Library Learning for LLM Agents: A Survey of Skill Acquisition Through Growing Toolboxes

Nikhil Togadiya, Tobias Sesterhenn and Christian Bartelt

[1*]CORE, Clausthal University of Technology, Clausthal-Zellerfeld, Germany.

*Corresponding author(s). E-mail(s): nikhil.togadiya@tu-clausthal.de;
Contributing authors: tobias.sesterhenn@tu-clausthal.de;
christian.bartelt@tu-clausthal.de;

Abstract

Large Language Model (LLM) agents augmented with tools have achieved promising results on complex tasks, but their tool-use capabilities are typically static limited to a fixed set of predefined tools. This static paradigm hampers adaptability: agents cannot learn new skills beyond their initial programming, making it difficult to generalize to open-ended scenarios. Library learning offers a paradigm shift: instead of one-shot tool use, agents continually discover new skills, retain them in a growing toolbox, reuse them on future tasks, and actively manage their skill library over time. This survey introduces library learning which is organized around four pillars: skill discovery, retention, reuse, and management; as the foundation for lifelong tool augmentation. Covering 11 frameworks published between 2023 and 2025 across five domains (web, code, games, theorem proving, robotics), we review mechanisms including autonomous code synthesis, vector-store memory, self-refinement, and library pruning. By comparing these approaches, we identify current limitations of static tool usage and survey how library-learning agents achieve skill acquisition and continual improvement. The survey's contributions include a novel taxonomy of library learning mechanisms, a critical comparative analysis across systems, and an outline of open research challenges, guiding the design of more adaptive and generalist LLM agents.

Keywords: Large language model agents, tool use, library learning, growing toolboxes, skill acquisition, skill discovery, skill retrieval, skill reuse

1 Introduction

1.1 Motivation: From Single-Shot Tool Use to Lifelong Capable LLM Agents

The trajectory of artificial intelligence, particularly within the domain of Large Language Models (LLMs), has undergone a fundamental shift from static information processing to dynamic, agentic interaction. Early *tool-use* paradigms where models invoked calculators or search engines via single-shot prompting have represented a significant advance[1]. However, this paradigm is inherently bounded by the context window horizon[2]. In traditional tool-use architectures, the agent's capabilities are defined exclusively by the tools provided in the system prompt at the moment of inference. The agent enters every task without procedural knowledge of prior interactions; it may know that a tool exists, but it retains no memory of *how* it successfully combined tools to solve a complex problem five minutes ago.

This limitation can become a bottleneck in long-horizon, open-ended environments such as software engineering, scientific discovery, or embodied control in games like Minecraft. Consider a software engineering agent tasked with refactoring a legacy codebase[3]. In a static tool-use paradigm, the agent may need to re-derive logic for navigating the file system, identifying dependencies, and running unit tests across repeated sub-tasks. It exhibits a functional form of catastrophic forgetting not of its pre-trained weights, but of its experiential history, with limited ability to build on prior successful workflows.

The emergence of **Library-Learning Approaches** addresses this fundamental deficiency. By shifting the agent's architecture from a stateless request-response model to a stateful, accumulative learning system, we enable the transition from "tool using" to "tool making." Agents equipped with growing toolboxes can define, implement, verify, and store their own tools. This capability unlocks **compositionality**, where complex behaviors are abstracted into reliable primitives (e.g., a debug_module() function) that can be invoked as atomic actions in future planning [4]. It introduces **cost efficiency**, allowing expensive "reasoning" models to distill their intelligence into lightweight code that cheaper models can execute(a pattern also seen in multi-agent software development[3]), drastically reducing the marginal cost of intelligence [5]. Furthermore, it enhances **robustness** by replacing probabilistic token generation with deterministic code execution for known tasks [6], reducing the surface area for hallucination and error [7].

1.2 Research Gap and Problem Statement

Despite the transformative potential of library learning, the current research landscape is fragmented and lacks a unified theoretical framework. Innovations are isolated within specific domains. The core problem is the absence of a standardized methodology for the lifecycle of a tool. How does an agent strictly differentiate between a one-off action and a reusable skill (Discovery)? How should these skills be represented as code, natural language narratives, or formal logic to maximize retrieval accuracy (Retention)? How can an agent navigate a library that grows from ten

tools to ten thousand without succumbing to the "retrieval bottleneck" or context saturation (Management)[8]? Current literature offers divergent answers to these questions, often without cross-pollination. For instance, TROVE implements aggressive logarithmic pruning to maintain efficiency [9], while LEGO-Prover advocates for a monotonically growing "forest" of knowledge [10]. This divergence highlights a tension between *archival* philosophies (keeping all knowledge) and *engineering* philosophies (optimizing for utility), creating a need for a comparative synthesis that critiques these architectural choices against the requirements of scalability, safety, and generalizability.

To structure our analysis and enable a systematic comparison across systems, we define the following four research questions

1.3 Research Questions

To synthesize a coherent understanding of this emerging field, this paper addresses the following research questions:

1. **Mechanistic Taxonomy:** What are the distinct mechanisms by which state-of-the-art agents discover, verify, and persist new skills? How do code-centric approaches differ from experience-centric approaches in terms of transferability and inference cost?

2. **Lifecycle Dynamics:** How do agents manage the growing complexity of their toolboxes? What are the trade-offs between automated pruning strategies (e.g., frequency-based trimming) and agentic maintenance (e.g., explicit CRUD operations)?

3. **Generalization Limits:** To what extent do self-generated tools generalize to out-of-distribution tasks, and how does the mode of discovery whether via social imitation, curriculum-driven exploration, or document mining impact this generalization?

4. **Safety & Robustness:** Given the ability of these agents to execute arbitrary code and modify their own capabilities, what verification protocols exist to prevent the propagation of malicious [11], incorrect, or "poisoned" tools within the library [12]?

1.4 Contributions

This paper provides a comprehensive survey and critical assessment of library-learning agents, offering:

1. **A Process-Oriented Taxonomy:** We structure the field by the **Library Learning Cycle**: *Discovery* $\rightarrow$ *Retention* $\rightarrow$ *Reuse* $\rightarrow$ *Management*. This framework reveals convergent evolution across disparate fields (e.g., how math provers and game agents both independently arrived at evolutionary skill generalization).

2. **Deep Comparative Analysis:** We contrast 11 seminal papers, including a direct comparison of MINDSTORES' *mental model* approach against Voyager's *procedural skill* approach, highlighting trade-offs between semantic flexibility and execution speed [13, 14].

3. **Critical Assessment of Safety:** We identify important gaps in current safety practice for self-modifying agents. We analyze limitations in existing verification and isolation mechanisms (for example, execution-proxy checks versus stronger sandboxing/formal methods) and outline a roadmap for safer library-learning systems [5, 10, 15, 16].

The remainder of this paper is organized as follows. Section 2 provides background on tool-calling agents and memory architectures. Section 3 describes our review methodology and selection criteria. Section 4 presents the core analysis organized around the four pillars of library learning. Section 5 offers a critical assessment across quality, evaluation, scalability, and safety dimensions. Section 6 identifies open research gaps, and Section 7 concludes.

2 Background & Core Concepts

Before analyzing specific implementations, it is essential to define the core concepts that underpin tool-calling agents and library learning. This shared vocabulary allows us to draw parallels between seemingly unrelated systems, such as a theorem prover and a Minecraft bot.

2.1 Tool-Calling LLM Agents

The foundation of this field is the tool-calling agent an LLM wrapper designed to overcome the static nature of pre-trained weights.

Planner-Executor Loops: The standard architecture, often termed the ReAct (Reasoning and Acting) loop, involves a cyclical process (Figure 1)[17]. The **Planner** (LLM) perceives the state and generates a thought trace[18] and a tool call [19](e.g., lookup_weather(city="London")). The **Executor** (a Python runtime, API interface, or game engine) executes this call and returns the observation. The Planner then integrates this observation to generate the next step. This loop transforms the LLM from a text generator into a decision-making engine.

Reflection and Self-Correction: Advanced agents incorporate reflection, a metacognitive step where the LLM critiques its own outputs[20]. If a tool execution fails (e.g., a SyntaxError in generated code), the reflection module analyzes the error trace and prompts the Planner to generate a corrected version. This is distinct from simple retrying; it involves semantic analysis of *why* the failure occurred.

2.2 Definitions and Terminology

To keep terminology consistent across domains, Table 1 defines the core terms used throughout this survey. We use a strict separation between *representation* (function, abstraction), *capability interface* (tool), and *competence level* (skill).

These are *operational definitions* for this survey, synthesized from recurring usage patterns in tool-use and library-learning systems rather than tied to a single canonical source [4, 9, 10, 13, 14, 17, 21–24].

Table 1: Formal Definitions of Core Library-Learning Terms

Term	Definition	Operational Role in This Survey
Function	A concrete executable mapping from inputs to outputs (for example, Python/JavaScript code, or a proof procedure in a formal system).	Atomic implementation unit that can be composed, wrapped, tested, and versioned.
Tool	A callable interface exposed to the planner/executor loop (for example API endpoint, wrapped function, simulator action, or service method).	Operational action interface used at inference time; one tool may internally call one or multiple functions.
Skill	A reusable capability that improves task performance across instances; it is an agent-level competence, not necessarily a single callable object.	Primary unit of analysis in this survey's lifecycle (discovery, retention, reuse, management).
Abstraction	A generalized representation that compresses recurring low-level solution patterns into reusable higher-level structure.	Transfer mechanism that increases reuse and reduces planning or synthesis complexity (for example helper routines, generalized lemmas, templates).
Library	A persistent memory store of learned artifacts and metadata (for example code/lemmas/experiences plus descriptions, embeddings, tests, provenance).	Long-horizon knowledge base that supports retrieval, verification, and maintenance.
Toolbox	The active set of callable tools available to the agent in the current run or context window.	Runtime working set; can be a filtered subset of the library and can also include externally provided tools.

