Evolving a Rebol-Inspired Language for a Synthetic Mind: A Framework Analysis and Strategic Directions

I. Introduction

A. Project Overview and Goals

The endeavor to create a synthetic "mind and soul" represents a significant ambition at the intersection of artificial intelligence, cognitive science, and programming language design. The core strategy involves leveraging a Large Language Model (LLM) as the foundational cognitive architecture. Central to this architecture is the development of a custom, Rebol-inspired programming language. This language is envisioned not merely as an implementation detail but as a fundamental component of the synthetic entity's cognitive processes, facilitating symbolic manipulation, knowledge representation, and the definition of its operational semantics. Key goals for this language include features such as AI-integrated error handling, robust support for Domain-Specific Languages (DSLs) to model cognitive functions, and seamless database integration for persistent memory and knowledge.

B. Report Objectives

This report aims to provide a comprehensive analysis and strategic advice concerning the evolution of this Rebol-inspired language. The objectives are:

1. To assess the conceptual coherence of using a Rebol-like language in conjunction with an LLM-based cognitive architecture.
2. To propose a minimal yet viable feature set for the language, enabling rapid prototyping and early-stage development, akin to Rebol's relationship with a core functional subset like Scheme.
3. To explore and synthesize research on related topics relevant to the project's broader aims, including tag-based file systems, integrated computing environments, and specific AI technologies.
4. To offer feedback and refinements on the proposed implementation plan for the language, specifically concerning the parser, core data structures (word/value, block), and the evaluation mechanism (eval/do) in the D programming language.

C. Methodology

The analysis presented herein is based on a review of the project's foundational posts, goals, and the specific technical considerations outlined. This is augmented by an examination of research materials covering Rebol, related programming languages (Lisp, Scheme, Tcl, Rye, Wren, Io), advanced AI concepts (SDRs, RAG architectures, LLM augmentation frameworks),

operating system design philosophies (Plan 9, BeOS/Haiku, WonderOS), and advanced error handling techniques. The synthesis of these diverse sources informs the recommendations for the language's design, implementation, and future evolution.

II. Conceptual Coherence and Language Philosophy

A. Rebol's Philosophy as a Foundation

Rebol's design philosophy offers a compelling foundation for a language intended to underpin a synthetic mind. Rebol, standing for Relative Expression Based Object Language, was conceived as more than just a programming language; it is also a language for representing data and metadata, providing a consistent architecture for computation, storage, and information exchange.[1] Its emphasis on being lightweight, employing relative expressions through "dialects," and its ability to seamlessly handle diverse datatypes make it particularly suitable for complex systems that require flexibility and expressive power without excessive complexity.[1]

For a project aiming to construct a "mind and soul," a language that inherently treats code and data uniformly, as Rebol does, is highly advantageous. Cognitive structures, memories, beliefs, and even processes can be represented naturally within the language's own structures (primarily blocks). Rebol's design rebels against the notion that software must be large and complex, a principle that aligns well with the challenge of modeling the intricacies of cognition without succumbing to unmanageable system bloat.[1] The language's capacity for symbolic processing and its foundation in denotational semantics provide a robust framework for defining the operational logic of a synthetic intellect.

B. "LLM as Cognitive Architecture" and Language Symbiosis

The paradigm of an "LLM as cognitive architecture" posits the LLM as the seat of semantic understanding, generative capabilities, and perhaps even emergent reasoning. However, LLMs, while powerful, often lack the structured, symbolic processing capabilities inherent in classical AI and traditional programming. A custom, Rebol-inspired language can bridge this gap, serving as the crucial symbolic manipulation layer. In this symbiotic relationship, the language defines the formal structures, data types, and control flows—the "grammar" of thought—while the LLM provides the rich semantic interpretation, pattern recognition, and content generation.

This language can thus become the connective tissue for the LLM's cognitive functions. It provides a formal system through which the LLM's "thoughts" or intermediate processing states can be explicitly represented, manipulated, and inspected. This is critical for debugging, understanding, and guiding the behavior of the synthetic mind. If the LLM generates a plan, that plan can be instantiated as a code block in the custom language. If the mind needs to reason about its beliefs, those beliefs can be structured data within the language. This structured approach offers a degree of precision and control that is difficult to achieve by interacting with an LLM through natural language prompts alone. The language provides the scaffolding upon which the LLM's more fluid capabilities can be organized and

directed.

C. Homoiconicity: Rebol vs. Lisp

Homoiconicity, where a program's code can be manipulated as data using the language itself, is a hallmark of both Lisp and Rebol.[2] Lisp achieves this primarily through S-expressions (nested lists), which represent both code and data structures.[2] Rebol, similarly, uses blocks (``) as its fundamental composite values, capable of holding sequences of other values, including words, literals, and other blocks, thereby representing both code and data structures.[2]

This property is profoundly significant for a project aiming to create an evolving "mind and soul." Homoiconicity facilitates powerful metaprogramming capabilities. The synthetic mind, through its own language, could inspect, analyze, transform, and even generate its own "mental programs" or cognitive strategies. If a learning process identifies a more efficient way to solve a problem, this new strategy can be encoded as a block of code and integrated into the mind's operational repertoire. DSLs, crucial for specialized cognitive functions, become easier to define and manipulate when the language treats their definitions as data.

The ability for the language to serve as a medium for the "soul's" self-representation and evolution is a direct consequence of homoiconicity. The mind's "source code"—its core beliefs, operational principles, and even its ethical framework—can be represented within the language itself. This allows for introspection (the mind examining its own structure) and self-modification (the mind adapting and evolving its structure over time). This is a far more integrated and dynamic approach than having a fixed codebase that is externally modified. The language becomes a living part of the entity it defines.

D. Dialecting for Cognitive Functions

Rebol's concept of "dialects" is arguably its most potent feature for creating extensible and expressive systems. A dialect is a sub-language, or a domain-specific language (DSL), designed for a particular purpose and typically interpreted by a specific function that processes a Rebol block in a custom way. This mechanism allows for the creation of highly specialized notations for different tasks without altering the core language syntax. For example, Rebol uses dialects for GUI layout (VID), drawing, and parsing.[3]

In the context of a synthetic mind, dialects offer a powerful paradigm for defining "cognitive primitives" or specialized languages for various mental functions. One could envision:

- A planning-dialect to express goals, actions, and conditional execution paths.
- A belief-dialect for representing knowledge, certainty levels, and sources of information.
- An emotion-dialect to model affective states, triggers, and responses.
- A learning-dialect to define rules for adaptation and knowledge acquisition.

Each of_these_ DSLs would be represented as a block in the custom Rebol-inspired language, processed by a dedicated interpreter function written in D (the host language for the interpreter). This approach makes the cognitive architecture highly modular and extensible. New cognitive capabilities can be added by defining new dialects, and existing ones can be refined without disrupting other parts of the system. The LLM itself could even participate in

the design or evolution of these dialects, perhaps by generating dialect specifications or suggesting improvements based on its understanding of the cognitive task at hand. This makes the system not only programmable but also "meta-programmable" at the level of its cognitive functions.

