Are Agents Just Automata? On the Formal Equivalence Between Agentic AI and the Chomsky Hierarchy

Roham Koohestani JetBrains Research Amsterdam, The Netherlands roham.koohestani@jetbrains.com

Anton Podkopaev

Constructor University and JetBrains Research Bremen, Germany, and Amsterdam, The Netherlands apodkopaev@constructor.university

Abstract

This paper establishes a formal equivalence between the architectural classes of modern agentic AI systems and the abstract machines of the Chomsky hierarchy. We posit that the memory architecture of an AI agent is the definitive feature determining its computational power and that it directly maps it to a corresponding class of automaton. Specifically, we demonstrate that simple reflex agents are equivalent to Finite Automata, hierarchical task-decomposition agents are equivalent to Pushdown Automata, and agents employing readable/writable memory for reflection are equivalent to TMs. This Automata-Agent Framework provides a principled methodology for right-sizing agent architectures to optimize computational efficiency and cost. More critically, it creates a direct pathway to formal verification, enables the application of mature techniques from automata theory to guarantee agent safety and predictability. By classifying agents, we can formally delineate the boundary between verifiable systems and those whose behavior is fundamentally undecidable. We address the inherent probabilistic nature of LLM-based agents by extending the framework to probabilistic automata that allow quantitative risk analysis. The paper concludes by outlining an agenda for developing static analysis tools and grammars for agentic frameworks.

ACM Reference Format:

1 Introduction

The rapid evolution of artificial intelligence has transitioned from reactive single-turn systems to proactive autonomous entities known as agentic AI. These systems are designed to perceive their environment, formulate complex plans, and execute multi-step tasks to

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

Conference'17, Washington, DC, USA

© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM. ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM https://doi.org/10.1145/nnnnnnn.nnnnnn

Ziyou Li Delft University of Technology Delft, The Netherlands ziyou.li@tudelft.nl

Maliheh Izadi
Delft University of Technology
Delft, The Netherlands
m.izadi@tudelft.nl

achieve specified goals with minimal human intervention [5]. While often described in terms of cognitive capabilities like *reasoning* and *planning*, an analysis of their underlying architectures reveals a more fundamental structure. At their core, modern agentic AI systems are implementations of abstract state-transition machines [14].

This paper introduces a formal framework that establishes a direct correspondence between agentic AI architectures and the classical computational models defined by the Chomsky hierarchy. We argue that the expressive power, and consequently the decidability and verifiability, of an agent are fundamentally determined by its memory model. By classifying agents according to the sophistication of their memory: from no memory to a stack to unbounded memory, we can map them directly to finite automata (FA), pushdown automata (PDA) and Turing machines (TM), respectively.

Goal and Contribution: The primary motivation for this work is to bridge the gap between the empirical, fast-paced development of agentic systems and the rigorous, formal methods of theoretical computer science. The current discourse lacks a unifying theory for classifying agents on the basis of their computational power. This leads to monolithic designs where Turing-complete (TC) capabilities are the default, even for tasks requiring far less complexity. This one-size-fits-all approach results in systems that are often inefficient and difficult to formally verify safety and correctness.

The main contributions of this paper are: (1) A Formal Mapping: We establish a clear memory-based equivalence between classes of agentic AI and levels of the Chomsky hierarchy. (2) The Right-Sizing Principle: We introduce a decision framework to design agents with the minimal computational power necessary for a given task. (3) Research Roadmap: We outline a series of research paths to explore which focus on formal verification and testing, as well as practical engineering issues related to right-sizing.

In the sections to follow, we will initially provide the required background for our work (section 2) followed by our proposed framework in section 3. We conclude the paper with a discussion of presented work in section 4.

2 Background and Related Work

2.1 Agentic AI Architectures

Modern agentic AI systems are composed of several key components that work in a continuous operational loop. This loop, often

a variant of the Sense-Plan-Act cycle ¹, dictates how the agent interacts with its environment. **Perception:** The agent takes in information from its environment, which could be user input, sensor data, or the state of a software system. **Reasoning (LLM):** Large Language Models (LLM) are often used as the *brain* of the agent. These are tasked with processing the perceived information, maintaining a model of the world, and deciding on the next course of action. **Action Layer:** The agent executes actions through a set of available tools or APIs, which can modify the environment or its internal state. **Memory:** This is the crucial component that stores information about past states, actions, and observations. The structure and capacity of this memory are central to our thesis.

Operational loop variants like ReAct (Reason+Act) [17], State-Flow [14], and SMOT (Step-wise Memory-augmented Task-oriented) [8] all represent different strategies for managing this cycle, but fundamentally operate as state-transition systems.

2.2 Automata Theory Refresher

Automata theory is the study of abstract machines and computational problems that can be solved using them [1]. The Chomsky hierarchy classifies these machines based on their computational power, which is directly tied to the type of memory they possess.

The Chomsky hierarchy organizes computational models by their memory capacity and expressive power: finite automata, with no memory, recognize regular languages; pushdown automata, equipped with a stack, recognize context-free languages; linear-bounded automata, with tape space proportional to input length, recognize context-sensitive languages; and TMs, with unbounded memory, recognize recursively enumerable languages.

A key trade-off exists in this hierarchy: as expressive power increases (from FA to TM), the decidability of fundamental properties (like halting or acceptance) decreases [12]. For FA, most interesting properties are decidable, differing from TMs [9].

2.3 Prior Uses of Formal Methods in AI Systems

The application of formal methods to AI is not new. Classical multiagent systems have been analyzed using model-checking tools like AJPF [3] and MCMAS [11]. In the context of LLMs, techniques such as grammar-constrained decoding [10], Linear Temporal Logic (LTL) monitors [2], and frameworks like Formal-LLM [7] have been used to enforce certain properties on the output.

However, a significant gap remains: there is no unifying, memory-based classification that links the architectural patterns of modern agentic AI to the fundamental decidability and complexity results from automata theory. This paper aims to fill that gap.

