separate pre-processing pass and integrates the filtering logic directly into the parsing process. The `html2markdown` library on Hex, for example, provides configurable options for `non_content_tags` and `navigation_classes` , demonstrating the importance of this feature in a content extraction context .

2.4.2. Decoding HTML Entities

HTML documents frequently contain character entity references (e.g., `&amp;` , `&lt;` , `&gt;` , `&nbsp;`) to represent special characters. These entities must be decoded into their actual character equivalents (`&` , `<` , `>` , a non-breaking space) during the conversion process to ensure the Markdown is readable. A regex-based approach would require a final substitution pass after all other conversions are complete. A mapping of common entities to their characters would be needed, and a pattern like `~r/&(amp|lt|gt|nbsp);/` would be replaced using a function that looks up the corresponding character in the map. This can be tedious and may not cover all possible named or numeric entities. The `:xmerl` library, however, provides built-in support for handling entities. When `:xmerl_scan.string/1` parses the document, it automatically decodes these entities into their character representations within the `:xmlText` nodes. This means that when the traversal function accesses the text content of a node, it is already in its decoded form. This is a significant advantage of the `:xmerl` approach, as it offloads the complexity of entity decoding to a robust, well-tested library, simplifying the conversion logic and ensuring more complete and accurate decoding.

2.4.3. Handling HTML Tables

HTML tables (`<table>` , `<thead>` , `<tbody>` , `<tr>` , `<th>` , `<td>`) are a complex structure to convert to Markdown. Markdown has its own syntax for tables, but it is not as expressive as HTML and can be challenging to generate programmatically, especially when dealing with rowspans, colspans, or complex formatting within cells. A full-featured table converter is likely out of scope for a zero-dependency, lightweight parser. However, a basic representation is desirable. In a regex-based system, a full table conversion would be extremely difficult and fragile. A more feasible fallback would be to simply strip the table tags and leave the raw text, perhaps with some basic formatting to indicate rows and columns, but this would lose the tabular structure. The `:xmerl` approach offers a better path to a more structured solution. A `handle_table/1` function could traverse the table's rows and cells, extract the plain text content from each cell, and then reconstruct it as a GitHub Flavored Markdown table. This would involve calculating the maximum width of each column for formatting and constructing the header and separator rows. While still a non-trivial task, the structured nature of the `:xmerl` tree makes it possible to systematically process the table's components. The `html2markdown` library on Hex explicitly lists "Table Conversion" as a feature, indicating that a robust solution is achievable, though it likely represents a significant implementation effort . For the initial Mimo implementation, a simpler approach of just extracting the text content from tables might be a pragmatic first step, with a full Markdown table conversion as a future enhancement.

3. Surgical Text Editing (The "Cursor" Layer)

The `edit_file` skill requires a robust and intelligent text replacement mechanism that can gracefully handle the imprecision often found in LLM-generated code blocks. A naive, exact string match is too brittle, as it will fail if the LLM introduces minor variations in whitespace or indentation. The `Mimo.Text.Editor` module will provide a `apply_patch/3` function that implements a whitespace-resilient "find and replace" logic. This function will be capable of locating a `search_block` within a larger `content` string, even if the whitespace does not match exactly, and replacing it with a `replace_block`. A critical feature of this function is its ability to detect ambiguity—that is, when the `search_block` matches multiple locations in the `content`—and report this back to the caller, preventing unintended and potentially destructive changes. This section details the design of this surgical text editing tool, covering its core function, resilient matching algorithms, and strategies for handling both unique and ambiguous matches.

3.1. Core `apply_patch/3` Function

The `apply_patch/3` function is the primary interface for the text editing skill. It is designed to be a pure function, taking the original content and the search-and-replace parameters as input and returning a new, modified content string along with a status report. This functional design makes it easy to test and reason about. The function's behavior is defined by its parameters and its structured return values, which clearly communicate the outcome of the operation, whether it was a success, a failure to find a match, or the discovery of an ambiguous match.

3.1.1. Function Signature and Parameters

The function signature will be `apply_patch(content :: String.t(), search_block :: String.t(), replace_block :: String.t(), opts \\ []) :: result`. The parameters are as follows:

- **`content`**: The entire text content of the file to be edited, provided as a string.
- **`search_block`**: The block of text to search for within the `content`. This is the "old" text that the LLM wants to replace.
- **`replace_block`**: The block of text that will replace the `search_block`. This is the "new" text provided by the LLM.
- **`opts`**: An optional keyword list of options to control the behavior of the function. A key option here would be `:occurrence`, which could be an integer to specify which match to replace (e.g., `1` for the first, `2` for the second) or the atom `:all` to replace all occurrences. The default would be `1`.