Table 1: Formal Definitions of Core Library-Learning Terms

Term	Definition	Operational Role in This Survey
Retrieval	The process of selecting relevant artifacts from memory using lexical matching, dense embeddings, symbolic indices, or hybrid search.	Gate between retention and reuse; directly affects inference cost, context budget, and downstream success.
Verification	Any mechanism that checks candidate artifacts before or after use (execution tests, static analysis, self-consistency, or formal proof checking).	Primary reliability control for preventing incorrect or unsafe skills from entering or remaining in the library.
Pruning	Deliberate removal of low-utility, redundant, or stale artifacts based on explicit criteria (for example frequency or quality scores).	Core management operation for controlling library growth and mitigating retrieval noise.
Refinement	Post-hoc improvement of existing artifacts through editing, generalization, or parameterization after observing failures or near-misses.	Management mechanism for improving transfer without full re-discovery.
Compositional Reuse	Solving a task by orchestrating multiple retrieved artifacts instead of invoking a single artifact directly.	Key reuse pattern in complex tasks where sequencing and dependency resolution dominate performance.
Provenance	Traceable origin information for an artifact (creator, source task, source agent, environment, and validation history).	Foundation for trust, auditability, and safety analysis, especially in socially learned or self-modifying systems.

2.3 Memory Architectures

Procedural Memory: Stores *how* to perform actions. In systems like CRADLE and Voyager, this is a library of executable functions. It is analogous to muscle memory or rote skills [14, 24].

Episodic Memory: Stores *what happened*. Systems like MINDSTORES use this to retain narrative tuples of (State, Task, Plan, Outcome). This allows the agent to recall specific past experiences ("I tried to mine iron with a wood pickaxe and failed")

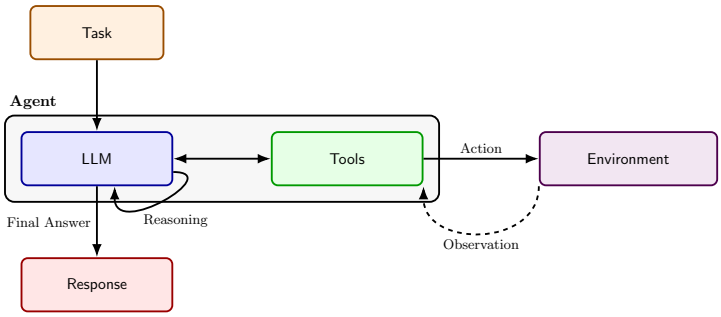

Fig. 1 The ReAct-style planner-executor loop: the planner (LLM) alternates between reasoning and tool calls, while the executor runs actions and returns observations. Adapted from [17].

to inform future planning, effectively building a "mental model" of the world dynamics [13, 25].

Semantic Memory: Stores generalized facts and beliefs[21]. In MindForge, semantic memory retains world knowledge (e.g., "Dirt can be mined by hand") that is distinct from specific episodes or executable skills, often updated through social interaction [15].

3 Review Methodology

3.1 Overview of Selected Literature

This paper surveys 11 key papers that represent the frontier of library-learning agents.

CRADLE (Tan et al., 2024): A framework for General Computer Control (GCC) that enables agents to interact with any software via screenshots and mouse/keyboard. It builds a procedural memory of Python skills to master complex games like Red Dead Redemption 2 [24].

LATM (Cai et al., 2024): "LLMs As Tool Makers" proposes a two-tier tool-learning architecture as "Tool Maker" that uses a high-capacity model (GPT-4) once to build tools and "Tool User" which uses a lightweight model (GPT-3.5) for repeated tool usage, reducing inference cost while maintaining performance [22].

LEGO-Prover (Wang et al., 2023): A neural theorem prover that evolves a library of verified Isabelle lemmas, using a modular "block-by-block" approach to solve complex proofs [10].

MindForge (Lică et al., 2025): An embodied agent framework extending Voyager with "Theory of Mind," allowing agents to learn skills socially via chat and observation rather than solely through trial-and-error [15].

MINDSTORES (Chari et al., 2025): A planning framework that builds a persistent "mental model" via an experience database of (state, task, plan, outcome) tuples, enabling outcome prediction and plan refinement [13].

ReGAL (Stengel-Eskin et al., 2024): "Refactoring for Generalizable Abstraction Learning" focuses on gradient-free learning by refactoring primitive programs into shared abstractions, maintaining a "Code Bank" that is actively pruned [4].

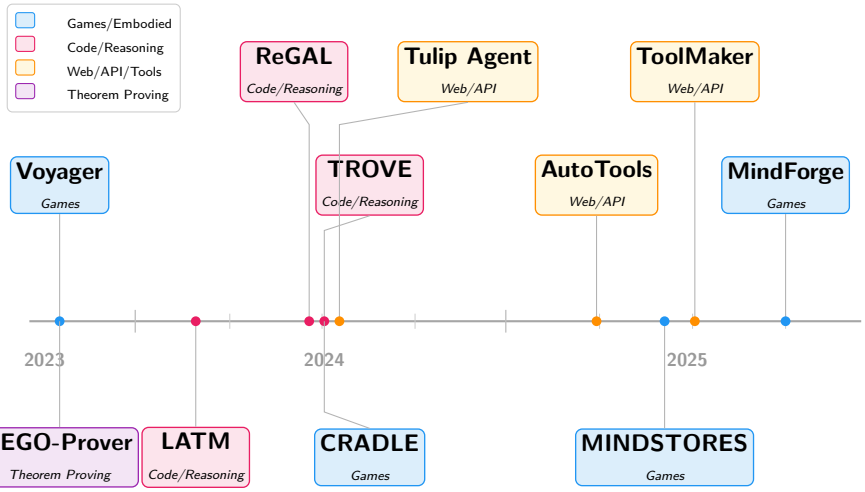

Fig. 2 Chronological timeline of the 11 surveyed library-learning frameworks (2023–2025), color-coded by application domain. The concentration of publications in mid-2024 reflects the rapid acceleration of research in this area.

AutoTools (Shi et al., 2025): A framework that automates the transformation of textual tool documentation into executable, verifiable Python functions, bridging the gap between static docs and agent actions [16].

ToolMaker (Wölflein et al., 2025): An agentic framework that converts scientific code repositories into LLM-compatible tools by autonomously handling dependency installation, environment setup, and unit testing within sandboxes [5].

TROVE (Wang et al., 2024): A method for inducing verifiable toolboxes that employs "agreement-based selection" and strict periodic pruning based on usage frequency to maintain efficiency [9].

Tulip Agent (Ocker et al., 2024): An architecture enabling agents to perform explicit CRUD (Create, Read, Update, Delete) operations on their own tool libraries, supporting recursive search and dynamic modification [23].

Voyager (Wang et al., 2023): The pioneer of embodied lifelong learning, using an automatic curriculum to drive the discovery of JavaScript skills in Minecraft, validated by execution feedback [14].

Figure 2 shows the chronological progression of these 11 frameworks from 2023 to 2025, color-coded by application domain.

3.2 Clustering Approach

The methodology for this review clusters papers by their primary contribution to the **Library Learning Cycle** rather than by application domain. This reveals the structural isomorphisms between different fields showing, for instance, how the *deduplication* logic in mathematical proving (LEGO-Prover) mirrors the *refactoring* logic in code synthesis (ReGAL).

Skill Discovery (Section 4.1): Contrasts environment-driven discovery (Voyager, CRADLE), code-centric refactoring (ReGAL, TROVE), repository mining (AutoTools, ToolMaker), and task-driven synthesis (LATM, Tulip).

Skill Retention (Section 4.2): Analyzes the storage substrates, comparing procedural code libraries (AutoTools, Tulip) with narrative experience archives (MINDSTORES).

Skill Reuse (Section 4.3): Examines how skills are retrieved and applied, from simple vector lookup to the complex "Dispatcher" routing in LATM.

Library Management (Section 4.4): Investigates the lifecycle strategies, contrasting monotonic growth (LEGO-Prover) with active pruning (TROVE).

Table 2 provides an overview of how the 11 surveyed papers are distributed across the four dimensions of library learning, revealing that most systems focus on discovery and retention, while fewer address comprehensive management strategies.