3 A Formal Theory for Agentic AI

To develop a formal theory for agents, we must first define the standard automata for establishing equivalence. We subsequently present our claims for equivalence for which we have provided proofs in Appendix.

3.1 Standard Automata Definitions

The definitions that follow are standard in the theory of computation and are consistent with the established literature. Additionally, Table 1 provides an overview of the relevant definitions.

Finite Automaton (FA). A DFA is a 5-tuple $M=(Q,\Sigma,\delta,q_0,F)$ where:

- *Q* is a finite set of states.
- Σ is a finite set of input symbols, called the alphabet.
- $\delta: Q \times \Sigma \to Q$ is the transition function.
- $q_0 \in Q$ is the start state.
- $F \subseteq Q$ is the set of accept states.

A Note on Σ being finite. For an agent, Σ represents a *finite* vocabulary of discrete perceptions (after tokenization). This differs from the potentially unbounded raw event space. We model this gap via a tokenizer $\tau: \mathcal{E} \to \Sigma$ and a bounded context window of size κ . The bounded buffer can be encoded in the finite control, and can therefore preserve regularity for Regular agents. When κ is not globally bounded relative to the input, classifications rise.

It is well-established that for every NFA, an equivalent DFA can be constructed using the subset construction method; thus, DFAs and NFAs recognize the same class of languages, being the regular languages. We define the extended transition where for $M = (Q, \Sigma, \delta, q_0, F)$, define $\hat{\delta}: Q \times \Sigma^* \to Q$ is given by $\hat{\delta}(q, \epsilon) = q$ and $\hat{\delta}(q, xa) = \delta(\hat{\delta}(q, x), a)$ for $x \in \Sigma^*$, $a \in \Sigma$.

Pushdown Automaton (PDA). A pushdown automaton is a 6-tuple $P = (Q, \Sigma, \Gamma, \delta, q_0, F)$ where:

- *Q* is a finite set of states.
- Σ is a finite input alphabet.
- Γ is a finite stack alphabet.
- $\delta: Q \times \Sigma_{\epsilon} \times \Gamma_{\epsilon} \to \mathcal{P}(Q \times \Gamma_{\epsilon}^{*})$ is the transition function, where $\Sigma_{\epsilon} = \Sigma \cup \{\epsilon\}$ and $\Gamma_{\epsilon} = \Gamma \cup \{\epsilon\}$.
- $q_0 \in Q$ is the start state.
- $F \subseteq Q$ is the set of accept states.

PDAs recognize the class of context-free languages [12]. In terms of acceptance and configurations, we use *acceptance by final state*. A configuration is $(q, w, \alpha) \in Q \times \Sigma^* \times \Gamma^*$. We write $(q, aw, \gamma\alpha) \vdash (q', w, \alpha')$ if $(q', \alpha') \in \delta(q, a, \gamma)$ (with ϵ -moves when $a = \epsilon$ or $\gamma = \epsilon$).

Linear Bounded Automaton (LBA). A linear bounded automaton is a non-deterministic TM with a tape that is linearly bounded by the length of the input. Formally, an LBA is a 7-tuple $M=(Q,\Sigma,\Gamma,\delta,q_0,q_{\rm accept},q_{\rm reject})$ where all components are defined as for a TM, with the following constraints [12]: (1) The input alphabet Σ contains two special endmarker symbols, a left endmarker < and a right endmarker >, which are not part of the initial input string w. (2) The initial tape configuration is < w >. (3) The machine's transitions cannot print over the endmarkers, nor can the tape head move to the left of < or to the right of >.

LBAs recognize the class of context-sensitive languages [12]. Note that standard multi-tape \rightarrow single-tape simulations preserve linear space bounds, hence the LBA class is encoding invariant.

Turing Machine. A standard, single-tape TM is a 7-tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$ where:

• *Q* is a finite set of states.

 $^{^{1}\}hbox{E.g., https://motion.cs.illinois.edu/RoboticSystems/AnatomyOfARobot.html}$

- ∑ is the input alphabet, which does not contain the blank symbol ⊔.
- Γ is the tape alphabet, where $\Sigma \subseteq \Gamma$ and $\sqcup \in \Gamma$.
- $\delta: Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is the transition function.
- $q_0 \in Q$ is the start state.
- $q_{\text{accept}} \in Q$ is the accept state.
- $q_{\text{reject}} \in Q$ is the reject state, where $q_{\text{accept}} \neq q_{\text{reject}}$.

TMs recognize the class of recursively enumerable languages.

3.2 Formalizing Agentic AI Classes

To prove equivalence, the agent architectures described in the paper must be abstracted into formal acceptor models. A core concept here is the *language of an agent*.

DEFINITION 3.1. Let A be an agentic AI system with a set of possible perceptions Σ . The language of agent A, denoted L(A), is the set of all finite sequences of perceptions $w \in \Sigma^*$ that cause the agent to transition from its defined initial configuration to any one of a set of designated final or accepting configurations (e.g., states corresponding to task completion, goal achievement, successful termination).

With the given definition in mind, it is important to notice that the success mentioned here does not correspond to the success of an agent in its task, but rather the end of an execution. With this definition, we can formalize the agent classes as language acceptors.

3.3 Regular Agents ~ Finite Automata

A **Regular Agent** is an agent with a finite, constant-bounded memory. Formally, it is an acceptor model whose computational power is equivalent to that of a Finite Automaton (FA), as defined and proven in Appendix B. Its state is determined solely by its position in a predefined, finite state graph. A Regular Agent can be formally defined as a Mealy Machine, a type of FA with output, $M = (S, s_0, \Sigma, \Gamma, \delta, \lambda)$, where:

- *S* is a finite set of states.
- $s_0 \in S$ is the initial state.
- Σ is a finite set of input symbols (perceptions).
- Γ is a finite set of output symbols (actions/tools).
- $\delta: S \times \Sigma \to S$ is the state transition function.
- $\lambda : S \times \Sigma \to \Gamma$ is the output function.