This signature provides a clear and flexible API. By making the `opts` list optional with a sensible default, the most common use case (replacing the first occurrence) is simple to invoke, while still allowing for more complex operations like replacing all instances of a pattern.

3.1.2. Return Values: Success, No Match, Ambiguous Match

The function will return a structured result that provides detailed feedback about the operation's outcome. The return type will be a tuple in the form `{:ok, result_map}` or `{:error, reason_map}` . This allows the calling skill to easily handle different scenarios.

- **Success**: On a successful replacement, it will return `{:ok, %{content: new_content, changed: true, occurrences_replaced: count}}` . The `new_content` is the modified string, `changed` is a boolean flag, and `occurrences_replaced` indicates how many replacements were made (which would be `1` for a single occurrence or the total count if `:all` was specified).
- **No Match**: If the `search_block` is not found in the `content` , it will return `{:error, %{reason: :no_match, message: "Search block not found"}}` . This clearly indicates that the operation failed because the target text was not present.
- **Ambiguous Match**: If the `search_block` is found multiple times and the `:occurrence` option is not `:all` , it will return `{:error, %{reason: :ambiguous_match, count: match_count, message: "Search block found #{match_count} times. Use 'occurrence' option to specify."}}` . This is a critical safety feature that prevents accidental multiple replacements and prompts the user (or the LLM) to be more specific.

This structured return value provides rich feedback, enabling the LLM to understand the result of its edit request and take corrective action if necessary, such as providing more context in the `search_block` or specifying an exact occurrence to target.

3.2. Whitespace-Resilient Matching

The core innovation of the `apply_patch/3` function is its ability to perform matches that are resilient to variations in whitespace. This is achieved by normalizing both the `search_block` and the `content` before performing the comparison, while carefully preserving the original formatting in the final replacement. This approach allows the LLM to provide a search block that is semantically correct but may have incorrect indentation, and the function will still be able to find the corresponding block in the original file.

3.2.1. Normalizing Whitespace in Search and Content Blocks

The key to resilient matching is to normalize whitespace in both the `search_block` and the `content` before searching. A simple and effective normalization strategy is to:

1. **Trim leading and trailing whitespace** from each line.
2. **Collapse consecutive internal whitespace** (spaces and tabs) into a single space.

This process transforms a block of code like:

```elixir
def   my_function(  arg1,   arg2 )  do
  result   =   arg1   +   arg2
  result
end
```

into a normalized form like:

```elixir
def my_function( arg1, arg2 ) do
result = arg1 + arg2
result
end
```

By applying this normalization to both the `search_block` and each potential matching block within the `content` , the function can perform an exact string comparison on these normalized versions. This effectively makes the match "fuzzy" with respect to whitespace, allowing it to succeed even if the original formatting is inconsistent.

3.2.2. Using Regex for Flexible Matching

An alternative or complementary approach to explicit normalization is to use a regular expression for the search. The `search_block` can be converted into a regex pattern where all literal whitespace characters (spaces, tabs, newlines) are replaced with a pattern like `\s+` , which matches one or more whitespace characters. For example, the search block `def my_function do` would be converted into the regex `~r/def\s+my_function\s+do/` . This allows the regex engine to handle the fuzzy matching for whitespace automatically. This method can be very powerful, but it requires careful escaping of any special regex characters that might be present in the `search_block` . A hybrid approach could be to use normalization for the overall block comparison but use a regex-based search to find the initial candidate locations of the block within the larger content, which might be more efficient than scanning the entire content line by line.

3.2.3. Preserving Original Whitespace in the `replace_block`

A crucial aspect of the replacement logic is that **the original whitespace of the matched block in the `content` must be preserved**. The LLM's `replace_block` should be inserted into the file with the same indentation and surrounding whitespace as the block it is replacing. This ensures that the new code integrates seamlessly into the existing file structure and does not introduce new formatting errors. To achieve this, the function must not only find the matching block but also capture the original whitespace. This can be done by performing the match on the normalized text but then using the start and end indices of the match in the *original, non-normalized* content to extract the block with its formatting intact. The `replace_block` is then inserted at this exact location. This ensures that the structural formatting of the file remains consistent, which is vital for maintainability and for preventing the introduction of new linting errors.

3.3. Match Logic and Ambiguity Detection

The `apply_patch/3` function must be able to intelligently handle different matching scenarios. It needs to find all possible matches, determine if the match is unique, and provide clear feedback if it is not. This logic is essential for building a reliable and safe editing tool that gives the user full control over the changes being made.

3.3.1. Finding All Occurrences of the `search_block`

The first step in the matching process is to find all occurrences of the normalized `search_block` within the normalized `content` . This can be done by iterating through the content line by line, creating a sliding window of text that is the same length as the `search_block` , normalizing this window, and comparing it to the normalized `search_block` . Each time a match is found, the start and end indices of the match in the original, non-normalized content are recorded. This process results in a list of all matching locations, which can then be used to determine uniqueness and perform the replacement.