Table 2: Main Clusters Overview: Distribution of Papers Across Library Learning Dimensions

Paper	Skill Discovery	Skill Retention	Skill Reuse	Library Management
Voyager	Environment-Driven Discovery	Procedural Code Libraries	Atomic Reuse via Direct Retrieval	–
CRADLE	Environment-Driven Discovery	Procedural Code Libraries	Adaptive Reuse via Parameterization	–
Mind-Forge	Environment-Driven Discovery	Multi-Memory Architectures	–	–
ReGAL	Code-Centric Discovery	Dual-Bank Tool Systems	Compositional Reuse via Programmatic Chaining	Engineering Paradigm
TROVE	Code-Centric Discovery	–	Atomic Reuse via Selection and Routing	Engineering Paradigm
Auto-Tools	Repository Mining	Dual-Bank Tool Systems	Compositional Reuse via Programmatic Chaining	–
TOOL-MAKER	Repository Mining	Dual-Bank Tool Systems	–	–
LATM	Task-Driven Discovery	Dual-Bank Tool Systems	Atomic Reuse via Selection and Routing	–

Table 2: Main Clusters Overview: Distribution of Papers Across Library Learning Dimensions

Paper	Skill Discovery	Skill Retention	Skill Reuse	Library Management
Tulip Agent	Task-Driven Discovery	Procedural Code Libraries	Atomic Reuse via Direct Retrieval	Interactive Paradigm
LEGO-Prover	Formal Verification-Driven Discovery	Procedural Code Libraries	Hybrid Adaptive Reuse via Imitation	Archival Paradigm
MIND-STORES	–	Experience Archives	–	–

Note: "–" indicates that the corresponding dimension was not a primary architectural contribution of the system, not necessarily that the mechanism is absent.

4 Library Learning Systems in Literature

This chapter constitutes the core of the survey, providing a detailed synthesis of the mechanisms employed by the surveyed systems. Figure 3 provides a hierarchical overview of the taxonomy, showing all four pillars and their sub-categories.

4.1 Skill Discovery Mechanisms

Following the terminology in Table 1, this subsection compares how systems operationalize skill discovery in practice. The literature presents four distinct paradigms.

4.1.1 Environment-Driven Discovery

In embodied settings, discovery is often a product of interaction loops where the environment provides the feedback signal for what constitutes a "skill."

Direct UI Exploration (CRADLE): CRADLE operates in the visually complex domain of General Computer Control (GCC). Its discovery mechanism is multimodal and often **instruction-based**. The agent uses OCR and visual grounding (via the Segment Anything Model, SAM) to "read" on-screen tutorials or tooltips (e.g., "Hold TAB to open weapon wheel"). It then translates these visual cues into executable Python skills (e.g., open_weapon_wheel()). This represents a form of *reactive discovery*, where the agent learns by perceiving the environment's affordances directly. The discovery is validated by execution success if the menu opens, the skill is saved [24].

Game World Exploration (Voyager): Voyager employs an **Automatic Curriculum** generated by GPT-4 to propose progressively harder tasks (e.g., "mine wood" → "craft table") within the open-ended Minecraft environment[26]. Discovery

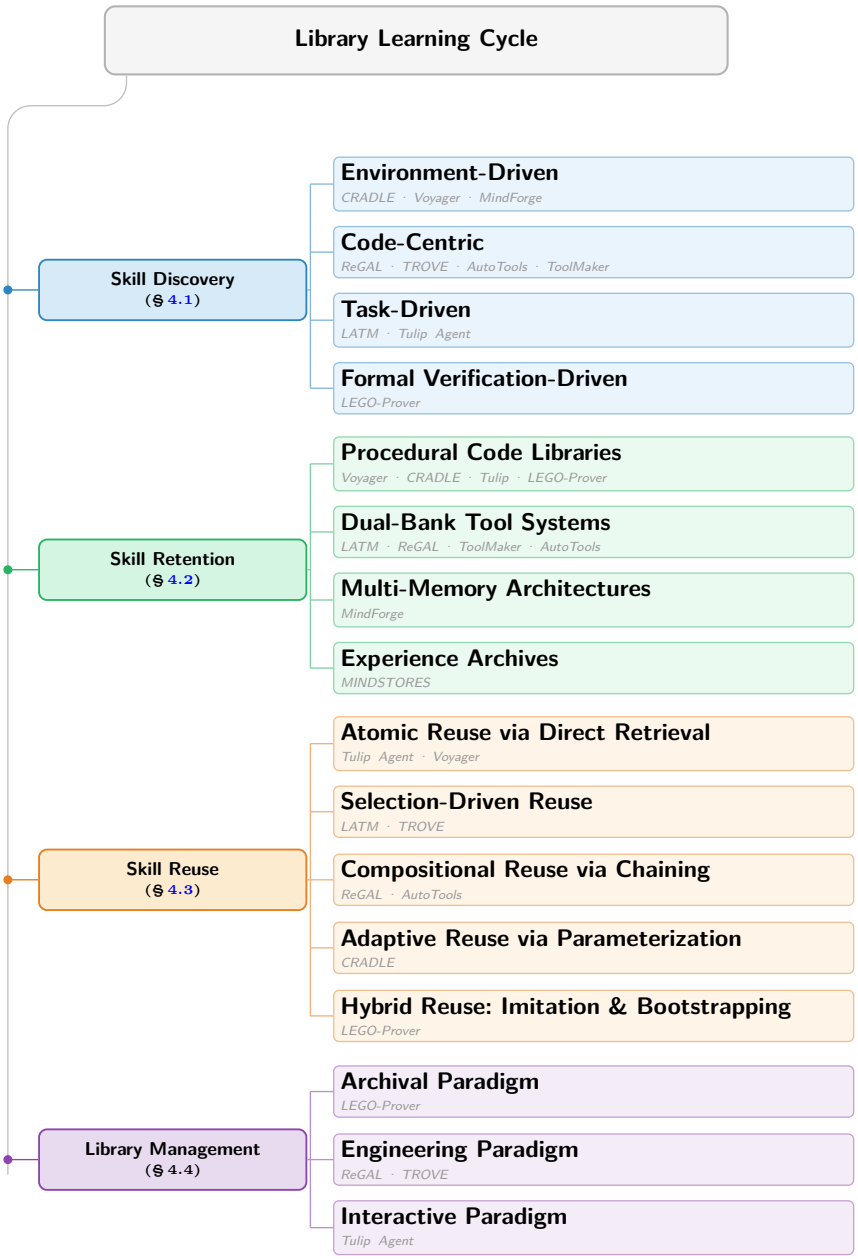

Fig. 3 Taxonomy of Library Learning mechanisms organized around the four pillars of the Library Learning Cycle. Each pillar decomposes into sub-categories; corresponding surveyed frameworks are listed in italics.

is driven by an **Iterative Prompting Mechanism**: the agent generates code, executes it, and refines it based on error traces or success signals. A skill is "discovered" only when the code successfully satisfies the curriculum goal as verified by a Critic module[27]. This is a "survival of the fittest" model where only functional code that solves a novel problem is crystallized into the library [14].

Socially-Mediated Exploration (MindForge): MindForge introduces a radical shift by decoupling discovery from individual trial-and-error. When a MindForge agent fails a task (a common occurrence for open-weight models), it initiates a **Communication Round**. It broadcasts a help request, and a more capable peer (or expert) provides the code or explanation. Here, discovery is **socially mediated**: the skill is acquired through cultural transmission. This allows agents to leapfrog the exploration phase by "downloading" knowledge from the collective, mimicking human social learning [15, 25].

4.1.2 Code-Centric Discovery

These systems operate on the code itself, discovering skills by identifying patterns and abstractions within existing programs.While earlier works demonstrated self-supervised tool use[1], these systems extend that by explicit refactoring.

Refactoring-Based Abstraction (ReGAL): ReGAL's discovery is intrinsic and bottom-up. It analyzes batches of "primitive programs" (solutions to specific tasks) and prompts an LLM to identify common sub-routines. If multiple programs contain logic to draw a polygon, ReGAL extracts this into a draw_polygon() helper function[28]. This process does not require new external tasks; it optimizes the agent's internal representation of knowledge, making future synthesis more efficient [4].

Streaming Composition (TROVE): TROVE processes a stream of tasks and attempts to solve them using three modes: *Import* (reuse), *Create* (new), or *Skip* (primitives). A skill is "discovered" (created) only if the *Create* mode solution is verified as correct via **self-consistency voting** and is determined to be simpler (lower AST depth) than the primitive solution. This ensures that the library only grows when a new abstraction provides a concrete efficiency gain [9].

Repository Mining (ToolMaker & AutoTools): These systems view "discovery" as the ingestion of external knowledge bases, addressing the challenge of navigating massive API datasets [8]. **AutoTools** parses raw API documentation to synthesize callable functions[16]. **ToolMaker** takes a more aggressive approach: given a GitHub URL, it autonomously navigates the repository, identifies dependencies, installs them within a Docker container, and wraps the codebase into an LLM-compatible tool. Here, discovery is the automated bridging of the gap between static code repositories and agentic capability [5].

4.1.3 Task-Driven Discovery

In this paradigm, skills are created reactively to satisfy specific user requests that cannot be met by the current toolbox.

Synthesis from Examples (LATM): LATM employs a **Dispatcher** that monitors incoming requests. When a "cache miss" occurs (i.e., a new type of task arrives), it triggers the **Tool Maker** (GPT-4). The Tool Maker uses **Programming by**

Example (PbE), taking a few instances of the task and synthesizing a generic Python function to solve them. Discovery is thus explicitly demand-driven; tools are manufactured only when a market (user demand) exists for them [22].

On-Demand Creation (Tulip Agent): The Tulip Agent features an **Auto-TulipAgent** mode where the agent itself decides when to create a tool. If a search of the tool library yields no results for a necessary sub-task, the agent uses a Create meta-tool to synthesize a new function on the fly. This treats discovery as an action within the agent's decision space, democratizing the creation process [23].