If the policy includes nondeterministic choices (e.g., sampling), the induced acceptor is an NFA; by the subset construction, it is equivalent to a DFA for language recognition. The LLM (or any policy) may be used as a *transition oracle*, but the architecture enforces that only the current state and the current (bounded-context) perception symbol influence the next state and output.

3.3.1 What Does and What Does Not Count As a Regular Agent? In order to best illustrate this we provide examples of what does and what does not constitute a regular agent. Examples that **do** fit: (1) **LLM-as-transition oracle.** The transition function $\delta(s, a)$ is computed by an LLM call, but the system architecture restricts control flow to a fixed graph (finite S). The oracle cannot introduce new states or persistent memory; it merely selects among predeclared edges. (2) **2018-style Siri/voice assistant.** Invocation \rightarrow intent detection \rightarrow slot disambiguation \rightarrow action \rightarrow termination. Each stage is a node; user replies label edges. No persistent plan stack

exists; the interaction is a short, acyclic dialogue with bounded prompts. (3) **Voice-commanded robot (no autonomy).** A dormant state waits for commands (e.g., $MOVE_TO_(x,y)$), executes a motion primitive, then returns to idle. The controller contains no subgoal stack and no writable memory beyond sensor inputs. Because behavior is bounded by M's finite graph, classic model checking decides reachability (safety) and inevitability (liveness).

Any form of unbounded *planning* or nested subtasking introduces a stack-like discipline (push subgoal, work, pop), which exceeds FA power (see §3.4). Likewise, giving the agent arbitrary read/write scratchpads moves it to TM power (§3.5), and therefore **does not fit** into this level.

3.4 Context-Free Agents ≃ Pushdown Automata

A **Context-Free Agent** is an agent that augments a finite state control with a stack (a Last-In, First-Out (LIFO) memory structure). This architecture corresponds directly to a Pushdown Automaton (PDA); see Appendix C for the formal equivalence. This allows the agent to handle nested subtasks and hierarchical plans. These agents are equivalent to Pushdown Automata (PDA). The transition function for a Context-Free Agent becomes:

$$\delta: S \times \Sigma \times Z \to S \times Z^*$$

where Z is the stack alphabet and Z^* is a string of stack symbols to be pushed. The agent's next state depends not only on the current state and input, but also on the symbol at the top of the stack.

To illustrate this, take the example of a project management agent that breaks down a high-level goal ("launch product") into a sequence of subtasks ("design," "develop," "test"). To complete "develop," it must first complete nested sub-subtasks ("write code," "run unit tests"). As it completes a subtask, it "pops" it from its plan stack and returns to the parent task. Although more powerful than Regular Agents, Context-Free Agents are still amenable to formal verification. Model-checking PDAs is decidable, though more complex than for FSMs. We can verify properties related to the proper execution of hierarchical plans, such as ensuring that all subroutines eventually return. Many classic properties become undecidable for general PDAs (e.g., language equivalence), while several remain decidable for deterministic PDAs. We therefore recommend designs that enforce strict LIFO subroutine discipline and determinism where possible.

3.5 TC Agents ≈ Universal Machines

A **TC Agent** is an agent with access to an unbounded, arbitrarily readable/writable memory, analogous to a TM's tape. As we formally demonstrate in Appendix D, this grants the agent universal computation capabilities. This memory can be a scratchpad, a vector database with unlimited storage, or any external memory that can be read from and written to arbitrarily.

The transition function for such an agent is:

$$\delta: S \times \Gamma \longrightarrow S \times \Gamma \times \{L,R\}$$

The agent can read the memory, write to it, and move its "head" left or right. This gives it universal computation capabilities. Note that here we encode the input on the tape (with $\Sigma \subseteq \Gamma$). When taking a concrete example of this type of agent, one can think of a research agent that can browse the web, save information to a file, read from

Computational Class	Automaton Model	Formal Definition	Agent Model	Formal Definition
Regular	Finite Automaton (FA)	$(Q, \Sigma, \delta, q_0, F)$	Regular Agent (A_R)	$(S, s_0, \Sigma, \delta, F)$
Context-Free	Pushdown Automaton (PDA)	$(Q, \Sigma, \Gamma, \delta, q_0, F)$	Context-Free Agent (A_{CF})	$(S, s_0, \Sigma, Z, \delta, F)$
Context-Sensitive	Linear Bounded	$(Q, \Sigma, \Gamma, \delta, q_0, q_a, q_r)$	Context-Sensitive	$(S, s_0, \Sigma, \Gamma, \delta, F)$
	Automaton (LBA)		Agent (A_{CS})	with bounded memory
Recursively	Turing Machine (TM)	(0,5,5,3,3,3)	TC	$(S, s_0, \Sigma, \Gamma, \delta, F)$
Enumerable	ruring wacinie (1wi)	$(Q, \Sigma, \Gamma, \delta, q_0, q_a, q_r)$	Agent (A_{TC})	with unbounded memory

Table 1: Consolidated view of formal Automata definitions.

that file, and iteratively refine a report based on the accumulated knowledge. The agent's memory (the file) is practically unbounded. The expressive power of TC Agents comes at the cost of decidability. As with any TM, fundamental questions like the Halting Problem are undecidable. We cannot, in the general case, prove that such an agent will terminate, or that it will not enter an unsafe state. Verification must rely on testing, runtime monitoring, and other incomplete methods.

3.6 Extension to Multi-Agent Systems and the Universal Machine

The framework can be extended to Multi-Agent Systems (MAS). Consider a MAS composed of n individual agents, $\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_n$, where each agent \mathcal{A}_i is a FA with its own set of states S_i . At any time, the global state of the system is a tuple representing the state of each agent: $s_{qlobal} \in S_1 \times S_2 \times \cdots \times S_n$.