3.3.2. Handling Unique Matches

If the search process finds exactly one match, the operation is straightforward. The function will take the `replace_block` and insert it into the `content` at the location of the unique match. The result will be a success tuple containing the new, modified content. This is the most common and ideal scenario, and the function is optimized for this case.

3.3.3. Handling Multiple/Ambiguous Matches

If the search process finds more than one match, the function's behavior depends on the `:occurrence` option.

- If `:occurrence` is an integer (e.g., `2`), the function will perform the replacement on the match at that specific index in the list of found matches. The result will be a success tuple.
- If `:occurrence` is `:all` , the function will iterate through all found matches and perform the replacement for each one. The result will be a success tuple with a count of the total number of replacements made.
- If `:occurrence` is not provided (defaulting to `1`) or is an integer that is out of bounds, and there are multiple matches, the function will **not** perform any replacement. Instead, it will return an `{:error, reason}` tuple with the `:ambiguous_match` reason, as described in section 3.1.2. This is a critical safety mechanism that prevents accidental, widespread changes to the file.

3.4. Replacement Strategies

The function must support different replacement strategies based on the user's intent, which is communicated through the `:occurrence` option. This provides fine-grained control over which matches are affected by the patch.

3.4.1. Replacing a Specific Occurrence

This is the default behavior, where `:occurrence` is an integer. The function will find all matches and then select the one at the specified index (1-based) for replacement. This is useful when the user knows that the `search_block` appears multiple times but only wants to change one specific instance. For example, if a function is defined twice in a file and the user only wants to update the second definition, they can provide the `search_block` for the function and set `occurrence: 2` .

3.4.2. Replacing All Occurrences

By setting the `:occurrence` option to `:all` , the user can instruct the function to replace every instance of the `search_block` found in the `content` . This is useful for global refactoring tasks, such as renaming a variable or function that is used throughout a file. The function will iterate through the list of all matches and perform the replacement for each one. Because the indices of the content string will shift after each replacement, it is crucial to perform the replacements in reverse order (from the end of the file to the beginning) to ensure that the indices of subsequent matches remain valid.

4. Security Sandboxing (The "Jail" Layer)

Since the Mimo system will execute file operations and shell commands based on AI-generated input, a robust security sandbox is non-negotiable. The "Jail" layer is designed to prevent two of the most critical and common vulnerabilities in systems that execute user input: **Path Traversal** and **Command Injection**. This layer will enforce strict boundaries on what the AI can access and execute, ensuring that it cannot escape its designated workspace or run arbitrary and potentially malicious commands. The implementation will rely purely on Elixir's standard library, using modules like `Path` and `System` in a secure manner. This section details the design of the security functions that will gate all file and terminal skills, providing a hardened barrier between the AI's intentions and the underlying operating system.

4.1. Path Traversal Prevention

Path traversal is a vulnerability that allows an attacker to access files and directories outside the intended web root or application directory. An AI could be tricked into reading or writing to sensitive system files like `/etc/passwd` by providing a path like `../../../etc/passwd` . The `Mimo.Security.Path` module will provide a function to validate and sanitize all file paths, ensuring they are confined to a predefined, safe root directory.

4.1.1. The `expand_safe/2` Function

The core of the path security layer is the `expand_safe/2` function. Its purpose is to take a potentially malicious relative path provided by the AI and a designated `root_dir`, and return a fully expanded, absolute path that is guaranteed to be within the `root_dir`. If the resolved path attempts to escape the `root_dir`, the function will return an error. This function will be called by all file-related skills before any `File` or `File.Stat` operation is performed.

4.1.2. Canonicalizing Paths with `Path.expand/1`

The first step in `expand_safe/2` is to canonicalize the input path using `Path.expand/1`. This function resolves any `.` (current directory) and `..` (parent directory) components in the path, turning a relative path like `foo/../bar` into the absolute path `/workspace/bar` (assuming the current working directory is `/workspace`). This is a crucial step because it prevents simple obfuscation techniques. For example, the malicious path `../../../etc/passwd` would be canonicalized to `/etc/passwd`, making the escape attempt explicit and easy to detect.

4.1.3. Enforcing a Root Directory Constraint

After canonicalizing the path, the function must check if the resulting path is still within the allowed `root_dir`. This is done by checking if the canonicalized path is a prefix of the `root_dir`. However, a simple string prefix check is not sufficient, as it could be fooled by paths like `/workspace-malicious/file.txt`. The correct way is to check if the `root_dir` is a prefix of the canonicalized path. A robust implementation would look like this:

elixir 复制

```elixir
def expand_safe(path, root_dir) do
  expanded_path = Path.expand(path, root_dir)

  if String.starts_with?(expanded_path, Path.expand(root_dir)) do
    {:ok, expanded_path}
  else
    {:error, :path_traversal_attempt, "Path '#{path}' attempts to
escape the root directory '#{root_dir}'."}
  end
end
```

This function first expands the `path` relative to the `root_dir`. It then checks if the resulting `expanded_path` starts with the expanded `root_dir`. If it does, the path is safe. If not, it means the path resolved to a location outside the sandbox, and an error is returned.