4.1.4 Formal Verification-Driven Discovery

Mathematical Lemma Synthesis (LEGO-Prover): In formal theorem proving, discovery is bifurcated. The **Prover** discovers problem-specific lemmas while attempting to solve a theorem (passive discovery). Crucially, the **Evolver** performs active discovery by applying **Directional Transformations** to existing skills generalizing them (e.g., replacing constants with variables, scaling complexity) to create new, abstract lemmas. These are formally verified by Isabelle. If they pass, they are added to the library. This allows the system to discover mathematical truths that were not explicitly requested but expand the agent's deductive reach [10, 29]

4.2 Skill Retention and Persistence Mechanisms

Consistent with Table 1, this subsection focuses on retention as an implementation problem: what artifacts are persisted, in which memory substrate, and under what validation constraints. We survey persistence strategies through a pattern-based taxonomy, distinguishing between monolithic repositories and dynamically managed systems.

Static Monotonic Systems

Static architectures retain verified skills indefinitely *without active forgetting or pruning*. These systems are characterized by append-only growth and permanent storage post-verification.

4.2.1 Procedural Code Libraries.

The predominant pattern employs a single vector-indexed repository storing executable functions. Skills are represented as code artifacts (typically Python or JavaScript) paired with semantic embeddings generated from natural language descriptions. The retention pipeline follows a three-stage verification process: execution validation, documentation generation, and vector indexing.

Voyager [14] implements this paradigm in the Minecraft domain, storing successfully executed Mineflayer scripts in a skill library indexed by GPT-3.5-generated descriptions embedded via text-embedding-ada-002. A dedicated critic module performs self-verification by inspecting environmental states (inventory, health) before committing skills. CRADLE [24] extends this pattern to general computer control, persisting Python functions wrapped in an `io_env` class and indexed by documentation

embeddings. Notably, CRADLE enforces parametric flexibility, retaining functions with configurable arguments (e.g., `move_up(duration=1)`) to enhance reusability.

Verification strictness varies across implementations. AutoTools [16] employs a two-stage filter: syntax parsing validates parameter-document alignment, while integration testing executes functions against prerequisite dependencies retrieved from cache. Tulip Agent [23] achieves dynamic persistence through runtime introspection, parsing Sphinx docstrings and type hints to automate library indexing without manual curation. The system maintains a `ToolLookup` dictionary mapping function identifiers to callable objects, enabling hot-loading of newly generated skills during execution.

A specialized variant within this cluster, LEGO-Prover [10], integrates formal verification. Before retention, the *Isabelle theorem prover* validates proof correctness and integrity by rejecting incomplete proofs containing `sorry` or `oops` keywords. Deduplication is enforced via the SequenceMatcher algorithm; skills exceeding a similarity threshold of 0.85 against existing entries are discarded. Despite this additional rigor, the architecture remains monotonic, accumulating 22,532 verified lemmas without pruning.

Table 3: Comparison of Verification Mechanisms in Procedural Code Libraries

Framework	**Verification Method**	**Deduplication Strategy**	**Embeddings Model**
Voyager	Self-verification (critic agent)	None (append-only)	text-embedding-ada-002
CRADLE	Execution-based self-verification	None (append-only)	text-embedding-ada-002
AutoTools	Syntax parsing + integration testing	None (append-only)	Not specified
Tulip	AST validation	None (append-only)	text-embedding-3-large
LEGO-Prover	Isabelle theorem proving	SequenceMatcher ($\theta = 0.85$)	text-embedding-ada-002

4.2.2 Dual-Bank Tool Systems.

This pattern bifurcates retention into complementary repositories: a *Code Bank* storing function definitions and a *Demo Bank* preserving validated usage examples. This separation amortizes creation costs by enabling cheaper models to reuse skills via in-context learning.

LATM [22] crystallizes this design, where a **Tool Maker** (GPT-4) generates generic functions and converts unit tests into demonstrations. These wrapped tools are persisted in a functional cache indexed by task name and API signature (e.g., `logical_deduction_five_objects`). Subsequent requests from a **Tool User** (GPT-3.5) retrieve both function and demonstrations, eliminating documentation reprocessing. Similarly, ReGAL [4] maintains a **Code Bank** "C" of helper functions and a **Demo Bank** "D" of refactored programs, but imposes stricter verification: a

skill is retained only if its refactored execution output matches the original primitive program exactly, i.e., $\hat{p}() = p()$.

ToolMaker [5] extends this pattern by persisting execution environments alongside code. The system records write actions A_w (bash commands, file modifications) during repository setup to generate reproducible environment definitions (Dockerfiles). This dual-pillar approach function logic plus environment snapshot ensures skills remain executable across heterogeneous systems, addressing dependency drift challenges inherent in long-term retention.

4.2.3 Multi-Memory Architectures.

MindForge [15] exemplifies explicit ontological separation, partitioning retention across three subsystems. Procedural Memory stores reusable code identical to Cluster 1 architectures. Episodic Memory employs RAG to retain interaction traces, embedding episodes via text-embedding-ada-002 and retrieving top-k matches (typically $k = 5$) for summarization. Semantic Memory maintains dynamic beliefs about world mechanics and partner mental states, updated through communication and perspective-taking. This architecture decouples skill permanence (procedural) from experiential refinement (episodic) and abstract knowledge (semantic), enabling sophisticated Theory-of-Mind reasoning while preserving executable capabilities.

4.2.4 Experience Archives.

MINDSTORES [13] deviates fundamentally by retaining non-executable interaction narratives. Each experience is stored as a tuple (s, t, p, o), where s represents environmental state, t denotes the task, p captures the action plan, and o records execution outcomes including failures. The system employs a Sentence-BERT model (all-MiniLM-L6-v2) to generate 768-dimensional L2-normalized embeddings, stored in a FAISS IndexFlatL2 for efficient similarity search. Notably, MINDSTORES intentionally avoids verification entirely, accumulating all experiences without deletion: transitioning from a "Wooden Door" task to a "Minecart" task expands the database from 26 to 355 entries without pruning. This passive retention strategy builds a comprehensive failure model but incurs linearly scaling computational overhead, a critical limitation discussed further in Section 4.4 on Library Management.

Dynamic Adaptive Systems

Dynamic systems actively manage library composition post-retention, implementing pruning or refinement mechanisms to maintain quality and efficiency.

4.3 Skill Reuse Mechanisms

Using the same terminology baseline (Table 1), we analyze reuse in terms of retrieval and application dynamics under different planning architectures.

4.3.1 Atomic Reuse via Direct Retrieval and Invocation

Atomic reuse represents the simplest paradigm: skills are retrieved via *semantic search* and directly executed without modification. This mechanism relies exclusively on

dense vector representations stored in skill libraries, enabling one-step finding and application.

Tulip Agent instantiates this architecture through a retrieval-augmented generation (RAG) framework for executable functions. The skill library is implemented as a vector store (ChromaDB) where function descriptions, extracted via introspection, are embedded using `text-embedding-3-small`. Finding occurs through subtask-level semantic search: a decomposition model breaks user queries into atomic components, each embedded and matched against the library via squared L2 distance. The top-5 nearest skills populate the prompt, and a selection model chooses one for immediate invocation. This recursive decomposition mechanism handles granularity mismatches by refining abstract subtasks until a similarity threshold is exceeded, enabling reuse even in libraries exceeding 100 tools while reducing inference costs by 2–3× compared to context stuffing. Auto-creation via CRUD operations extends the library when retrieval fails, immediately making new skills available for subsequent reuse [23].

Voyager operates similarly but optimizes for embodied domains. Its skill library stores JavaScript functions with descriptions generated by GPT-3.5, indexed by `text-embedding-ada-002` embeddings. Finding augments queries with environment feedback: the agent generates a plan using GPT-3.5, combines it with game state observations, then retrieves the top-5 most relevant skills (96.5% top-5 accuracy). Application follows a code-as-policies paradigm, where GPT-4 composes a *new* function that *calls* retrieved skills, preventing catastrophic forgetting through externalized memory. This zero-shot generalization allows immediate reuse in novel environments, with skills invoked sequentially without parameter adaptation[14].

Table 2 compares these architectures. Both employ semantic retrieval, but Tulip's decomposition enables handling larger skill libraries, while Voyager's environmental grounding excels in open-ended tasks. Neither mechanism adapts skill parameters, classifying them as purely atomic.

Table 4: Atomic Reuse: Tulip Agent vs. Voyager

Feature	Tulip Agent	Voyager
Finding Mechanism	Subtask decomposition + k-NN search on embeddings	Plan augmentation + semantic matching
Application	Direct invocation via selection model	Code-as-policies composition (calls retrieved skills)
Key Detail	Recursive refinement for granularity mismatch	Environment feedback in query construction
Performance	2–3× cost reduction; auto-creation for gaps	96.5% retrieval accuracy; zero-shot generalization
Domain	General programmatic tasks	Embodied agents (Minecraft)

4.3.2 Selection-Driven Atomic Reuse: Routing and Cost Optimization

This class introduces explicit selection as a finding axis, where a router or dispatcher determines *whether* to reuse existing skills or create new ones. Application remains atomic but is optimized for computational economy, amortizing creation costs across multiple invocations.

LATM employs a multi-tier architecture comprising a Tool Maker (GPT-4) and a Tool User (GPT-3.5). The skill library functions as a functional cache storing wrapped tools Python functions bundled with usage demonstrations. Finding is orchestrated by a Dispatcher, a lightweight LLM that analyzes incoming queries to produce cache hits or misses, often utilizing internal reasoning traces [18] to determine solvability. On a hit, the Tool User reuses the function by parsing natural language into arguments through in-context learning; on a miss, the Tool Maker generates a new skill. This routing reduces asymptotic cost from $O(nC)$ to $O(nc+C)$, where C is the expensive model's cost and c the cheap model's cost. The Tool User achieves GPT-4-level accuracy at a fraction of the inference cost, with application limited to argument instantiation rather than skill modification[22].