This composition is analogous to the construction of a Deterministic Finite Automaton (DFA) from a Nondeterministic Finite Automaton (NFA). In an NFA, the machine can be in a set of states simultaneously. The equivalent DFA has a single state for every possible subset of the NFA's states. Similarly, our MAS can be modeled as a single, composite FSM where each state in the composite machine corresponds to one specific permutation tuple of the individual agents' states. Since it is a well-established theorem that NFAs and DFAs have equivalent expressive power, the entire multiagent state machine is computationally no more powerful than a single, albeit vastly larger, FSM.

The critical leap in capability occurs when these agents, regardless of their individual simplicity, are given access to a shared, unbounded, readable/writable memory, i.e., a tape. In this scenario, the collection of FSMs acts as a distributed *head* for a single TM. The composite FSM dictates the state transitions, while the shared memory provides the computation capability. Therefore, a multiagent system of simple Regular Agents combined with a shared unbounded memory is computationally equivalent to a Universal TM. Note that we assume atomic read—write (or a serialized scheduler) over the shared memory. Concurrency does not increase computational power beyond TM, but without atomicity the operational semantics become more intricate.

3.7 Right-Sizing Agents

The core engineering principle derivable from our proposed theory is "right-sizing". We define this as selecting the minimal computational class sufficient for the task. To assist with the process of

choosing and appropriate architecture, we provide a decision diagram in Figure 1 in addition to a mapping from popular existing frameworks to our proposed hierarchy in Table 2.

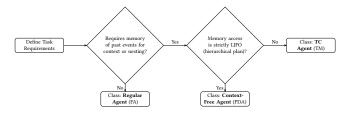


Figure 1: Decision Flowchart for Right-Sizing Agentic AI

Table 2: Mapping of Agentic Frameworks to Computational Classes

Framework/Pattern	Typical Architecture	Equivalent Class
Rule-Based Chatbots, IFTTT ^a , n8n ^b	Fixed decision trees, direct trigger–action logic.	Regular (FA)
CrewAI ^c , AutoGen (Hierarchical) [13]	Manager–worker patterns, task decomposition and delegation.	Context-Free (PDA)
LangGraph (Acyclic/Tree) ^d	Graph-based control flow for structured, nested tasks.	Context-Free (PDA)
ReAct [17], Auto-GPT [15]	Unconstrained reasoning loop with a readable/writable scratchpad.	TC (TM)
LangGraph (Cyclic) ^d	General graphs allowing arbitrary state transitions and memory updates.	TC (TM)
[a] https://ifttt.com/ [b] https://	//n8n.io/ [c] https://www.crev ai.com/	v [d] https://www.lang chain.com/langgraph

4 Discussion

4.1 Implications for Engineering Practice

This formal framework has significant practical implications for the design and deployment of agentic AI systems.

Cost and Latency Savings By constraining the agent's computational path to a set of possible paths (Regular) or a structured hierarchical flow (Context-Free), these simpler architectures inherently limit the number of execution steps. This prevents the costly iterative loops and potential non-termination of TC systems, leading to substantial savings in LLM inference costs and lower latency. More importantly, it reduces the analytical complexity, making the agent's behavior formally verifiable.

Regulatory Compliance In safety-critical domains like finance or healthcare, the ability to formally prove properties is a necessity. An agent specified as an FSM has a **transparent and fully auditable structure**. Every possible state and transition is explicitly defined, meaning auditors can guarantee the agent will never enter an undefined state or perform an unapproved action. While the LLM's decision to take a *specific* transition at runtime remains a *black box*, its choices are strictly confined to the pre-defined, verifiable pathways of the FSM.

Improved Reliability Finally, choosing less powerful computational classes leads to more predictable behavior. This results in more reliable and robust systems that are less prone to the emergent and sometimes undesirable actions of agents with unrestricted memory and reasoning capabilities.

4.2 General Implications of the Theory

4.2.1 Importing Computability Theory. The most significant consequence of establishing these equivalences is the ability to directly import results from computability theory into the domain of AI safety and verification. The boundary between decidability and undecidability is now sharply defined within the agent hierarchy.

Verifiable Agent Classes (Regular, Context-Free, Context-Sensitive). For agents proven equivalent to FAs, and PDAs, fundamental verification questions are algorithmically solvable, or decidable. For Regular Agents, standard properties (e.g., reachability/safety) are decidable. For Context-Free Agents, many properties remain decidable under determinism (DPDA), but language equivalence is undecidable in general for PDAs; it is decidable for deterministic pushdown automata, albeit with extremely high complexity. For Context-Sensitive Agents, whether a computation halts on a given input is decidable (via finiteness of configurations).

Unverifiable Agent Class (TC). For agents proven equivalent to TMs, Rice's Theorem applies. This theorem states that any nontrivial semantic property of the language recognized by a TM is undecidable. This has direct and severe consequences for verification. In term of **Undecidability of Safety:** It is impossible to create a general algorithm that can take any TC Agent and an unsafe property and determine whether the agent could ever exhibit that property. Not only that, but also **Undecidability of Halting:** It is impossible to create a general algorithm to determine whether an arbitrary TC Agent will terminate on a given sequence of perceptions.

4.2.2 A Foundation for Probabilistic Agent Models. The equivalences established here are a prerequisite for addressing the probabilistic nature of LLM-based agents. An LLM-implemented transition function δ is not deterministic. This can be modeled by extending our classes to their probabilistic counterparts (e.g., Probabilistic Finite Automata, PFA). With this foundation, verification transforms. Instead of asking "Can the agent reach an unsafe state?", we can now ask a quantitative one: "What is the probability that the agent will reach an unsafe state?". This shifts agent verification from absolute guarantees to quantitative risk analysis, a domain familiar in policy and management.