III. Minimal Language Core for Rapid Prototyping (Rebol's Scheme)

To facilitate rapid prototyping, the initial version of the custom Rebol-inspired language should focus on a minimal yet expressive core, analogous to how Scheme represents a minimalist Lisp. This core should capture the essence of Rebol's "code as data" philosophy and its block-based structure.

A. Core Datatypes

While Rebol boasts a rich set of over 40 datatypes [4], a minimal viable prototype can start with a more constrained set. The absolute essentials for a Rebol-like foundation are:

- **block!**: The cornerstone datatype, representing ordered sequences of other values. Blocks are used for code, lists, and general data structures.[5] Their ability to contain any other datatype, including other blocks, is fundamental.
- **word!**: Symbols used for variables, function names, and keywords within dialects.[5] It's crucial to support different functional types of words early on, even if their full semantics are built out incrementally:
    - word (normal word, for variable lookup or function call)
    - set-word! (e.g., my-var:) for assignment.
    - lit-word! (e.g., 'my-symbol) to represent the word itself as a symbol without evaluation.
    - get-word! (e.g., :my-var) to fetch the value of a variable (though this can sometimes be syntactic sugar over normal word evaluation depending on the exact evaluation model chosen).
- **string!**: For textual data, GUI elements, and communication with the LLM.
- **integer!** and **decimal!**: For numerical computations.
- **logic!**: Boolean values (true, false) for conditional logic.[4]

Other Rebol datatypes like char!, date!, time!, tuple!, file!, url!, tag!, email!, money!, pair!, issue!, and binary! [4] can be added progressively as the need arises. The initial focus should be on block! and the various word! types, as these are most critical for establishing the language's Rebol-like character, its "code as data" nature, and its dialecting capabilities. Without them, the language would lose its fundamental Rebol essence.

B. Evaluation Model: do and reduce

The evaluation model should be simple and predictable, adhering to Rebol's principles:

- **do function**: The primary mechanism for evaluating a block!. It processes the expressions within the block and typically returns the result of the last evaluated

expression.[6] Blocks themselves are data and are not evaluated unless explicitly passed to do or a function that internally uses do (like if or loop).[6]
- **reduce function**: Evaluates each expression within a block and returns a new block containing all the results.[6] This is essential for constructing blocks of evaluated values.
- **No Operator Precedence**: Rebol famously eschews complex operator precedence rules. Evaluation within a block proceeds strictly from left to right.[5] For example, 2 + 3 * 10 evaluates to 50.
- **Parentheses for Grouping**: To control the order of evaluation, parentheses () can be used. Expressions within parentheses are evaluated first, and their result is then used in the surrounding expression.[5] Note that in Rebol, paren! is a distinct datatype that forces immediate evaluation of its contents.[4] For a minimal start, simple grouping behavior might suffice before implementing paren! as a full datatype.

C. Functions: Definition and Application

A simple mechanism for defining and applying functions is essential:
- **Function Definition**: A native function (e.g., func or function) should allow the creation of user-defined functions. A common Rebol pattern is my-func: func [spec-block][body-block], where spec-block defines arguments and local variables, and body-block contains the code to be executed.
- **Function Application**: Functions are called by placing the function name (a word!) followed by its arguments: my-func arg1 arg2.
- **Blocks as Arguments**: The ability to pass unevaluated blocks as arguments to functions is critical for creating dialects and control structures.[7] The function itself then decides how and when to evaluate these blocks.

D. Words and Scoping

- **Word Characteristics**: Words should be case-insensitive, as in Rebol.[5] They can contain hyphens and other special characters as defined by Rebol's rules.[5]
- **Assignment**: Basic assignment is achieved using set-word!: my-variable: 123.
- **Scoping Model**: For a rapid prototype, implementing a simple **lexical scoping** mechanism is advisable. This involves managing a stack of contexts (environments), where variables are looked up by searching from the current context outwards. While Rebol features a more complex "definitional scoping" model where symbols can carry invisible bindings established at load/parse time [8], this adds significant complexity to initial implementation. Lexical scoping is a well-understood and more straightforward starting point, with definitional scoping being a potential future refinement.

E. Minimal Native Functions

A small set of built-in functions, implemented in D, will be necessary to make the language usable:
- **Core Evaluators**: do, reduce.

- **Control Flow**: if (evaluates a condition and then one of two blocks), loop (repeats a block a number of times) or a more general while.
- **Output/Introspection**: print (to display values), probe (to inspect values, similar to Rebol's probe).
- **Function Definition**: func (or function).
- **Variable Setting**: set (often implicit via set-word!, but an explicit set function can be useful).
- **Series/Block Operations**: first, next (or second, third, etc., or a general pick), append, copy (Rebol's copy is important as values are not copied by default on assignment).
- **Type Checking**: Predicates like block?, word?, string?, integer?, etc.

Drawing inspiration from Scheme's minimalism [9], the initial set of native functions should be kept as small as possible, focusing on those that enable the core evaluation loop and basic data manipulation. Many other functions can then be defined within the language itself once this core is established.

IV. Implementation Plan Feedback (Parser, Structs, Eval in D)

The proposed implementation plan—parser, word/value struct, block struct, and eval/do in D—is a sound approach for building the Rebol-inspired language.

A. Parser Design

The parser is the first crucial component, transforming raw text input into the language's internal data structures.

1. **Lexer/Tokenizer**:
    - The lexer's primary responsibility is to break the input stream into a sequence of tokens. Each token should represent a fundamental Rebol datatype or a structural element.
    - It must recognize Rebol's diverse literal forms for numbers (integer, decimal), strings (double-quoted and brace-delimited), times, dates, tuples, files (%file), URLs, tags (<tag>), email addresses, issues (#issue), binary data (#{...}), and characters (#"c").[4]
    - Words, including their variants (word:, 'word, :word), and special characters that form operators or delimiters ([ ] ( ) { } : ; /) must also be identified.[5]
    - Whitespace (spaces, tabs, newlines) generally acts as a separator but is significant in some contexts (e.g., ending a word). Comments (typically semicolon to end-of-line in Rebol) should be discarded.
2. **Parser**:
    - Given Rebol's relatively simple, regular syntax (primarily sequences of values within blocks), a **recursive descent parser** is often a suitable and straightforward choice.
    - The parser will consume tokens from the lexer and construct an internal

representation of the program. This could be an Abstract Syntax Tree (AST), but for Rebol-like languages, it's often more direct to build a sequence of "value" structures, typically a main block representing the script.
    - The parser must correctly handle nested structures, particularly blocks within blocks.
    - While Rebol itself has a powerful PARSE dialect for defining grammars [3], this is a feature *of* Rebol used for general parsing tasks within Rebol programs. For implementing the language itself from scratch in D, a traditional parser (like recursive descent) is needed for the initial bootstrapping of the language. The *principles* of rule-based series processing found in PARSE might offer inspiration for the parser's internal logic, but PARSE itself is not the tool to parse the base language.

B. Core Data Structures: Word/Value and Block Structs in D

These D structures will represent the language's data at runtime.