TROVE similarly prioritizes selection but integrates it into a streaming context. Its skill library persists across examples as a shared function library F, initially empty. For each query, the system samples three generation modes: `IMPORT` (reuse existing skills), `CREATE` (define new functions), and `SKIP` (use primitives). Finding is implicit in mode selection; application follows a complexity-reduction tiebreaker: if `IMPORT` and `CREATE` yield correct answers, the solution with fewer operations is preferred, inherently favoring reuse. Skills are ranked by usage frequency in prompts, creating a positive feedback loop where reusable functions gain visibility. A trimming mechanism removes functions used fewer than $\lambda = \frac{1}{2} \times \log_{10}(n)$ times every 200 steps, ensuring only generic skills survive (79–98% library size reduction)[9].

Table 5: Selection-Driven Atomic Reuse: LATM vs. TROVE

Feature	LATM	TROVE
Finding	Dispatcher-mediated cache hit/miss routing	Mode-based sampling (IMPORT / CREATE / SKIP)
Application	Argument instantiation by Tool User	Direct invocation; complexity-based selection
Key Detail	Multi-tier cost amortization; demonstrations in wrapped tools	Frequency-based prompt ranking; $\lambda = \frac{1}{2}\log_{10}(n)$ trimming
Performance	GPT-3.5 Tool User matches GPT-4 accuracy	79–98% library reduction; favors simpler reused code
Domain	General natural language tasks	Streaming programmatic tasks

4.3.3 Compositional Reuse via Programmatic Chaining and Dependency Resolution

Compositional reuse elevates complexity by chaining multiple skills into executable programs, resolving input-output dependencies across execution steps. Finding leverages retrieval or relevance learning, while application integrates skills with control flow.

ReGAL constructs solutions by combining retrieved abstractions with primitive operations. Its skill library contains verified helper functions (the Code Bank C) and demonstration programs (the Demo Bank D). Finding retrieves up to 20 relevant functions from C via vector similarity between queries and function descriptions, supplemented by 10 in-context learning examples split between primitive and refactored programs. Application employs a ReAct-style prompt[17], explicitly asking the model to reason about which functions are relevant before generating code that calls them. This compositionality allows frequent functions to be reused more than 10 to 20 times across test examples, particularly in domains like TextCraft where helpers encapsulate game dynamics[4].

AutoTools formalizes chaining through dependency resolution. Its skill library $\mathcal{F} = \{(f_i, c_i, r_i)\}$ stores functions with usage examples c_i and return types r_i. Finding is trained via **Relevance Learning**, where the LLM selects relevant function identifiers from candidates. Application generates programs with standard control flow (`for` loops, `if` statements) that chain skills by parsing intermediate results to resolve dependencies. A **Function Learning** task optimizes step-level code generation, while variable caching persists results across multi-turn interactions. Integration verification during creation ensures new skills are tested against existing ones, guaranteeing interoperability. This mechanism consumes fewer tokens than documentation-heavy baselines, amortizing an average 2,703-token encapsulation cost across future tasks[16].

Table 6: Compositional Reuse: ReGAL vs. AutoTools

Feature	**ReGAL**	**AutoTools**
Finding	Vector similarity + split ICL budget	Relevance Learning to select functions
Application	ReAct-style gluing of abstractions + primitives	Programmatic chaining with dependency resolution
Key Detail	Demo Bank provides usage templates; 10-20+ reuse frequency	Variable caching; Integration verification for interoperability
Performance	High reuse in synthesis (LOGO shapes, TextCraft)	2,703-token amortized cost; multi-turn execution
Domain	Program synthesis, game environments	General tool use APIs

4.3.4 Adaptive Reuse via Parameterization and Contextual Instantiation

Adaptive reuse extends composition by dynamically instantiating skill parameters based on contextual observations, bridging high-level semantics with low-level execution. Finding remains retrieval-based, but application involves runtime parameter calculation.

CRADLE embodies this architecture through its Skill Curation and Action Planning modules. The skill library (Procedural Memory) stores Python functions encapsulating keyboard/mouse operations, indexed by `text-embedding-ada-002` embeddings of code and documentation. Finding computes cosine similarity between task descriptions and stored skill embeddings, retrieving the top K candidates to avoid overwhelming the Large Multimodal Model (LMM). Application is distinctly adaptive: the LMM (GPT-4o) instantiates retrieved skills by calculating parameters from current screenshots. For example, `move_forward(duration)` receives `duration=2` derived from target distance; `click_label(label_id)` receives a bounding box ID extracted via visual perception. This parameterization enables **skill composition**, where atomic skills are hierarchically nested (e.g., `turn_and_move_forward` bundles common sequences). Composite skills reduce backbone model interactions, making reuse more efficient than primitive-only generation[24].

4.3.5 Complex Hybrid Reuse: Imitation, Templating, and Bootstrapping

The most sophisticated reuse mechanisms hybridize direct retrieval with adaptive synthesis, where retrieved skills serve as templates for generating new, domain-specific variants. This creates a feedback loop that expands the skill library itself.

LEGO-Prover operates in theorem proving, where skills are verified lemmas in Isabelle. Its skill library maintains a lemma vector store indexed by embeddings. Finding constructs queries from formal statements and decomposer-generated subgoals, retrieving the top $n_f = 4$ or 6 lemmas via k-NN search. Application splits into two modes: **Direct Reuse** (51% of cases) copies retrieved lemmas verbatim into proofs via the `using` keyword; **Imitation** (49% of cases) treats lemmas as templates, synthesizing new lemmas that adapt the retrieved proof structure to novel constraints (e.g., generalizing `prod_1n_4n` to `prod_frac_common_factor`). This hybridity is amplified by the **Evolver**, which bootstraps the library: the Directional Transformer reuses existing skills to generate variants, while the Request Solver retrieves skills to close unsolved subgoals, growing the library from seeds to 22,532 skills. Ablation shows removing the library drops success rates from 50.4% to 47.1% on miniF2F-valid [10].

Table 7: Adaptive and Hybrid Reuse: CRADLE vs. LEGO-Prover

Feature	CRADLE (Adaptive)	LEGO-Prover (Hybrid)
Finding	Top-K cosine similarity on embeddings	k-NN on formal statements + subgoal queries

Table 7: Adaptive and Hybrid Reuse: CRADLE vs. LEGO-Prover

Feature	CRADLE (Adaptive)	LEGO-Prover (Hybrid)
Application	Contextual parameterization from visual input	51% direct copy; 49% imitation/templating
Key Detail	Duration/ID calculation from screenshots; hierarchical composition	Evolver bootstraps library to 22,532 lemmas; dual reuse modes
Performance	Reduced LMM interactions via composite skills	50.4% $\rightarrow$ 47.1% ablation drop; bridges human-formal gap
Domain	General computer control (games, software)	Neural theorem proving (Isabelle)

4.4 Library Management

Beyond the initial discovery and storage of skills, the long-term evolution of the skill library presents a distinct architectural challenge. Once skills are persisted, systems must decide how to maintain, prune, and enhance them over time. The literature reveals three philosophical paradigms that govern this lifecycle: archival preservation, empirical engineering, and interactive curation. These approaches diverge fundamentally in their treatment of library size, quality metrics, and the mutability of stored artifacts.

4.4.1 The Archival Paradigm: Preservation Through Formal Verification

In domains where correctness is non-negotiable, library management can adopt a monotonic growth strategy that prioritizes formal guarantees over efficiency. The LEGO-Prover system exemplifies this approach, treating its skill library as an immutable archive of mathematically verified lemmas[10]. Quality is defined primarily by binary verification through the Isabelle theorem prover. Each candidate skill, whether generated by the solving agent or the Evolver module, must pass compilation checks; proofs containing placeholders such as `sorry` or `oops` are automatically rejected. This verification gate provides strong correctness guarantees for accepted skills within the formal system.

Critically, this paradigm rejects pruning in its reported setup. The skill library is described as a "massive forest" that expands continuously from zero to over twenty thousand skills during training[10]. Skills are not deleted during this process, reflecting the assumption that formally verified knowledge however specialized may prove valuable for future proofs. Enhancement occurs through additive evolution: the Evolver module transforms existing skills along four directional axes (concept identification, parameterization, complexity scaling, and dimensional extension), while generalized variants are stored as *new* child nodes rather than overwriting their progenitors. This creates a skill-evolving tree where ancestral skills remain accessible.

Deduplication is addressed only at the point of entry, where a `SequenceMatcher` algorithm filters candidates exceeding 0.85 textual similarity to existing skills. However, semantic redundancy is tolerated, as the system lacks mechanisms to detect or merge functionally equivalent lemmas. The primary limitation of this approach lies in scalability: the absence of pruning leads to unbounded growth, potentially overwhelming retrieval mechanisms and computational resources. Furthermore, the reliance on formal verification restricts applicability to domains where such verification is feasible, rendering the paradigm unsuitable for open-ended programming or natural language tasks.