4.2.3 A Note on the Definition of Agency. Our working definition of agency requires the ability to formulate and execute plans to achieve goals. Plans inherently imply a dynamic set of pending subgoals and return points. When managed with a strict LIFO discipline, this is precisely the behavior of a stack. Therefore, we argue that any genuinely planning agent (whose plan depth is not artificially bounded) is at least PDA-equivalent. An FSM-governed system may be interactive and use an LLM, but it is not agentic in that it cannot form, store, and revise arbitrarily deep hierarchical plans.

4.3 Limitations and Threats to Validity

Our framework has several limitations: (1) The LLM as a Stateful Actor: Our model treats the LLM as a stateless "transition oracle." In reality, the LLM's decisions are based on its entire context window, which acts as a form of memory. Our FA and PDA models hold only if the system's architecture strictly enforces that the context window passed to the LLM is either constant-sized (for FA) or managed like a stack (for PDA), and that information outside this disciplined memory is not used for control flow decisions. (2) State-Space Explosion: While problems for FSMs and PDAs are decidable in theory, the number of states can be astronomically large in practice, making naive model-checking infeasible. Abstraction and symbolic model-checking techniques are necessary to apply these methods to real-world agents. (3) Abstraction Fidelity: Modeling a complex agentic system as a formal automaton requires simplification. If the model does not accurately capture the agent's control flow, any guarantees derived from it are meaningless.(4) Probabilistic Nature of LLMs: Our core model assumes a deterministic transition function. In practice, this means abstracting the LLM's output as the most likely action, which may not hold when using higher temperature sampling during inference.

5 Conclusion and Outlook

We introduced a memory-centric framework that connects agentic AI architectures to the Chomsky hierarchy. This correlation offers a practical *right-sizing* guideline which proposes opting for the weakest class that fulfills the task to minimize costs and latency, ensure clearer assurance cases, and enable decidable verification (where possible). The approach applies to multi-agent scenarios and effectively incorporates probabilistic behavior by shifting from querying absolute safety to assessing quantitative risk. Though our framework relies on strict memory control and accurate models, it clarifies actual engineering trade-offs and identifies where model checking, abstraction, and symbolic techniques are most beneficial.

Looking forward, we see two concrete trajectories to operationalize the theory. First, *minimal-class synthesis*: compile high-level task specifications into the weakest sufficient automaton and emit both an executable agent and proof artifacts (or conformance evidence) supporting the chosen class. Second, *hybrid architectures with runtime guards*: wrap TM-level components in verifiable FA/DPDA *cores* that mediate tools and enforce safety via monitors, combining partial static guarantees with runtime verification. Together with probabilistic verification for calibrated risk, these directions form a toolchain that turns language agents into a practical standard for building more efficient and predictable agentic AI.

We invite the community to contribute reference agents and case studies that probe class boundaries under realistic memory constraints and stochastic policies. A shared conformance and benchmarking suite would make right-sizing an auditable engineering decision rather than a design heuristic.

References

- [1] W. R. Ashby, J. T. Culbertson, M. D. Davis, S. C. Kleene, K. De Leeuw, D. M. Mac Kay, J. Mc Carthy, M. L. Minsky, E. F. Moore, C. E. Shannon, N. Shapiro, A. M. Uttley & J. Von Neumann (1956): Automata Studies. (AM-34). Princeton University Press. Available at https://www.jstor.org/stable/j.ctt1bgzb3s.
- [2] Itay Cohen & Doron Peled (2024): End-to-End AI Generated Runtime Verification from Natural Language Specification. doi:10.1007/978-3-031-73741-1_23. Available at https://link.springer.com/chapter/10.1007/978-3-031-73741-1_23.
- [3] Louise A. Dennis (2018): The MCAPL Framework including the Agent Infrastructure Layer an Agent Java Pathfinder. Available at https://joss.theoj.org/papers/10.211 05/joss.00617.
- [4] Louise A. Dennis, Michael Fisher, Matthew P. Webster & Rafael H. Bordini (2012): Model checking agent programming languages. Automated Software Engineering 19(1), pp. 5–63, doi:10.1007/s10515-011-0088-x. Available at https://link.sprin ger.com/article/10.1007/s10515-011-0088-x. Company: Springer Distributor: Springer Institution: Springer Label: Springer Publisher: Springer US.
- [5] Junda He, Christoph Treude & David Lo (2025): LLM-Based Multi-Agent Systems for Software Engineering: Literature Review, Vision, and the Road Ahead. ACM Trans. Softw. Eng. Methodol. 34(5), pp. 124:1–124:30, doi:10.1145/3712003. Available at https://dl.acm.org/doi/10.1145/3712003.
- [6] Junda He, Christoph Treude & David Lo (2025): LLM-Based Multi-Agent Systems for Software Engineering: Literature Review, Vision, and the Road Ahead. ACM Transactions on Software Engineering and Methodology 34(5), pp. 1–30, doi:10.1145/3712003. Available at https://dl.acm.org/doi/10.1145/3712003.
- [7] Zelong Li, Wenyue Hua, Hao Wang, He Zhu & Yongfeng Zhang (2024): Formal-LLM: Integrating Formal Language and Natural Language for Controllable LLMbased Agents, doi:10.48550/arXiv.2402.00798. Available at http://arxiv.org/abs/24 02.00798. ArXiv:2402.00798 [cs].
- [8] Jia Liu & Jie Shuai (2023): SMoT: Think in State Machine, doi:10.48550/arXiv.2312.17445. Available at http://arxiv.org/abs/2312.17445. ArXiv:2312.17445 [cs] version: 1.
- [9] Salvador Lucas (2021): The origins of the halting problem. Journal of Logical and Algebraic Methods in Programming 121, p. 100687, doi:10.1016/j.jlamp.2021.100687.
 Available at https://www.sciencedirect.com/science/article/pii/S2352220821000
 50X
- [10] Kanghee Park, Jiayu Wang, Taylor Berg-Kirkpatrick, Nadia Polikarpova & Loris D'Antoni (2024): Grammar-Aligned Decoding, doi:10.48550/arXiv.2405.21047. Available at http://arxiv.org/abs/2405.21047. ArXiv:2405.21047 [cs].
- [11] SAIL (2006): MCMAS. SAIL. Available at https://sail.doc.ic.ac.uk/software/mcmas/. See "https://sail.doc.ic.ac.uk/software/mcmas/".
- [12] Michael Sipser (2012): Introduction to the Theory of Computation. Cengage Learning. Google-Books-ID: 4J1ZMAEACAAJ.
- [13] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W. White, Doug Burger & Chi Wang (2023): AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation, doi:10.48550/arXiv.2308.08155. Available at http://arxiv.org/abs/2308.08155. ArXiv:2308.08155 [cs].
- [14] Yiran Wu, Tianwei Yue, Shaokun Zhang, Chi Wang & Qingyun Wu (2024): StateFlow: Enhancing LLM Task-Solving through State-Driven Workflows, doi:10.48550/arXiv.2403.11322. Available at http://arxiv.org/abs/2403.11322. ArXiv.2403.11322 [cs].
- [15] Hui Yang, Sifu Yue & Yunzhong He (2023): Auto-GPT for Online Decision Making: Benchmarks and Additional Opinions, doi:10.48550/arXiv.2306.02224. Available at http://arxiv.org/abs/2306.02224. ArXiv:2306.02224 [cs].
- [16] Hui Yang, Sifu Yue & Yunzhong He (2023): Auto-GPT for Online Decision Making: Benchmarks and Additional Opinions, doi:10.48550/arXiv.2306.02224. Available at http://arxiv.org/abs/2306.02224. ArXiv:2306.02224 [cs].
- [17] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan & Yuan Cao (2023): ReAct: Synergizing Reasoning and Acting in Language Models, doi:10.48550/arXiv.2210.03629. Available at http://arxiv.org/abs/2210.03629. ArXiv:2210.03629 [cs].