1. **Value Struct**:
    - This will be a cornerstone struct, likely implemented as a **discriminated union** in D. It needs to represent every possible datatype in the language.
    - It should contain a type field (an enum: TYPE_WORD, TYPE_BLOCK, TYPE_INTEGER, TYPE_STRING, TYPE_FUNCTION, etc.) and a union (data) holding the actual D representation of the value (e.g., long for integers, double for decimals, string for D strings, Block* for blocks, WordData* or an ID for words).
2. **Word/Symbol Representation**:
    - Words (symbols) in the language need to be **interned**. This means each unique word string maps to a single, canonical representation (e.g., a unique integer ID or a pointer to a shared string object). Interning is crucial for efficient comparison (words can be compared by ID/pointer instead of string comparison) and memory usage.
    - A global symbol table (hash map) will manage the mapping from strings to their interned representations.
    - The Value struct for a word! datatype would store this interned ID or pointer. Different word types (e.g., set-word!, lit-word!) might be distinguished by the type field in the Value struct or by additional flags within a WordData struct if words carry more information.
3. **Block Struct**:
    - A block! is fundamentally an ordered sequence of Value structs. In D, this can be implemented using std.container.array.Array!Value or a custom dynamic array structure.
    - It should support efficient appending of new values, iteration, and potentially random access (though many Rebol series operations are sequential).
    - A significant consideration, if aiming for close Rebol compatibility, is that composite values (like blocks) assigned to variables are typically not copied by default; a specific copy function is required to create a distinct duplicate. This

implies that the Block struct might need to manage shared data or implement copy-on-write semantics if blocks are passed around and modified, to avoid unintended side effects. For an initial prototype, deep copying on assignment or modification might be simpler, deferring advanced sharing mechanisms.

C. eval/do Implementation in D

The eval (or do) function is the heart of the interpreter.

1. **Core Loop**:
    - The main eval function will take a pointer to a Block struct (or a representation of a block) as input.
    - It will iterate through the Value structs contained within this block, one by one.
2. **Evaluation Logic per Value Type**:
    - **Literals** (integers, strings, etc.): These values evaluate to themselves. The function simply returns a copy or a reference to the literal value.
    - **word!**:
        - The word is looked up in the current **context** (environment or symbol table).
        - If it's bound to a variable, the variable's Value is returned.
        - If it's bound to a function (native or user-defined), this signals a function call. The evaluator prepares to gather arguments and execute the function.
        - If unbound, it's typically an error, unless it's a lit-word! (which returns the word symbol itself).
    - **set-word! (e.g., my-var:):**
        - The evaluator expects the *next* Value in the block to be the expression whose result will be assigned.
        - This next expression is evaluated.
        - The resulting Value is then bound to the word! (stripped of its colon) in the current context.
    - **Function Calls**:
        - If a word! resolves to a function, the evaluator collects the required number of subsequent Values from the block as arguments.
        - **Argument Evaluation**: For most functions, these argument expressions are themselves evaluated *before* being passed to the function. Special forms (like if, or func itself) may quote some of their arguments (i.e., receive them unevaluated as blocks or words).
        - **Native Functions**: If the function is a native one (implemented as a D function), the D function is called with the (evaluated) D representations of the arguments. The D function returns a Value struct, which becomes the result of the expression.
        - **User-Defined Functions**: If it's a user-defined function (represented as a Value of TYPE_FUNCTION, likely containing a specification block for parameters and a body block):
            1. A new context (scope/frame) is created, linked to the function's definition context (for lexical scoping).

    2. The evaluated argument Values are bound to the parameter names (words) within this new context.
    3. The eval function is then called recursively on the function's body Block. The result of this recursive call is the result of the function call.
3. **Context Management**:
    - A mechanism for managing **contexts** (also known as environments or scopes) is essential. A context maps word identifiers (interned IDs) to their Value structs.
    - For lexical scoping, a stack of contexts is typically used. When a function is called, a new context frame is pushed onto the stack. When a word is looked up, the stack is searched from the top (current context) downwards.
    - Rebol's "definitional scoping," where symbols can carry invisible bindings that dictate their lookup [8], is an advanced feature. It implies that bindings are resolved or annotated more statically, perhaps during parsing or loading, and these binding annotations are carried with the code blocks. Implementing this faithfully from the start is complex.
    - For a D-based prototype aiming for quick development, starting with a more traditional **lexical scoping** model using a stack of context frames (e.g., hash maps from word ID to Value) for eval/do is a pragmatic approach. This is a well-understood mechanism and easier to implement and debug. Once this foundational lexical scoping is robust, the more nuanced Rebol-specific binding model can be investigated as a significant enhancement. This staging makes the "prototyped quickly" goal more attainable while still providing a path towards greater Rebol fidelity.

This structured approach to implementation, starting with a simpler lexical model and clearly defined data structures, should provide a solid base for the custom language.

V. Evolving the Language: Advanced Features for Cognitive Functionality

Beyond the core language, specific advanced features are crucial for realizing the "mind and soul" aspiration, particularly those that integrate AI capabilities directly into the language's fabric.

A. AI-Integrated Error Handling

Traditional error handling often stops at displaying a message and stack trace. For a synthetic mind, errors are learning opportunities. AI-integrated error handling can transform runtime failures into rich diagnostic and recovery processes.

- **Concept**: When an error occurs, the language runtime should not just halt or throw a simple exception. Instead, it should:
    1. Capture extensive context: the error type, message, the specific code block/expression that failed, the current call stack, and the state of relevant variables.

2. Pass this rich contextual information to the integrated LLM.
3. The LLM can then analyze this context to provide:
    - Natural language explanations of *why* the error likely occurred, beyond a simple type mismatch.
    - Suggestions for code modifications to fix the error.
    - Hypotheses about incorrect assumptions or flawed logic leading to the error.
    - Potentially, alternative code paths or strategies to achieve the intended goal.

- **Resumable Exceptions**: This is a key mechanism for enabling sophisticated AI-driven recovery. Inspired by systems like Smalltalk [13] and the Common Lisp Condition System [15], an error signals a "condition" rather than immediately unwinding the stack. This allows handlers—which could include or consult the LLM—to decide on a course of action:
    - **Resume**: Continue execution from the point of error, possibly with a corrected value or modified state provided by a handler or the LLM. Smalltalk's #resume: is notable for not unwinding the stack.[13]
    - **Retry**: Re-attempt the failed operation. This might occur after the LLM suggests a change to the environment or input that could lead to success. Smalltalk's #retryUsing: allows restarting with a modified block.[13]
    - **Restart**: Transfer control to a predefined, named "restart point" higher up in the call stack. Common Lisp's invoke-restart is a powerful example.[17] The LLM could analyze the situation and suggest which available restart is most appropriate.
    - **Propagate/Decline**: If no handler can resolve the issue, the error can be propagated further up the stack, or a handler can explicitly decline to handle it.
- **Comparison with Other Systems**:
    - Objective-C's NSException (for programmer errors) and NSError (for runtime/recoverable errors) offer a useful distinction.[18] An AI-integrated system might leverage such a distinction to prioritize LLM intervention for certain error classes.
    - The Io programming language uses a more conventional try/catch mechanism with an Exception object that can be raised.[20] For AI integration, the richness of the Exception object and the ability of the catch block to communicate with an LLM would be paramount. Resumability is not a standard feature in such systems.
- **Justified Programming and Reason Parameters**: The concept of "Justified Programming — Reason Parameters That Answer 'Why'" [24], while not fully elaborated in the provided materials, hints at a paradigm where computational actions are associated with explicit justifications or reasons. If the custom language were to incorporate such a feature—perhaps through metadata annotations on code blocks, special comment forms, or even a dedicated dialect for "justified execution"—the AI-integrated error handling could become significantly more powerful. When an error occurs, the LLM would receive not only the state of the program but also the *stated intentions or reasons* behind the sequence of operations that led to the failure. This would enable the