4.2. Command Injection Prevention

Command injection is a vulnerability that allows an attacker to execute arbitrary commands on the host operating system. This can happen if user input is passed directly to a shell interpreter without proper sanitization. An AI could be tricked into running a command like `ls; rm -rf /` if the input is not properly validated. The security layer will implement a strict whitelist and sanitization strategy to prevent this.

4.2.1. Whitelisting Allowed Commands

The most effective defense against command injection is to **never pass unsanitized user input to a shell**. The `Mimo.Security.Command` module will maintain a strict whitelist of allowed commands. Any command requested by the AI that is not on this list will be rejected immediately. This is a simple but powerful security measure that drastically reduces the attack surface. The whitelist will be a module attribute, for example:

elixir 复制

```elixir
@allowed_commands MapSet.new(["ls", "cat", "grep", "find", "echo",
"pwd", "whoami"])
```

This list should be as minimal as possible, containing only the commands that are absolutely necessary for the skills to function. Dangerous commands like `rm`, `mv`, `curl`, `wget`, and any shell interpreters like `sh` or `bash` will be explicitly forbidden.

4.2.2. Sanitizing Command Arguments

Even with a command whitelist, there is still a risk of injection through command arguments. For example, if `grep` is a whitelisted command, an attacker could try to inject a command via the search pattern: `grep -i "pattern" file.txt; cat /etc/passwd`. To prevent this, all arguments passed to a whitelisted command must be sanitized. This means escaping or rejecting any characters that have special meaning to the shell. The

`System.cmd/3` function in Elixir is safer than using a shell directly, as it does not invoke a shell by default and passes arguments as a list, which prevents shell interpretation of special characters within the arguments. However, if the command itself is a shell script or if shell features are needed, extreme caution is required.

4.2.3. Blocking Shell Meta-Characters (`;` , `|` , `&&`)

A key part of sanitization is to scan all command arguments for shell meta-characters. These are characters that the shell uses to chain commands or perform other special operations, such as `;` , `|` , `&` , `&&` , `||` , `$(...)` , and backticks. If any of these characters are found in a command argument, the request should be rejected. A simple sanitization function could be implemented using a regular expression:

elixir 复制

```elixir
@shell_meta_chars ~r/[;|&$`<>]/

def sanitize_args(args) when is_list(args) do
  Enum.all?(args, fn arg ->
    not String.match?(arg, @shell_meta_chars)
  end)
end
```

This function checks each argument in the list and returns `true` only if none of them contain a forbidden meta-character. If this check fails, the command execution is aborted with an error. This provides a strong defense against the most common forms of command injection.

4.3. System Call Security

The final layer of security involves the safe execution of the whitelisted and sanitized command. This is handled by the `System` module, but its options must be configured securely to limit the command's capabilities and prevent it from hanging or consuming excessive resources.

4.3.1. Using `System.cmd/3` Safely

`System.cmd/3` is the preferred method for executing external commands in Elixir. It is safer than using `System.shell/1` because it does not, by default, invoke a shell, which eliminates a large class of injection vulnerabilities. The command and its arguments are passed as a list, which prevents the shell from interpreting special characters in the arguments. The function will be used as follows: `System.cmd(command, args, options)` . The `command` will be the whitelisted command, and `args` will be the sanitized list of arguments.

4.3.2. Setting a Secure Working Directory

The `System.cmd/3` function accepts an option list that includes a `:cd` key to set the working directory for the command. This is a critical security feature. All commands executed by the AI should be run from within the designated sandbox directory (the same `root_dir` used for file operations). This prevents commands from accessing files outside their workspace by using relative paths. The `expand_safe/2` function can be used to ensure that any path provided by the AI is resolved within the sandbox, and this sandbox path can then be passed as the `:cd` option.

4.3.3. Enforcing Command Timeouts

It is essential to enforce a timeout on all external commands to prevent them from hanging indefinitely and consuming system resources. The `System.cmd/3` function accepts a `:timeout` option (in milliseconds). A reasonable default timeout (e.g., 30 seconds) should be set for all commands. If a command exceeds this timeout, `System.cmd/3` will return an error. This ensures that the Mimo system remains responsive and is not vulnerable to denial-of-service attacks via long-running or hanging processes. The timeout value could also be made configurable on a per-skill basis, allowing for longer-running tasks if necessary, but it should always be bounded.