The appendix is structured as follows: Using the definitions presented in **subsection 3.1** we formalizes our proposed agent classes as language acceptors, and establishes a clear basis for comparison in **Appendix A**. **Appendix B through Appendix D** present the formal proofs of equivalence for each class of agent. Each section

demonstrates, by mutual simulation, that an agent class and its corresponding automaton recognize the exact same class of languages.

A Formalizing Agentic AI Classes

Regular Agent. In the paper, we define a Regular Agent as a Mealy machine, which includes an output function λ for generating actions . For the purpose of language recognition and classification within the Chomsky hierarchy, the output component is irrelevant. The computational power is determined by the state-transition dynamics. We therefore define a Regular Agent acceptor as a 5-tuple $A_R = (S, s_0, \Sigma, \delta, F)$ where:

- *S* is a finite set of internal agent states.
- $s_0 \in S$ is the initial state.
- Σ is a finite set of perceptions (the input alphabet).
- δ: S × Σ → S is the state transition function, determined by the agent's reasoning engine.
- $F \subseteq S$ is a set of accepting states.

Note that if the agent policy is nondeterministic, A_R corresponds to an NFA ($\delta: S \times \Sigma \to \mathcal{P}(S)$); by the subset construction this is equivalent to a DFA for recognition purposes.

Context-Free Agent. Augmenting the finite control with a stack memory, we define a Context-Free Agent as a 6-tuple $A_{CF}=(S,s_0,\Sigma,Z,\delta,F)$ where:

- S, s_0 , Σ , F are as defined for the Regular Agent.
- *Z* is a finite stack alphabet, for the agent's LIFO memory.
- $\delta: S \times \Sigma_{\epsilon} \times Z_{\epsilon} \to \mathcal{P}(S \times Z_{\epsilon}^*)$ is the transition function, where the agent's next state and stack operation depend on the current state, perception, and the symbol at the top of the stack.

Context-Sensitive Agent. Although not explicitly detailed in the text of , a Context-Sensitive Agent is implied by the Chomsky hierarchy table presented therein and is required to complete the hierarchy. This agent class is defined by a memory model analogous to an LBA's tape. A Context-Sensitive Agent is a tuple $A_{CS}=(S,s_0,\Sigma,\Gamma,\delta,F)$ whose computation on an input perception sequence $w\in\Sigma^*$ is constrained to a memory tape of length $k\cdot|w|$ for some constant k. Its transition function δ maps the current state, perception, and memory content to a new state and updated memory. For the sake of brevity in the paper we stick to the three-layered hierarchy presented.

TC Agent. An agent with access to an unbounded, arbitrarily readable/writable memory structure is defined as a TC Agent. Its formal definition mirrors that of a TM. It is a tuple $A_{TC} = (S, s_0, \Sigma, \Gamma, \delta, F)$ where its transition function $\delta: S \times \Gamma \to S \times \Gamma \times \{L, R\}$ operates on an unbounded memory tape, where $\Sigma \subseteq \Gamma$. The agent's internal state tracks the TM state, and its memory acts as the TM tape. Finally, for TC agents we encode the input on the memory tape and let $\delta: S \times \Gamma \to S \times \Gamma \times \{L, R\}$ operate on that tape (with $\Sigma \subseteq \Gamma$).

B The Equivalence of Regular Agents and Finite Automata

This section presents the first and most direct of the equivalence proofs. It formally establishes that the set of perception sequences recognized by Regular Agents constitutes precisely the class of regular languages.

THEOREM B.1. A language L is recognized by a Regular Agent A_R if and only if L is a regular language.

The proof proceeds by construction in two parts and demonstrates that for any Regular Agent, an equivalent FA can be constructed, and vice versa.

B.1 From Regular Agent to FA (\Rightarrow)

LEMMA B.2. For any Regular Agent $A_R = (S, s_0, \Sigma, \delta, F)$, there exists a DFA $M = (Q, \Sigma, \delta', q_0, F')$ such that $L(M) = L(A_R)$.