LLM to perform a more "cognitive" diagnosis, assessing whether the error stemmed from a faulty mechanical step, a flawed premise in the reasoning, or an incorrect justification for an action. For instance, the LLM might respond: "The operation to access memory location X failed. This operation was justified by the reason 'retrieve sensory input Y'. However, sensory input Y was marked as unreliable due to reason Z. Perhaps a different justification or a data validation step is needed before proceeding." Resumable exceptions could then allow the LLM to suggest alternative actions based on revised reasons or justifications, leading to a more adaptive and intelligent recovery process. This moves beyond simple error correction to a deeper understanding of intent and failure within the synthetic mind.

B. Domain-Specific Languages (DSLs) for Cognitive Modules

Rebol's dialecting capability is its most significant strength for creating DSLs.[3] A dialect is a "sub-language used for a specific purpose", typically a Rebol block interpreted in a specialized way by a function. This is ideal for defining the building blocks of a synthetic mind.

- **Application to Cognitive Modules**:
  - **Planning DSL**: plan do-sub-plan [sub-action "C"]]]
  - **Belief/Knowledge Representation DSL**: belief "water is wet" type fact certainty 0.99 source "direct_experience:touch"
  - **Emotional Modeling DSL**: emotion fear intensity 0.7 trigger [event "loud_noise"] response [action "seek_shelter"]
  - **Learning Rule DSL**: learn when] then
- **Implementation**: Each DSL would be a block in the custom Rebol-like language. This block is then passed to a dedicated D function (the dialect's interpreter) that parses and executes the DSL's specific semantics. The LLM could play a role in designing these DSLs, translating high-level cognitive concepts into formal dialect structures, or even evolving the dialects over time. Rebol's PARSE dialect itself is an example of such a powerful, embedded DSL for parsing [3]; similar specialized dialects can be created for various cognitive functions. The key is that these DSLs leverage the language's fundamental datatypes (block!, word!, string!, etc.) and its "code as data" nature, making them seamlessly integrable into the overall system. The flexibility of functions accepting blocks as parameters is fundamental to this approach.[7]

C. Database Integration

A synthetic mind requires robust mechanisms for persistent storage of its knowledge, memories, learned behaviors, and the evolving state of its "soul."

- **Rebol's Approach**: Rebol itself can manage structured data within blocks, which can be serialized to files. For more complex or large-scale persistence, it can interface with external database systems. The "REBOL Official Guide" mentions the development of a "distributed database management system and consumer database dialect" as an illustrative example of Rebol's capabilities in this area.[29]
- **Integration Strategies**:

    - **Native-like Persistence**: Design dialects for querying and manipulating data stored in large, Rebol-like block structures. These blocks can be persisted as files or in a specialized block-oriented database. This leverages the language's inherent data structuring capabilities.
    - **External Database Connectivity**: Provide native functions or a dialect to connect to standard database systems (SQL, NoSQL, graph databases). The LLM could assist by generating queries in the target database's native language (e.g., SQL) based on higher-level requests formulated in the custom language.
- **Relevant Technologies for Cognitive Persistence**:
    - **Triadic Memory**: This system, designed for storing and retrieving ordered triples of Sparse Distributed Representations (SDRs) (X, Y, Z), is highly relevant.[30] It allows for associative recall (e.g., given X and Y, find Z), which is powerful for semantic relationships, analogies, and pattern completion. The custom language could feature a dialect for interacting with a Triadic Memory, with commands like store-triple sdrX sdrY sdrZ and query-triple sdrX sdrY?.
    - **mem0**: A persistent memory layer specifically designed for LLMs and AI agents, offering multi-level memory (User, Session, Agent state).[33] The custom language could serve as the interface for the synthetic mind to manage its own experiences, learned preferences, and interaction histories using a system like mem0.

These advanced language features, particularly AI-integrated error handling and DSLs for cognitive functions, combined with robust database integration, will be pivotal in transforming the custom language from a mere programming tool into an active component of the synthetic mind's cognitive processes.

VI. Supporting Ecosystem: Integrated Environments and Knowledge Representation

The language for the synthetic mind will operate within a broader ecosystem of data management and interaction. Concepts from advanced file systems and integrated computing environments can provide valuable inspiration.

A. Tag-Based File Systems and Semantic Organization

Traditional hierarchical file systems often impose rigid structures. A more flexible approach, suitable for the dynamic and interconnected nature of knowledge in a mind, is offered by tag-based systems.

- **Core Concept**: As proposed by Nayuki, files are treated as immutable sequences of bytes, each identified by a content hash (e.g., SHA-256).[34] Metadata, including descriptive tags, are stored as separate, linkable objects rather than being embedded in the file or dictated by a directory path. Tags can range from simple strings to complex, structured objects representing relationships or semantic categories.[34]
- **Relevance to Project**: The synthetic mind will generate, process, and retrieve vast

quantities of information—memories, sensory data, learned concepts, internal states. A tag-based organization, managed through the custom Rebol-like language, could provide a highly adaptable and powerful way to structure this "knowledge base."
    - The content of a "file" (a hash-identified data blob) could be a raw memory trace, a processed sensory input stream, or a serialized cognitive model.
    - Tags, represented as word!s, string!s, or even block!s in the custom language, could denote concepts, emotions, temporal information, causal links, sources, or any other relevant metadata associated with the data blob.
- **Practical Examples**: The BeOS File System (BFS), and its successor in Haiku, demonstrated early capabilities in this direction with extensive metadata support, indexing, and querying features that allowed files to be treated almost like records in a database.[35] Files in BFS possess attributes like mimetypes that can be queried, offering a richer organization than simple filenames and paths.
- **Implementation within the Custom Language**: The language could incorporate built-in functions or a dedicated dialect for creating data blobs, generating their hashes, and then creating, associating, querying, and managing tags linked to these blobs. The actual storage could be a content-addressable system.
- **Language-Defined Schemas for Tagged Data**: Nayuki's proposal highlights the eventual need for data schemas to manage complex tags and enable more sophisticated, relational database-like queries.[34] Rebol's dialects are, in essence, schemas for interpreting blocks of data. This suggests a powerful synergy: the custom Rebol-inspired language itself could be used to define the "schemas" for both the tags and the data blobs they describe. A "tag schema" could be a Rebol-like block that specifies the expected structure and datatypes for a complex tag (e.g., a tag representing a causal relationship might require fields for cause, effect, and certainty-level). The language's own parsing and validation capabilities (even if custom-implemented, inspired by Rebol's PARSE) could then be used to ensure that tagged data structures adhere to these schemas. This approach would create a self-describing, internally consistent knowledge base, verifiable by the language itself. The LLM could also participate by helping to generate, understand, or evolve these schemas based on the semantic content it processes. Therefore, the language should be designed with features that facilitate the definition and validation of data schemas, perhaps through a specific schema-dialect.