4.4.2 The Engineering Paradigm: Cyclical Maintenance and Empirical Quality

When libraries serve as functional toolboxes rather than formal archives, management adopts a software engineering mindset characterized by periodic refactoring and garbage collection. Systems in this cluster treat skills as mutable artifacts subject to continuous evaluation against empirical performance metrics. ReGAL and TROVE represent two instantiations of this philosophy, differing primarily in their pruning criteria and refinement policies [4, 9].

Both systems implement explicit periodic pruning cycles. In ReGAL, maintenance occurs every five training batches through two complementary processes: `editCodeBank` refines overfitted skills, while `pruneCodeBank` removes persistently failing functions[4]. The pruning decision relies on a quality score $s = |P| - \sum_{p \in F} 1/n_p$, where $|P|$ counts successful uses and F penalizes failures weighted by the number of functions invoked in each failing program. Skills scoring below threshold θ are deleted. This metric reflects a fault-tolerant view: a function's value is assessed by its net contribution to system success, accounting for contextual complexity.

TROVE, conversely, employs a usage-frequency metric $\lambda = C \times \log_{10}(n)$, where n is the number of processed examples and $C = 0.5$. Tools used fewer than λ times every 200 steps are pruned, operating under the heuristic that truly reusable skills will naturally be selected by the agent's retrieval mechanism. This approach achieves library size reductions of 74–90% without accuracy loss, demonstrating that many generated skills are task-specific cruft.

The two systems diverge critically on refinement. ReGAL's `editCodeBank` process actively repairs skills by prompting the LLM with passing and failing examples, generating more general implementations when the edited version passes more unit tests[4]. TROVE explicitly rejects such refinement, finding that it encourages overoptimization to the current task, reducing future reusability[9]. This reveals a tension within the engineering paradigm: refinement can enhance generality but risks catastrophic forgetting of cross-task applicability.

Table 8: Engineering Paradigm Variants in Library Management

Mechanism	ReGAL (Quality-Driven)	TROVE (Usage-Driven)
Pruning Trigger	Every 5 batches	Every 200 steps
Pruning Metric	Quality score $s = \|P\| - \sum_{p \in F} 1/n_p$	Usage threshold $\lambda = C \log_{10}(n)$
Refinement Policy	In-place editing via LLM feedback	Explicitly prohibited
Quality Gate	Execution verification with retry	Executability + self-consistency voting
Deduplication	Implicit via clustering pre-processing	Implicit via pruning

4.4.3 The Interactive Paradigm: Agent-Driven Explicit Curation

The Tulip Agent architecture represents a departure from automated management, positioning the LLM as an interactive librarian with direct CRUD privileges over its skill library. Rather than relying on background processes or statistical thresholds, all management operations are intentional actions executed by the agent in response to task demands or explicit instruction.

The AutoTulipAgent variant is equipped with meta-tools for Create, Read, Update, and Delete operations. When existing skills prove inadequate, the agent generates new Python functions with Sphinx-style docstrings and type hints, validates them via `ast` parsing, and dynamically loads them using `importlib` making them immediately available without system restart. Enhancement occurs through the Update tool, which modifies failing functions by feeding the existing code and error context back to the LLM for correction. Notably, the system tracks no usage statistics or success rates; quality assurance is purely syntactic and structural.

Pruning is similarly agent-initiated. The Delete tool allows the agent to remove skills explicitly, but no automated pruning mechanism exists[23]. This design sacrifices efficiency for interpretability and human oversight, ensuring that removal decisions are observable and explicable rather than emergent from opaque metrics. Deduplication is limited to prevention: the agent searches the library before creating new skills, but no post-hoc merging of semantically similar functions occurs.

The primary limitation is scalability in open-ended scenarios. Without automated pruning, the library may accumulate rarely-used skills, and without usage tracking, the agent lacks data-driven signals to guide its curation decisions. The paradigm is best suited for domains requiring human-in-the-loop validation or where metric design is unreliable.

5 Critical Assessment

Having examined the architectural mechanisms underlying library-learning agents in the preceding chapters, this section provides a critical comparative assessment

of the surveyed frameworks. We evaluate these systems across four key dimensions: the quality of toolbox construction in terms of generality and correctness, the rigor and standardization of evaluation protocols employed across domains, scalability considerations encompassing both inference cost and performance scaling, and the often-overlooked aspects of safety and verification.

5.1 Toolbox Construction Quality

The quality of a generated toolbox is determined by the *generality* and *correctness* of its tools.

Generality: ReGAL and LEGO-Prover excel here. LEGO-Prover's "Evolver" explicitly transforms skills to be more general (e.g., parameterizing constants), creating a robust library that covers a wide problem space. ReGAL's editing process actively refactors specific code into generic helpers. In contrast, Voyager's skills are often over-fitted to specific game contexts; a skill to "fight a zombie" might fail if the zombie is in water, unless the agent explicitly generalized the targeting logic. TROVE's rejection of refinement (rectification) suggests a philosophical trade-off: it prefers simple, atomic tools over complex, generalized ones that might be prone to overfitting.

Correctness: LEGO-Prover provides the strongest correctness guarantees among surveyed systems by requiring formal verification (Isabelle) for accepted lemmas. CRADLE and Voyager rely on execution feedback, which is a weaker proxy for correctness (a tool may work in observed runs but still fail in edge cases). AutoTools sits between these extremes, using static analysis and integration testing to improve documentation-function alignment.

5.2 Evaluation Protocol

The field suffers from a lack of standardized benchmarks[30], though convergence is appearing.

Embodied Benchmarks: Voyager and MINDSTORES use MineDojo[26], allowing for direct comparison. MINDSTORES shows a 9.4% mean improvement over DEPS, leveraging its experience database to solve tasks with fewer iterations [13]. CRADLE expands this to commercial games (RDR2, Cities: Skylines), but relies on "human evaluation" for complex tasks where programmatic success detection is hard, limiting reproducibility [24].

Reasoning Benchmarks: LATM, TROVE, and ReGAL largely use BigBench, MATH, and TabMWP. These are robust, quantitative benchmarks. TROVE's finding that it enables "31% faster human verification" is a novel metric that should be adopted more widely, measuring not just agent success but the *interpretability* and *human utility* of the solution [9].

For quick cross-system comparison, Table 9 centralizes evaluation settings and representative outcomes for all 11 surveyed frameworks.

Table 9: Central Evaluation Matrix for 11 Library-Learning Frameworks

Framework	Benchmark Family	Datasets / Tasks	Key Reported Metric
Voyager	Embodied benchmark	MineDojo tasks (open-ended Minecraft progression)	96.5% top-5 retrieval accuracy
CRADLE	Embodied + real-software environments	RDR2, Cities: Skylines, OSWorld tasks	Human-evaluated task completion
MindForge	Embodied multi-agent benchmark	MineDojo collaborative trajectories	3× tech-tree milestones; 2.3× unique items
MINDSTORES	Embodied planning benchmark	MineDojo task suite (compared to DEPS baseline)	+9.4% mean improvement over DEPS
LEGO-Prover	Formal theorem-proving benchmark	miniF2F (valid/test split)	50.4% with library vs 47.1% without
ReGAL	Program-synthesis/reasoning benchmarks	LOGO, Date Understanding, TextCraft, MATH, TabMWP	+11.5% (LOGO), +26.1% (Date), +8.1% (TextCraft)
TROVE	Mathematical reasoning + table/visual QA benchmarks	MATH; TabMWP/WTQ/HiTab; GQA	79–98% library reduction; 31% faster human verification
LATM	Language reasoning/tool-use benchmark suite	BIG-Bench tasks (Logical Deduction, Tracking Shuffled Objects, Dyck Language, Word Sorting, CRT) + Scheduling Meeting	GPT-4 maker + GPT-3.5 user reaches GPT-4-level performance with lower inference cost
Tulip Agent	Internal tool-use benchmark	3 task categories, 20 tasks each	2–3× cost reduction
AutoTools	API tool-use benchmarks	RestBench, ToolBench, AutoTools-Eval	2,703-token encapsulation cost (amortized)
ToolMaker	Scientific-repository benchmark	TM-Bench (15 repository-to-tool tasks)	80% correctness (12/15)

Note: Entries use paper-reported evaluation names; several systems rely on internal or mixed suites rather than a single standardized public benchmark.

5.3 Scalability

Scalability is viewed through the lens of **inference cost** and **performance scaling**.

Cost Efficiency: LATM is explicitly designed for this. By shifting most calls to a cheaper "Tool User," it reports substantial cost reduction while preserving strong task performance relative to stronger single-model baselines. Compared with always-on

high-capacity inference (as in GPT-4-centric loops such as Voyager), the Maker/User split offers a pragmatic efficiency pattern, though broader production-scale comparisons remain limited [14, 22].

Performance Scaling: LEGO-Prover demonstrates positive scaling: removing the learned library drops success from 50.4% to 47.1% on miniF2F-test, confirming the library's contribution. However, TROVE shows that for general reasoning, *smaller is better*; aggressive pruning (keeping only top 10% of tools) maintained or improved accuracy, suggesting that "library bloat" is a major noise factor for LLM planners that lack robust retrieval [9]. MINDSTORES faces a scalability challenge: retrieving raw text experiences incurs a high context-window cost[2], as it must re-read long narratives for every plan, making it expensive for very long histories [13].

5.4 Safety and Verification

Across the surveyed papers, safety evaluation remains less mature than task-performance evaluation. Broader benchmark efforts (AgentSafetyBench [31], R-Judge[32]) and recent surveys [12] indicate that robust risk-aware agent behavior is still an open challenge.