Construction B.1. We construct the DFA M by establishing a direct correspondence with the components of the agent A_R . (1) Let the set of states Q = S. (2) The input alphabet Σ is identical for both. (3) Let the start state $q_0 = s_0$. (4) Let the set of accept states F' = F. (5) Define the transition function δ' to be identical to the agent's transition function δ . That is, for every state $s \in S$ and perception $a \in \Sigma$, let $\delta'(s, a) = \delta(s, a)$.

PROOF. The construction establishes a one-to-one mapping between the components of A_R and M. We prove by induction on the length of an input string $w \in \Sigma^*$ that the computation of A_R on w mirrors that of M. Let $\hat{\delta}$ be the extended transition function for A_R and $\hat{\delta}'$ be the extended transition function for M.

Base Case. For |w| = 0, $w = \epsilon$. By definition, $\hat{\delta}(s_0, \epsilon) = s_0$ and $\hat{\delta}'(q_0, \epsilon) = q_0$. Since $q_0 = s_0$, the property holds.

Inductive Step. Assume that for any string w of length n, $\hat{\delta}(s_0, w) = \hat{\delta}'(q_0, w)$. Consider a string x = wa of length n + 1, where $a \in \Sigma$. The computation proceeds as follows:

$$\begin{split} \hat{\delta}'(q_0,wa) &= \delta'(\hat{\delta}'(q_0,w),a) & \text{ (by def. of ext. transition function)} \\ &= \delta'(\hat{\delta}(s_0,w),a) & \text{ (by the I.H.)} \\ &= \delta(\hat{\delta}(s_0,w),a) & \text{ (by construction, since } \delta' = \delta) \\ &= \hat{\delta}(s_0,wa) & \text{ (by def. of ext. transition function)} \end{split}$$

Thus, for any string w, A_R ends in state $s = \hat{\delta}(s_0, w)$ if and only if M ends in state $q = \hat{\delta}'(q_0, w)$, where s = q. An input string w is in $L(A_R)$ if and only if $\hat{\delta}(s_0, w) \in F$. By our proof, this is true if and only if $\hat{\delta}'(q_0, w) \in F'$. Since F = F', it follows that $w \in L(M)$. Therefore, $L(A_R) = L(M)$. As M is a DFA, $L(A_R)$ is a regular language. \square

B.2 From FA to Regular Agent (⇐)

LEMMA B.3. For any FA $M = (Q, \Sigma, \delta, q_0, F)$, there exists a Regular Agent $A_R = (S, s_0, \Sigma, \delta', F')$ such that $L(A_R) = L(M)$.

Construction B.2. This construction is the converse of the previous one. We construct the agent A_R to simulate the automaton M. (1) Let the set of states S = Q. (2) The perception alphabet Σ is identical. (3) Let the initial state $s_0 = q_0$. (4) Let the set of accepting states F' = F. (5) Define the agent's transition function δ' to be identical to δ .

PROOF. The argument is symmetric to that of the previous Lemma. The agent A_R is constructed to be a perfect simulator of the FA M.

Every state in M corresponds to a state in A_R , and the transitions are identical. Consequently, for any input string w, A_R will be in an accepting state if and only if M is in an accepting state. Therefore, $L(A_R) = L(M)$.

Note: in the paper we define the Regular Agent using a Mealy machine model, which includes an output function $\lambda:S\times\Sigma\to\Gamma$ to produce actions. The proof above deliberately omits this component. This is because the Chomsky hierarchy classifies computational models based on their *language recognition* capabilities, not their output-generating (transducer) capabilities. Also, we must note that we only consider the deterministic case. For nondeterministic policies, replace δ by $\delta:S\times\Sigma\to\mathcal{P}(S)$ and interpret $\hat{\delta}$ in the standard NFA sense.

This proof thus formalizes that the core capability of a Regular Agent, that is, what it can decide based on a sequence of perceptions, is determined entirely by its finite memory (its states), which is the defining characteristic of a FA.

C The Equivalence of Context-Free Agents and Pushdown Automata

This section extends the formal equivalence to the next level of the Chomsky hierarchy. It proves that agents augmented with a stack-based memory, possess the computational power equivalent to that of Pushdown Automata, thereby recognizing the class of context-free languages.

Theorem C.1. A language L is recognized by a Context-Free Agent A_{CF} if and only if L is recognized by a Pushdown Automaton P.

The proof strategy again relies on mutual simulation, demonstrating that the formal definitions of a Context-Free Agent and a Pushdown Automaton are structurally isomorphic.

C.1 From Context-Free Agent to Pushdown Automaton (⇒)

LEMMA C.2. For any Context-Free Agent $A_{CF} = (S, s_0, \Sigma, Z, \delta, F)$, there exists a Pushdown Automaton $P = (Q, \Sigma, \Gamma, \delta', q_0, F')$ such that $L(P) = L(A_{CF})$.

Construction C.1. We construct the PDA P by a direct mapping of the components from the agent A_{CF} . (1) Let the set of states Q = S. (2) The input alphabet Σ is identical. (3) Let the stack alphabet $\Gamma = Z$. (4) Let the start state $q_0 = s_0$. (5) Let the set of accept states F' = F. (6) The PDA's transition function δ' is defined directly from the agent's transition function δ . For every transition rule in the agent of the form $(s', \alpha) \in \delta(s, a, z)$, where $s, s' \in S$, $a \in \Sigma_{\epsilon}$, $z \in Z_{\epsilon}$, and $\alpha \in Z^*$, we create a corresponding PDA transition rule $(s', \alpha) \in \delta'(s, a, z)$. We use acceptance by final state in both models The simulator preserves the single-step relations \vdash over configuration (s, w, α)

PROOF. The proof proceeds by induction on the number of computational steps. We show that a configuration (s, w, α) , representing the agent's state, remaining perceptions, and stack memory, is reachable in A_{CF} if and only if the identical configuration (s, w, α) is reachable in the constructed PDA P. Since the components and transition rules are mapped one-to-one, every computational path

in A_{CF} has a corresponding path in P. Therefore, an input string w is accepted by A_{CF} (i.e., leads to a state in F) if and only if it is accepted by P (i.e., leads to a state in F'). Thus, $L(A_{CF}) = L(P)$. \square

C.2 From Pushdown Automaton to Context-Free Agent (⇐)

LEMMA C.3. For any Pushdown Automaton $P = (Q, \Sigma, \Gamma, \delta, q_0, F)$, there exists a Context-Free Agent $A_{CF} = (S, s_0, \Sigma, Z, \delta', F')$ such that $L(A_{CF}) = L(P)$.