B. Integrated Computing Environments (ICEs)

The concept of an ICE, where all system components are deeply interconnected through common protocols and data representations, offers a model for how the synthetic mind might operate.

- **Plan 9 from Bell Labs**: This operating system exemplified the philosophy of "everything is a file," where all system resources (files, devices, network connections, processes) are accessible through a unified file system interface via the 9P protocol.[39] Per-process namespaces allowed users to customize their view of these resources, fostering a highly

integrated and adaptable environment.[40]

- **WonderOS / Itemized OS**: Alexander Obenauer's research explores an "itemized" user environment where "items"—granular pieces of information or functionality—are the fundamental units, distinct from the applications that render them or the services that supply them.[41] This aims for greater fluidity, interoperability, and operator modifiability. OLLOS, an experimental instance, organizes all items on a unified timeline.[44]
- **XXIIVV / Uxn**: This project by Rekka & Devine represents a "clean-slate computing stack" featuring the Uxn virtual machine, a minimal bytecode instruction set, and a self-hosted ecosystem of tools written in Uxntal (Uxn's assembly language).[45] It prioritizes portability, resilience against software obsolescence, and simplicity, allowing the entire environment to be understood and maintained by a single individual.
- **Relevance to the Synthetic Mind**: The synthetic mind itself can be conceptualized as a specialized, self-contained ICE. The custom Rebol-inspired language would serve as its "shell," its primary interaction modality, and the language for its internal "programs." The dialects defined within this language would function as its "tools" and "applications." The tag-based knowledge system would be its "file system."
- **The Synthetic Mind as a Self-Hosted, Evolvable ICE**: Drawing from the principles of Uxn (self-containment, minimalism) [45], WonderOS (operator modifiability, itemization) [42], and Plan 9 (unified resource access) [40], the synthetic mind, powered by its Rebol-like language, could evolve into a self-hosted system. The language interpreter acts as the "kernel." Its dialects are the system utilities and cognitive applications. Crucially, leveraging the "code as data" nature of the language (homoiconicity), the mind could use its LLM-driven analytical capabilities to inspect, debug, modify, and extend its own components—its dialects, its core functions, its knowledge schemas—at runtime. This mirrors the dynamic, live-coding environments of Lisp Machines or Smalltalk systems, where the system is perpetually malleable by its user (or in this case, by the "mind" itself). This aligns directly with the aspiration of creating an evolving "mind and soul." The language is not merely a tool *for* the mind; it becomes an integral, modifiable part of the mind's own operating environment and cognitive architecture. Therefore, designing the language and its interpreter with robust introspection and self-modification capabilities from the outset is a strategic imperative. Consideration should be given to how new dialects or even core language functions could be defined, validated, and integrated dynamically by the synthetic entity itself.

C. Interfacing with Advanced AI Concepts

The custom language must be able to effectively interface with and orchestrate various advanced AI components and theories.

- **Sparse Distributed Representations (SDRs) and Triadic Memory**:
    - SDRs are high-dimensional binary vectors, mostly sparse (few active bits), where semantic meaning is encoded in the pattern of active bits. They are well-suited for representing concepts and measuring semantic similarity through overlap.[46]
    - Triadic Memory, as described by Peter Overmann, stores and associatively

retrieves triples of SDRs (X, Y, Z).[30] This is highly valuable for representing and querying relational knowledge.
    - **Language Interface**: The custom language could natively support an SDR datatype and provide functions or a dialect for operations like store-triple sdrX sdrY sdrZ and query-triple sdrX sdrY?variableZ, allowing the mind to directly manipulate and query this powerful associative memory.
- **Retrieval Augmented Generation (RAG) Architectures**: RAG systems enhance LLM responses by retrieving relevant information from external knowledge sources and providing it as context.
    - **PathRAG**: Retrieves key relational paths from a knowledge graph to structure prompts.[50] The custom language could represent these paths as structured data (e.g., blocks of words and links) and use them in constructing prompts for its internal LLM.
    - **HippoRAG**: Inspired by human long-term memory, this framework integrates knowledge across documents using knowledge graphs and Personalized PageRank.[51] The language could manage interactions with HippoRAG's indexing and retrieval mechanisms, perhaps through a specialized dialect.
    - **Generative FrameNet**: Dynamically generates task-specific semantic frames for information retrieval.[53] The custom language could use a dialect to define, invoke, or even help construct these frames to guide the LLM's information access.
- **LLM Augmentation Frameworks**: These frameworks provide patterns for improving LLM reasoning, tool use, and safety.
    - **Language Hooks**: A modular framework where small, conditional programs ("hooks") trigger based on generated text to augment LLM reasoning, decoupling tool usage from the model and prompt.[54] The custom Rebol-like language is a natural fit to be the language in which these hooks are written, or to define their trigger conditions and actions.
    - **CoRE (LLM as Interpreter)**: This system proposes using an LLM to interpret and execute natural language instructions, pseudo-code, or flow programs.[57] The Rebol-inspired language, with its clear block structure and dialecting capabilities, could serve as a more structured and powerful form of "pseudo-code" or "flow programming language" for the CoRE LLM-interpreter.
    - **CaMeL (Capabilities for Machine Learning)**: A defense against prompt injection that uses a dual-LLM architecture (Privileged and Quarantined) and a capability-based execution layer with policy checks.[60] The Privileged LLM generates plans (e.g., in pseudo-Python) that are then executed. The custom Rebol-like language could be used to define these plans or the security policies that CaMeL enforces.
- **Persistent Memory Systems for LLMs**: Systems like **mem0** provide an intelligent, multi-level memory layer for AI agents, enabling them to remember user preferences and learn over time.[33] The custom language would be the natural interface for the synthetic mind to interact with such a memory system, storing its "experiences," learned

associations, and evolving self-model.

- **Dialects as Facades for Complex AI Subsystems**: Many of these advanced AI systems (Triadic Memory, RAGs, LLM augmentation frameworks) possess their own complex APIs, data formats, or operational models. Instead of exposing this complexity directly to the core logic of the synthetic mind or requiring the LLM to master each idiosyncratic interface, the Rebol-inspired language can leverage its dialecting capability. Specialized "facade dialects" can be created for each major AI subsystem. For example:
    - A triadic-memory-dialect could offer simple commands like remember (A relates-to B as C) or what-follows (A relates-to B).
    - A knowledge-retrieval-dialect could provide high-level queries like find-supporting-evidence-for [belief-block] using-rag-hipporag.
    - A secure-action-dialect could wrap interactions with a CaMeL-like framework. This approach encapsulates the interaction logic with each subsystem, promoting modularity within the synthetic mind's architecture. It allows underlying AI components to be updated, reconfigured, or even replaced with alternatives without requiring extensive rewrites of the mind's core "cognitive programs." The dialects provide stable, semantically appropriate interfaces tailored to the mind's needs.