Sandboxing: ToolMaker uses Docker containers and a closed-loop debugging pipeline to isolate and validate generated tools during repository-to-tool conversion [5]. Tulip Agent performs structural safeguards such as AST parsing and typed function interfaces, but does not provide formal semantic verification of tool behavior [23].

Provenance and Data Quality Risks: Socially learned systems such as MindForge demonstrate strong collaborative gains, but also increase dependence on the quality of shared trajectories and datasets. The MindForge discussion explicitly notes potential risks in data sources and the use of safety filters, suggesting that provenance tracking and source auditing are important future requirements for socially learned libraries [15].

Evaluation Gap: Most systems rely on execution success, unit tests, or integration checks as practical correctness proxies. These methods improve reliability, but they do not fully capture semantic safety properties or downstream harm. AutoTools strengthens verification with documentation-function consistency checks and integration testing, yet its guarantees remain bounded by API-level constraints [16].

6 Research Gaps and Opportunities

Based on the analysis, several critical gaps emerge that represent fertile ground for future research.

6.1 The "Negative Knowledge" Gap

Among the surveyed systems, MINDSTORES most explicitly stores failure trajectories as useful data, using them to predict and avoid future failures [13]. Other systems (such as Voyager and CRADLE) focus primarily on successful code retention after reflection loops[20]. This may leave potentially useful negative signal underutilized, since failure traces can reveal boundary conditions of current policies. A promising

direction is to maintain a "Graveyard of Failures": a database of anti-patterns (e.g., "Do not use wood pickaxe on iron ore") that can be queried to prune search before planning.

6.2 The Forgetting Mechanism

TROVE introduces algorithmic pruning, but it is rudimentary (frequency-based). There is a need for *Semantically-Aware Forgetting*. Just as the human brain consolidates memories during sleep, agents need offline phases where they can reorganize their library merging similar functions (e.g., fight_zombie and fight_skeleton into fight_mob), refactoring code for efficiency, and deleting obsolete skills without explicit user prompts. Such a consolidation phase could substantially improve retrieval precision by reducing redundancy and semantic overlap in the skill library.

6.3 Social Immunity and Trust

MindForge demonstrates the upside of social learning, but systematic trust and provenance controls are still under-specified in current library-learning pipelines. As multi-agent exchange expands, future systems should incorporate **Trust Modeling**: recording who contributed a skill, under which environment and validation conditions, and how often that skill succeeds after reuse. Safety benchmark literature [31, 32] reinforces the need for explicit auditing and policy checks before high-impact reuse, especially for socially transferred artifacts.

6.4 Cross-Domain Transfer

Current libraries are domain-locked. A skill learned in Minecraft (e.g., "explore until you find resource X") is conceptually applicable to web navigation (e.g., "browse until you find product X"), but the code is incompatible. Research into *Abstract Skill Representations* perhaps using pseudo-code or formal logic intermediate layers could allow systems like CRADLE to transfer high-level strategies from *Stardew Valley* to *Excel* without starting from zero.

7 Conclusion

This survey reviewed 11 library-learning frameworks published between 2023 and 2025, revealing a converging architecture in which LLM agents couple a planner with persistent memory to accumulate executable abstractions. Organizing these systems around four pillars discovery, retention, reuse, and management yields the following answers to our research questions.

RQ1 (Mechanistic Taxonomy). Five discovery regimes emerge: environment-driven (Voyager, CRADLE), code-centric (ReGAL, TROVE), repository mining (AutoTools, ToolMaker), task-driven (LATM, Tulip), and formal verification-driven (LEGO-Prover). Code-centric and formalized approaches improve transfer through reusable abstractions, while the Maker/User separation in LATM reduces repeated inference cost. Experience-centric retention (MINDSTORES) offers richer contextual adaptation at the cost of higher retrieval overhead [4, 5, 9, 10, 13, 14, 16, 22–24].

RQ2 (Lifecycle Dynamics). Management strategies range from archival growth (LEGO-Prover), through engineering-style pruning (ReGAL, TROVE), to explicit agentic CRUD control (Tulip). Automated pruning can reduce library size by 79–98% (TROVE) without accuracy loss, but must balance compression against potential removal of low-frequency yet high-value skills [4, 9, 10, 23].

RQ3 (Generalization Limits). Self-generated tools generalize effectively when discovery is paired with abstraction and verification (ReGAL refactoring, LEGO-Prover evolution). Pure trial-and-error skills remain brittle outside their original context unless explicitly generalized; discovery mode therefore determines transfer robustness [4, 10, 14, 15].

RQ4 (Safety & Robustness). Current safeguards are uneven. Formal verification (LEGO-Prover) and sandboxed execution (ToolMaker) represent the strongest protections, while most systems still rely on execution success as a correctness proxy. Open challenges include extending verification beyond execution-level checks, auditing provenance in socially learned behaviors, and establishing dedicated safety benchmarks for library-learning systems [5, 10, 14–16, 22].

Looking ahead, tighter integration of the "mental models" of MINDSTORES, the cost efficiency of LATM, and the formal rigor of LEGO-Prover will be needed to move agents from static tool users toward adaptive, self-improving systems whose growing toolboxes embody an evolving understanding of their world.

References

[1] Schick, T., Dwivedi-Yu, J., Dessi, R., Raileanu, R., Lomeli, M., Hambro, E., Zettlemoyer, L., Cancedda, N., Scialom, T.: Toolformer: Language models can teach themselves to use tools. In: Oh, A., Naumann, T., Globerson, A., Saenko, K., Hardt, M., Levine, S. (eds.) Advances in Neural Information Processing Systems, vol. 36, pp. 68539–68551. Curran Associates, Inc., ??? (2023). https://proceedings.neurips.cc/paper_files/paper/2023/file/d842425e4bf79ba039352da0f658a906-Paper-Conference.pdf

[2] Packer, C., Wooders, S., Lin, K., Fang, V., Patil, S.G., Stoica, I., Gonzalez, J.E.: MemGPT: Towards LLMs as Operating Systems. arXiv. arXiv:2310.08560 [cs] (2024). https://doi.org/10.48550/arXiv.2310.08560 . http://arxiv.org/abs/2310.08560

[3] Qian, C., Liu, W., Liu, H., Chen, N., Dang, Y., Li, J., Yang, C., Chen, W., Su, Y., Cong, X., Xu, J., Li, D., Liu, Z., Sun, M.: ChatDev: Communicative agents for software development. In: Ku, L.-W., Martins, A., Srikumar, V. (eds.) Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pp. 15174–15186. Association for Computational Linguistics, Bangkok, Thailand (2024). https://doi.org/10.18653/v1/2024.acl-long.810 . https://aclanthology.org/2024.acl-long.810/

[4] Stengel-Eskin, E., Prasad, A., Bansal, M.: Regal: refactoring programs to discover generalizable abstractions. In: Proceedings of the 41st International Conference

on Machine Learning. ICML'24. JMLR.org, ??? (2024)

[5] Wölflein, G., Ferber, D., Truhn, D., Arandjelovic, O., Kather, J.N.: LLM agents making agent tools. In: Che, W., Nabende, J., Shutova, E., Pilehvar, M.T. (eds.) Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pp. 26092–26130. Association for Computational Linguistics, Vienna, Austria (2025). https://doi.org/10.18653/v1/2025.acl-long.1266 . https://aclanthology.org/2025.acl-long.1266/

[6] Gao, L., Madaan, A., Zhou, S., Alon, U., Liu, P., Yang, Y., Callan, J., Neubig, G.: Pal: program-aided language models. In: Proceedings of the 40th International Conference on Machine Learning. ICML'23. JMLR.org, ??? (2023)

[7] Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H.P.d.O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., Ryder, N., Pavlov, M., Power, A., Kaiser, L., Bavarian, M., Winter, C., Tillet, P., Such, F.P., Cummings, D., Plappert, M., Chantzis, F., Barnes, E., Herbert-Voss, A., Guss, W.H., Nichol, A., Paino, A., Tezak, N., Tang, J., Babuschkin, I., Balaji, S., Jain, S., Saunders, W., Hesse, C., Carr, A.N., Leike, J., Achiam, J., Misra, V., Morikawa, E., Radford, A., Knight, M., Brundage, M., Murati, M., Mayer, K., Welinder, P., McGrew, B., Amodei, D., McCandlish, S., Sutskever, I., Zaremba, W.: Evaluating Large Language Models Trained on Code. arXiv. arXiv:2107.03374 [cs] (2021). https://doi.org/10.48550/arXiv.2107.03374 . http://arxiv.org/abs/2107.03374

[8] Qin, Y., Liang, S., Ye, Y., Zhu, K., Yan, L., Lu, Y., Lin, Y., Cong, X., Tang, X., Qian, B., Zhao, S., Hong, L., Tian, R., Xie, R., Zhou, J., Gerstein, M., Li, D., Liu, Z., Sun, M.: ToolLLM: Facilitating Large Language Models to Master 16000+ Real-world APIs. arXiv. arXiv:2307.16789 [cs] (2023). https://doi.org/10.48550/arXiv.2307.16789 . http://arxiv.org/abs/2307.16789

[9] Wang, Z.Z., Neubig, G., Fried, D.: Trove: inducing verifiable and efficient toolboxes for solving programmatic tasks. In: Proceedings of the 41st International Conference on Machine Learning. ICML'24. JMLR.org, ??? (2024)