D. Table: Relevance of Supporting Technologies to Project Goals

To systematically illustrate how these diverse research areas can contribute to the project, the following table summarizes their relevance and potential integration points:

Technology/Concept	Brief Description	Relevance to 'Mind/Soul' Project	Potential Integration via Custom Language
Tag-Based File Systems (e.g., Nayuki's model [34], BFS/Haiku [35])	Files as immutable, hash-identified blobs; metadata and relationships as external, linkable tags. Queryable attributes.	Flexible, semantic organization of vast internal knowledge, memories, sensory data, and learned concepts.	Dialect for creating/managing data blobs and tags. Language-defined schemas for tag structures and data validation. Native functions for querying the tagged knowledge base.
Integrated Computing Environments (ICEs) (e.g., Plan 9 [40], WonderOS [42], Uxn [45])	Unified access to all resources; customizable namespaces; self-contained, modifiable systems.	Provides a model for the synthetic mind as a self-hosted, evolvable system where all its components are managed and modified through its own	The language itself forms the core of the ICE. Introspection and metaprogramming features allow the "mind" to modify its own language

		language.	constructs, dialects, and tools.
Sparse Distributed Representations (SDRs) [46]	High-dimensional, sparse binary vectors where meaning is encoded in active bits; good for semantic similarity.	Biologically inspired representation for concepts, percepts, and internal states, enabling robust pattern matching and similarity assessment.	Native SDR datatype. Functions for SDR creation (encoding), comparison (overlap, distance), and manipulation (union, intersection).
Triadic Memory [30]	Associative memory for storing and recalling (X,Y,Z) triples of SDRs.	Foundation for semantic memory, allowing storage and retrieval of relationships, analogies, and sequences.	Dialect for store-triple, query-triple (e.g., recall Z given X, Y), and other associative memory operations.
Retrieval Augmented Generation (RAG) Architectures (PathRAG [50], HippoRAG [51], Generative FrameNet [53])	Enhancing LLM responses by retrieving and incorporating information from external knowledge sources.	Grounding the LLM's outputs in factual data, enabling more accurate and contextually relevant reasoning and communication for the synthetic mind.	Dialects to define queries for different RAG systems, to structure retrieved information (e.g., relational paths, semantic frames), and to integrate this information into LLM prompts.
LLM Augmentation Frameworks (Language Hooks [55], CoRE [58], CaMeL [61])	Architectural patterns for improving LLM reasoning, tool use, safety, and modularity.	Providing structured ways to extend the LLM's capabilities, manage its interactions with tools/environment, and enforce safety/policy constraints.	The custom language can serve as the scripting/programming layer for hooks, the structured input for CoRE-like interpreters, or the plan/policy definition language for CaMeL-like systems. Dialects can abstract these frameworks.
Persistent Memory for LLMs (e.g., mem0 [33])	Specialized memory layers for AI agents to retain experiences, preferences, and state over time.	Enabling the synthetic mind to learn from past interactions, maintain a consistent persona ("soul"), and adapt its	Dialect or native functions for storing and retrieving information from the persistent memory

		behavior.	system, managing different memory stores (e.g., episodic, semantic, procedural).

This integrated approach, where the custom language serves as the central nervous system connecting these diverse technologies, is key to realizing a coherent and evolvable synthetic mind.

VII. Conclusion and Strategic Roadmap

The development of a Rebol-inspired language for a synthetic mind and soul is a multifaceted challenge that blends innovative language design with cutting-edge AI research. The analysis indicates a high degree of conceptual coherence between the chosen language paradigm and the project's ambitious goals.

A. Summary of Key Recommendations

1. **Embrace Rebol's Philosophy**: Leverage Rebol's strengths in "code as data," rich datatypes, and dialecting as the core philosophy. This provides the flexibility needed for representing and manipulating cognitive structures.
2. **Minimal Core First**: For rapid prototyping, focus on a minimal language core: block!, word! (and variants), essential literals, do/reduce evaluation, basic function definition/application with lexical scoping.
3. **Strategic D Implementation**: Implement the parser, core data structures (Value, Block), and eval/do loop in D. Start with lexical scoping for simplicity, with Rebol's definitional scoping as a future enhancement.
4. **AI-Integrated Error Handling**: Design error handling with resumable exceptions (inspired by Smalltalk/Lisp) to allow LLM-driven diagnosis and recovery. Explore "reason parameters" to provide deeper context to the LLM.
5. **DSLs for Cognition**: Utilize dialects extensively to define specialized languages for cognitive functions like planning, belief representation, and learning.
6. **Robust Persistence**: Integrate with database solutions, particularly exploring concepts like Triadic Memory for associative knowledge and systems like mem0 for LLM state persistence.
7. **Ecosystem Integration**: Draw inspiration from tag-based file systems (for knowledge organization) and integrated computing environments (for the mind's operational model). Use dialects as facades to interface with complex AI subsystems.
8. **Self-Evolution**: Design the language with introspection and metaprogramming capabilities from the outset to enable the synthetic mind to evolve its own cognitive software.
9. **Ethical Encoding**: Consider using the language itself (e.g., through dedicated dialects or data structures) to encode and enforce ethical principles and behavioral constraints.

B. Phased Approach to Language Development and Integration

A phased development approach is recommended to manage complexity and achieve incremental progress:

- **Phase 1: Core Language Prototype (MVP)**
  - Implement the minimal datatypes (block!, word!, string!, integer!, logic!).
  - Implement do and reduce with left-to-right evaluation.
  - Implement basic function definition (func) and application.
  - Establish simple lexical scoping.
  - Implement a handful of essential native functions (if, loop, print, set, first, copy).
  - Build a basic parser and eval loop in D.
  - Goal: A working interpreter for a tiny, Scheme-like subset of the language.
- **Phase 2: Essential Rebol-isms and Datatype Expansion**
  - Add more Rebol datatypes as needed (e.g., decimal!, char!, path!, refinement!).
  - Refine word types and their evaluation (set-word!, get-word!, lit-word!).
  - Implement a broader set of Rebol-standard native functions.
  - Improve the parser to handle more complex Rebol syntax.
  - Goal: A language that feels distinctly Rebol-like and can execute more complex scripts.
- **Phase 3: Initial LLM Integration & Basic DSLs**
  - Develop the FFI (Foreign Function Interface) in D to allow the language to call out to the LLM and receive responses.
  - Create a simple dialect for LLM interaction (e.g., ask-llm [prompt-block]).
  - Develop one or two foundational DSLs for core cognitive tasks (e.g., a command execution dialect, a simple goal-setting dialect).
  - Goal: Demonstrate the language orchestrating basic LLM-driven behavior.
- **Phase 4: Advanced Features & Ecosystem Interfaces**
  - Implement AI-integrated error handling with resumable exceptions and LLM consultation.
  - Develop more complex DSLs for planning, knowledge representation, and learning.
  - Integrate with chosen database/persistence solutions (e.g., Triadic Memory, mem0, tag-based storage).
  - Build dialect facades for interacting with external AI systems (e.g., RAG frameworks).
  - Goal: A language that actively supports complex cognitive processes and manages its knowledge ecosystem.
- **Phase 5: Self-Evolution Capabilities and Refinement**
  - Introduce robust introspection features (e.g., examining function bodies, contexts).
  - Enable runtime modification of language constructs (e.g., defining new native functions or modifying dialects programmatically by the "mind" itself).
  - Explore and potentially implement Rebol's definitional scoping model if deemed beneficial.
  - Focus on performance optimization and robustness of the interpreter.

    - Implement mechanisms for encoding and enforcing ethical constraints via the language.
    - Goal: A language that can support a truly adaptive and evolving synthetic mind, capable of self-improvement and operating within defined ethical boundaries.

C. Considerations for Future Evolution Towards the "Mind and Soul" Aspiration

The journey towards a synthetic "mind and soul" is long and fraught with profound questions. The language designed for it will be more than just code; it will be the very medium of its existence and evolution.

- **Introspection and Self-Modification**: The ultimate success of the language in this context hinges on its ability to allow the synthetic mind to understand and alter its own "mental software." This requires deep introspection capabilities (examining its own code, data structures, and execution states) and safe, controlled mechanisms for self-modification.
- **Ethical Framework Encoding**: The concept of a "soul" invariably brings ethical considerations to the forefront. A significant potential of this custom language lies in its ability to serve as a formal medium for encoding and enforcing ethical principles or core behavioral constraints. Rather than relying solely on the often opaque and sometimes unpredictable nature of LLM alignment through prompting or fine-tuning, core ethical rules could be expressed as immutable data structures or specialized dialects within the language. For example, any action plan generated by the LLM (and represented as a code block) might be required to pass through an "ethics-check-dialect" before execution. This dialect would evaluate the plan against the encoded ethical rules. The language's structure itself can enforce that certain critical operations *must* be validated by these ethical DSLs. This approach offers a more transparent, inspectable, and potentially verifiable way to build safety and alignment into the synthetic mind, making ethical reasoning a first-class, structural component of its cognitive architecture.
- **Long-Term Vision**: The language should not be seen as a static entity but as one that co-evolves with the synthetic mind's cognitive capabilities. As the "mind" learns and develops, it may identify limitations in its own language or discover needs for new expressive forms. The capacity for self-modification should, ideally, extend to evolving the language itself.

By strategically combining the practical expressiveness of Rebol with advanced AI integration and a clear-sighted development roadmap, this project can make significant strides towards its ambitious and fascinating goals. The language is not just a tool for building the mind; it is an integral part of what the mind will become.

Works cited

1. What is REBOL?, accessed May 31, 2025, https://www.rebol.com/what-rebol.html
2. Homoiconicity - Wikipedia, accessed May 31, 2025, https://en.wikipedia.org/wiki/Homoiconicity

3. philosophy : Why Rebol, Red, and the Parse dialect are Cool, accessed May 31, 2025, http://blog.hostilefork.com/why-rebol-red-parse-cool/
4. REBOL Datatypes - Random Geekery, accessed May 31, 2025, https://randomgeekery.org/post/2004/12/rebol-datatypes/
5. Chapter 3 - Quick Tour - REBOL Language, accessed May 31, 2025, https://www.rebol.com/docs/core23/rebolcore-3.html
6. Chapter 4 - Expressions - REBOL Language, accessed May 31, 2025, https://www.rebol.com/docs/core23/rebolcore-4.html
7. philosophy : The Central Newbie Question about Rebol/Red, accessed May 31, 2025, http://blog.hostilefork.com/arity-of-rebol-red-functions/
8. rebol : Rebol vs. Lisp Macros - HostileFork Blog, accessed May 31, 2025, http://blog.hostilefork.com/rebol-vs-lisp-macros/
9. What are the minimal set of primitives a Lisp implementation can be built on, one complete enough even though it may be slow? - Reddit, accessed May 31, 2025, https://www.reddit.com/r/lisp/comments/18zpzlw/what_are_the_minimal_set_of_primitives_a_lisp/
10. A Scheme Primer - Spritely Institute, accessed May 31, 2025, https://files.spritely.institute/papers/scheme-primer.html
11. Scheme (programming language) - Wikipedia, accessed May 31, 2025, https://en.wikipedia.org/wiki/Scheme_(programming_language)
12. files.spritely.institute, accessed May 31, 2025, https://files.spritely.institute/papers/scheme-primer.pdf
13. GNU Smalltalk User's Guide: Handling exceptions, accessed May 31, 2025, https://www.gnu.org/software/smalltalk/manual/html_node/Handling-exceptions.html
14. Smalltalk/X Programmers guide - Exception Handling, accessed May 31, 2025, https://live.exept.de/doc/online/english/programming/exceptions.html
15. Exploring the Condition System of Common Lisp - YouTube, accessed May 31, 2025, https://www.youtube.com/watch?v=tT4cPB4Ap5k
16. Common Lisp - The Tutorial - Fast, Fun and Practical (with CLOG) - Reddit, accessed May 31, 2025, https://www.reddit.com/r/lisp/comments/196lain/common_lisp_the_tutorial_fast_fun_and_practical/
17. 9.1 Condition System Concepts | Common Lisp (New) Language ..., accessed May 31, 2025, https://lisp-docs.github.io/cl-language-reference/chap-9/j-b-condition-system-concepts
18. Exception handling in Objective-C | GeeksforGeeks, accessed May 31, 2025, https://www.geeksforgeeks.org/exception-handling-in-objective-c/
19. Error Handling in Objective-C | GeeksforGeeks, accessed May 31, 2025, https://www.geeksforgeeks.org/error-handling-in-objective-c/
20. Exception handling - Wikipedia, accessed May 31, 2025, https://en.wikipedia.org/wiki/Exception_handling
21. General Exception and Error Handling Best Practices for Compiled Languages : r/ProgrammingLanguages - Reddit, accessed May 31, 2025,