[10] Wang, H., Xin, H., Zheng, C., Li, L., Liu, Z., Cao, Q., Huang, Y., Xiong, J., Shi, H., Xie, E., Yin, J., Li, Z., Liao, H., Liang, X.: LEGO-Prover: Neural Theorem Proving with Growing Libraries. arXiv. https://doi.org/10.48550/arXiv.2310.00656 . http://arxiv.org/abs/2310.00656

[11] Yu, M., Meng, F., Zhou, X., Wang, S., Mao, J., Pan, L., Chen, T., Wang, K., Li, X., Zhang, Y., An, B., Wen, Q.: A survey on trustworthy llm agents: Threats and countermeasures. In: Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V.2. KDD '25, pp. 6216–6226. Association for Computing Machinery, New York, NY, USA (2025). https://doi.org/10.1145/3711896.3736561 . https://doi.org/10.1145/3711896.3736561

[12] Anonymous: Mind the agent: A comprehensive survey on large language model-based agent safety. In: Submitted to CS598 LLM Agent 2025 Workshop (2025). under review. https://openreview.net/forum?id=DHe0UXipKU

[13] Chari, A., Reddy, S., Tiwari, A., Lian, R., Zhou, B.: MINDSTORES: Memory-Informed Neural Decision Synthesis for Task-Oriented Reinforcement in Embodied Systems. arXiv. https://doi.org/10.48550/arXiv.2501.19318 . http://arxiv.org/abs/2501.19318

[14] Wang, G., Xie, Y., Jiang, Y., Mandlekar, A., Xiao, C., Zhu, Y., Fan, L., Anandkumar, A.: Voyager: An Open-Ended Embodied Agent with Large Language Models. arXiv. https://doi.org/10.48550/arXiv.2305.16291 . http://arxiv.org/abs/2305.16291

[15] Lică, M., Shirekar, O., Colle, B., Raman, C.: MindForge: Empowering Embodied Agents with Theory of Mind for Lifelong Cultural Learning. arXiv. version: 5. https://doi.org/10.48550/arXiv.2411.12977 . http://arxiv.org/abs/2411.12977

[16] Shi, Z., Gao, S., Yan, L., Feng, Y., Chen, X., Chen, Z., Yin, D., Verberne, S., Ren, Z.: Tool learning in the wild: Empowering language models as automatic tool agents. In: Proceedings of the ACM on Web Conference 2025. WWW '25, pp. 2222–2237. Association for Computing Machinery, New York, NY, USA (2025). https://doi.org/10.1145/3696410.3714825 . https://doi.org/10.1145/3696410.3714825

[17] Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., Cao, Y.: ReAct: Synergizing Reasoning and Acting in Language Models. arXiv. arXiv:2210.03629 [cs] (2023). https://doi.org/10.48550/arXiv.2210.03629 . http://arxiv.org/abs/2210.03629

[18] Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., Chi, E., Le, Q.V., Zhou, D.: Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. Advances in Neural Information Processing Systems **35**, 24824–24837 (2022)

[19] ichter, b., Brohan, A., Chebotar, Y., Finn, C., Hausman, K., Herzog, A., Ho, D., Ibarz, J., Irpan, A., Jang, E., Julian, R., Kalashnikov, D., Levine, S., Lu, Y., Parada, C., Rao, K., Sermanet, P., Toshev, A.T., Vanhoucke, V., Xia, F., Xiao, T., Xu, P., Yan, M., Brown, N., Ahn, M., Cortes, O., Sievers, N., Tan, C., Xu, S., Reyes, D., Rettinghouse, J., Quiambao, J., Pastor, P., Luu, L., Lee, K.-H., Kuang, Y., Jesmonth, S., Joshi, N.J., Jeffrey, K., Ruano, R.J., Hsu, J., Gopalakrishnan, K., David, B., Zeng, A., Fu, C.K.: Do as i can, not as i say: Grounding language in robotic affordances. In: Liu, K., Kulic, D., Ichnowski, J. (eds.) Proceedings of The 6th Conference on Robot Learning. Proceedings of Machine Learning Research, vol. 205, pp. 287–318. PMLR, ??? (2023). https://proceedings.mlr.press/v205/ichter23a.html

[20] Shinn, N., Cassano, F., Gopinath, A., Narasimhan, K., Yao, S.: Reflexion: language agents with verbal reinforcement learning. In: Proceedings of the 37th International Conference on Neural Information Processing Systems. NIPS '23. Curran Associates Inc., Red Hook, NY, USA (2023)

[21] Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W.-t., Rocktäschel, T., Riedel, S., Kiela, D.: Retrieval-augmented generation for knowledge-intensive nlp tasks. In: Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M.F., Lin, H. (eds.) Advances in Neural Information Processing Systems, vol. 33, pp. 9459–9474. Curran Associates, Inc., ??? (2020). https://proceedings.neurips.cc/paper_files/paper/2020/file/6b493230205f780e1bc26945df7481e5-Paper.pdf

[22] Cai, T., Wang, X., Ma, T., Chen, X., Zhou, D.: Large Language Models as Tool Makers. arXiv. version: 2. https://doi.org/10.48550/arXiv.2305.17126 . http://arxiv.org/abs/2305.17126

[23] Ocker, F., Tanneberg, D., Eggert, J., Gienger, M.: Tulip Agent – Enabling LLM-Based Agents to Solve Tasks Using Large Tool Libraries. arXiv. https://doi.org/10.48550/arXiv.2407.21778 . http://arxiv.org/abs/2407.21778

[24] Tan, W., Zhang, W., Xu, X., Xia, H., Ding, Z., Li, B., Zhou, B., Yue, J., Jiang, J., Li, Y., An, R., Qin, M., Zong, C., Zheng, L., Wu, Y., Chai, X., Bi, Y., Xie, T., Gu, P., Li, X., Zhang, C., Tian, L., Wang, C., Wang, X., Karlsson, B.F., An, B., Yan, S., Lu, Z.: Cradle: Empowering Foundation Agents Towards General Computer Control. arXiv. https://doi.org/10.48550/arXiv.2403.03186 . http://arxiv.org/abs/2403.03186

[25] Park, J.S., O'Brien, J., Cai, C.J., Morris, M.R., Liang, P., Bernstein, M.S.: Generative agents: Interactive simulacra of human behavior. In: Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology. UIST '23. Association for Computing Machinery, New York, NY, USA (2023). https://doi.org/10.1145/3586183.3606763 . https://doi.org/10.1145/3586183.3606763

[26] Fan, L., Wang, G., Jiang, Y., Mandlekar, A., Yang, Y., Zhu, H., Tang, A., Huang, D.-A., Zhu, Y., Anandkumar, A.: Minedojo: building open-ended embodied agents with internet-scale knowledge. In: Proceedings of the 36th International Conference on Neural Information Processing Systems. NIPS '22. Curran Associates Inc., Red Hook, NY, USA (2022)

[27] Ni, A., Iyer, S., Radev, D., Stoyanov, V., Yih, W.-t., Wang, S.I., Lin, X.V.: Lever: learning to verify language-to-code generation with execution. In: Proceedings of the 40th International Conference on Machine Learning. ICML'23. JMLR.org, ??? (2023)

[28] Ellis, K., Wong, L., Nye, M., Sablé-Meyer, M., Cary, L., Anaya Pozo,

L., Hewitt, L., Solar-Lezama, A., Tenenbaum, J.B.: Dreamcoder: growing generalizable, interpretable knowledge with wake–sleep bayesian program learning. Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences **381**(2251), 20220050 (2023) https://doi.org/10.1098/rsta.2022.0050 https://royalsocietypublishing.org/rsta/article-pdf/doi/10.1098/rsta.2022.0050/1327751/rsta.2022.0050.pdf

[29] Wu, Y., Jiang, A.Q., Li, W., Rabe, M., Staats, C., Jamnik, M., Szegedy, C.: Autoformalization with large language models. In: Koyejo, S., Mohamed, S., Agarwal, A., Belgrave, D., Cho, K., Oh, A. (eds.) Advances in Neural Information Processing Systems, vol. 35, pp. 32353–32368. Curran Associates, Inc., ??? (2022). https://proceedings.neurips.cc/paper_files/paper/2022/file/d0c6bc641a56bebee9d985b937307367-Paper-Conference.pdf

[30] Liu, X., Yu, H., Zhang, H., Zhan, Y., Wang, K., Zhang, S., Yuan, B., Wang, H., Wen, H., Zhao, Z., Zheng, W., Dong, Y., Tang, J.: Agentbench: Evaluating LLMs as agents. In: The Twelfth International Conference on Learning Representations (2024). https://openreview.net/forum?id=zAdUB0aCTQ

[31] Zhang, Z., Cui, S., Lu, Y., Zhou, J., Yang, J., Wang, H., Huang, M.: Agent-safetybench: Evaluating the safety of llm agents. arXiv preprint arXiv:2412.14470 (2024)

[32] Yuan, T., He, Z., Dong, L., Wang, Y., Zhao, R., Xia, T., Xu, L., Zhou, B., Li, F., Zhang, Z., Wang, R., Liu, G.: R-judge: Benchmarking safety risk awareness for LLM agents. In: Al-Onaizan, Y., Bansal, M., Chen, Y.-N. (eds.) Findings of the Association for Computational Linguistics: EMNLP 2024, pp. 1467–1490. Association for Computational Linguistics, Miami, Florida, USA (2024). https://doi.org/10.18653/v1/2024.findings-emnlp.79 . https://aclanthology.org/2024.findings-emnlp.79/