https://www.reddit.com/r/ProgrammingLanguages/comments/1izfbi7/general_exception_and_error_handling_best/
22. io tutorial, accessed May 31, 2025, https://iolanguage.org/tutorial.html
23. io reference, accessed May 31, 2025, https://iolanguage.org/reference/
24. Justified Programming — Reason Parameters That Answer “Why ..., accessed May 31, 2025, https://youtu.be/OrQ9swvm_VA
25. Justifying a Software Development Project - Ambysoft Inc., accessed May 31, 2025, https://ambysoft.com/essays/projectjustification.html
26. How do you justify more code being written by following clean code practices?, accessed May 31, 2025, https://softwareengineering.stackexchange.com/questions/388802/how-do-you-justify-more-code-being-written-by-following-clean-code-practices
27. Justified Programming — Reason Parameters That Answer “Why”, by bisqwit - Reddit, accessed May 31, 2025, https://www.reddit.com/r/programming/comments/6swde7/justified_programming_reason_parameters_that/
28. Obfuscated C programs: Introduction - YouTube, accessed May 31, 2025, https://m.youtube.com/watch?v=rwOI1biZeD8&pp=ygULI2NvbnRlc3RlbmM%3D
29. REBOL: The Official Guide (Book/CD Package): Goldman, Elan, Blanton, John - Amazon.com, accessed May 31, 2025, https://www.amazon.com/REBOL-Official-Guide-Book-Package/dp/007212279X
30. PeterOvermann/TriadicMemory: Cognitive Computing with ... - GitHub, accessed May 31, 2025, https://github.com/PeterOvermann/TriadicMemory
31. Triadic Memory — A Fundamental Algorithm for Cognitive ..., accessed May 31, 2025, https://discourse.numenta.org/t/triadic-memory-a-fundamental-algorithm-for-cognitive-computing/9763
32. Triadic Memory — A Fundamental Algorithm for Cognitive Computing - #8 by cezar_t - Related Papers - HTM Forum, accessed May 31, 2025, https://discourse.numenta.org/t/triadic-memory-a-fundamental-algorithm-for-cognitive-computing/9763/8
33. mem0ai/mem0: Memory for AI Agents; SOTA in AI Agent ... - GitHub, accessed May 31, 2025, https://github.com/mem0ai/mem0
34. Designing better file organization around tags, not hierarchies, accessed May 31, 2025, https://www.nayuki.io/page/designing-better-file-organization-around-tags-not-hierarchies
35. Be File System - Wikipedia, accessed May 31, 2025, https://en.wikipedia.org/wiki/Be_File_System
36. Metadata/database filesystem - OS - Haiku Community, accessed May 31, 2025, https://discuss.haiku-os.org/t/metadata-database-filesystem/2445
37. Semantic file system - Wikipedia, accessed May 31, 2025, https://en.wikipedia.org/wiki/Semantic_file_system
38. Migration to Package Management - Haiku OS, accessed May 31, 2025, https://www.haiku-os.org/docs/develop/packages/Migration.html

39. Plan 9: Not (Only) A Better UNIX | PPT - SlideShare, accessed May 31, 2025, https://www.slideshare.net/slideshow/plan-9-not-only-a-better-unix/15381711
40. css.csail.mit.edu, accessed May 31, 2025, https://css.csail.mit.edu/6.824/2014/papers/plan9.pdf
41. Alexander Obenauer, accessed May 31, 2025, https://alexanderobenauer.com/
42. WonderOS, accessed May 31, 2025, https://wonderos.org/
43. Future (desktop) operating systems — a collection of inspirations - the stream, accessed May 31, 2025, https://stream.thesephist.com/updates/1640102564
44. OLLOS - Alexander Obenauer, accessed May 31, 2025, https://alexanderobenauer.com/ollos/
45. XXIIVV — devlog, accessed May 31, 2025, https://wiki.xxiivv.com/site/devlog.html
46. Sparse Distributed Representations - Numenta, accessed May 31, 2025, https://www.numenta.com/assets/pdf/biological-and-machine-intelligence/BaMI-SDR.pdf
47. HTM School | Numenta, accessed May 31, 2025, https://www.numenta.com/resources/htm/htmschool/
48. Encoders - Numenta, accessed May 31, 2025, https://www.numenta.com/assets/pdf/biological-and-machine-intelligence/BaMI-Encoders.pdf
49. Sparse Distributed Representations - Sean's Blog, accessed May 31, 2025, https://seanpedersen.github.io/posts/sparse-distributed-representations
50. arxiv.org, accessed May 31, 2025, https://arxiv.org/pdf/2502.14902
51. OSU-NLP-Group/HippoRAG: [NeurIPS'24] HippoRAG is a ... - GitHub, accessed May 31, 2025, https://github.com/OSU-NLP-Group/HippoRAG
52. [2502.14802] From RAG to Memory: Non-Parametric Continual Learning for Large Language Models - arXiv, accessed May 31, 2025, https://arxiv.org/abs/2502.14802
53. aclanthology.org, accessed May 31, 2025, https://aclanthology.org/2025.neusymbridge-1.11.pdf
54. accessed December 31, 1969, https://arxiv.org/pdf/2412.05967v1
55. arxiv.org, accessed May 31, 2025, https://arxiv.org/html/2412.05967v1
56. [Papierüberprüfung] Language hooks: a modular framework for augmenting LLM reasoning that decouples tool usage from the model and its prompt - Moonlight | AI Colleague for Research Papers, accessed May 31, 2025, https://www.themoonlight.io/de/review/language-hooks-a-modular-framework-for-augmenting-llm-reasoning-that-decouples-tool-usage-from-the-model-and-its-prompt
57. accessed December 31, 1969, https://arxiv.org/pdf/2405.06907v1
58. arxiv.org, accessed May 31, 2025, https://arxiv.org/html/2405.06907v1
59. arXiv:2502.14345v1 [cs.AI] 20 Feb 2025, accessed May 31, 2025, https://arxiv.org/pdf/2502.14345?
60. CaMeL offers a promising new direction for mitigating prompt ..., accessed May 31, 2025, https://simonwillison.net/2025/Apr/11/camel/
61. Operationalizing CaMeL: Strengthening LLM Defenses for Enterprise Deployment - arXiv, accessed May 31, 2025, https://arxiv.org/html/2505.22852v1
62. arxiv.org, accessed May 31, 2025, https://arxiv.org/pdf/2503.18813

63. www.arxiv.org, accessed May 31, 2025, https://www.arxiv.org/pdf/2505.22852
64. Defeating Prompt Injections by Design - ResearchGate, accessed May 31, 2025, https://www.researchgate.net/publication/390176637_Defeating_Prompt_Injections_by_Design/fulltext/67e374dc72f7f37c3e8e75df/Defeating-Prompt-Injections-by-Design.pdf?origin=scientificContributions
65. [2502.11896] CAMEL: Continuous Action Masking Enabled by Large Language Models for Reinforcement Learning - arXiv, accessed May 31, 2025, https://arxiv.org/abs/2502.11896
66. Beyond Single-Turn: A Survey on Multi-Turn Interactions with Large Language Models, accessed May 31, 2025, https://arxiv.org/html/2504.04717v1
67. Abstract - Nicholas Carlini, accessed May 31, 2025, https://nicholas.carlini.com/writing/2019/advex_papers_with_abstracts.txt
68. [2505.22852] Operationalizing CaMeL: Strengthening LLM Defenses for Enterprise Deployment - arXiv, accessed May 31, 2025, http://arxiv.org/abs/2505.22852
69. [2503.18813] Defeating Prompt Injections by Design - arXiv, accessed May 31, 2025, https://arxiv.org/abs/2503.18813
70. Defeating Prompt Injections by Design - arXiv, accessed May 31, 2025, https://arxiv.org/pdf/2503.18813?