Construction C.2. This construction is the symmetric inverse of the previous one. The agent A_{CF} is constructed to be a perfect simulator of the PDA P. (1) Let the set of states S = Q. (2) The perception alphabet Σ is identical. (3) Let the agent's stack alphabet $Z = \Gamma$. (4) Let the initial state $s_0 = q_0$. (5) Let the set of accepting states F' = F. (6) The agent's transition function δ' is defined to mirror the PDA's transition function δ . For every PDA transition $(q', \alpha) \in \delta(q, a, \gamma)$, we create an agent transition $(q', \alpha) \in \delta'(q, a, \gamma)$.

PROOF. The argument for correctness is identical to that in the previous Lemma. The agent's components and transition logic are defined to be isomorphic to those of the PDA. As a result, the agent simulates the PDA's computation step for step, and they necessarily accept the same set of input strings.

The mapping holds under strictly nested subtask management (LIFO): no cross-returns, no arbitrary jumps into non-top frames, and no parallel subroutines that interleave stacks. Under these constraints, hierarchical plans are isomorphic to a stack discipline and are justifyable using the PDA equivalence.

LEMMA C.4 (PLANNING IMPLIES PDA). Let an agent maintain a stack of frames, each frame encoding the current subgoal and return point, and restrict control-flow to push on subgoal entry and pop on completion. Then the induced acceptor is equivalent to a (possibly deterministic) pushdown automaton.

PROOF SKETCH. Encode each frame as a symbol from a finite stack alphabet Z; the controller's finite states are S. On subgoal entry, transition with a push; on return, transition with a pop; ϵ -moves handle intra-frame steps. This yields a PDA $(S, \Sigma, Z, \delta, s_0, F)$ recognizing the same language of successful perception traces.

D The Equivalence of TC Agents and TMs

This final proof establishes the equivalence for the highest level of the Chomsky hierarchy. It formalizes the claim from that agents possessing an unbounded, arbitrarily readable and writable memory are computationally equivalent to the universal model of computation, the TM.

Theorem D.1. A language L is recognized by a TC Agent A_{TC} if and only if L is a recursively enumerable language (i.e., recognized by a TM).

The proof is one of mutual simulation, showing that each model has necessary components to execute the computations of the other.

D.1 From TC Agent to TM (\Rightarrow)

LEMMA D.2. For any TC Agent

$$A_{TC} = (S, s_0, \Sigma, \Gamma, \delta, F)$$

there exists a TMM = $(Q, \Sigma, \Gamma', \delta', q_0, q_{accept}, q_{reject})$ such that $L(M) = L(A_{TC})$.

Construction D.1. We construct a TM M that simulates the agent A_{TC} . A multi-tape TM provides a clear conceptual model, which can be simulated by a standard single-tape TM. (1) Tape 1 (Input Tape): This tape holds the input perception sequence w and is treated as read-only. (2) Tape 2 (Memory Tape): This tape simulates the agent's unbounded memory or scratchpad. In a more practical sense, when implementing an AI agent, one can also think of RAG-based retrieval in this same setting. (3) Simulation Logic: The TM's finite control Q and transition function δ' are programmed to execute the agent's transition function δ . (4) Acceptance: If the agent's simulated state enters the set of accepting states F, the TM M transitions to its own accept state, $q_{\rm accept}$. Note that here we follow the previously set convention that the input is encoded on the memory tape; the transition depends on the current tape symbol in Γ

PROOF. The construction provides a direct simulation. The agent's unbounded memory maps directly to the TM's infinite tape. The agent's reasoning engine, encapsulated in its transition function δ , is a finite set of rules that can be implemented by the TM's finite control. A configuration of the agent is fully captured by a configuration of the TM. Therefore, the agent A_{TC} accepts an input sequence w if and only if the TM M halts in an accepting state, which implies $L(A_{TC}) = L(M)$.

D.2 From TM to TC Agent (\Leftarrow)

LEMMA D.3. For any TM $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$, there exists a TC Agent A_{TC} such that $L(A_{TC}) = L(M)$.

Construction D.2. We construct an agent A_{TC} that is programmed to simulate the TM M. (1) **Agent Memory:** The agent's unbounded, read/write memory is used to store the contents of M's tape. (2) **Agent State:** The agent's internal state $s \in S$ is a tuple (q, i), where $q \in Q$ is the current state of the simulated TM M, and i is an integer representing the current position of M's tape head. (3) **Agent Logic:** The agent's transition function δ' is an algorithm that implements M's transition function δ . (4) **Acceptance:** The set of accepting states F for the agent consists of all internal states (q, i) where $q = q_{\text{accept}}$.

PROOF. The agent A_{TC} is effectively a universal computer programmed to execute a simulation of another specific computer, M. Its components are sufficient to track every aspect of M's configuration. The agent's computation will halt in an accepting state if and only if the simulated machine M halts in its accepting state. Therefore, $L(A_{TC}) = L(M)$.

This proof provides the formal justification for classifying agents that use an unconstrained "scratchpad," such as those in the ReAct or Auto-GPT frameworks, as TC. The critical insight is that the

physical medium of the memory is irrelevant from a computability perspective. The defining characteristics are operational: unbounded storage capacity, arbitrary read access to any part of the memory, and arbitrary write access. Any system that provides these three features is computationally equivalent to a TM's tape. This

equivalence is not metaphorical; it is a direct consequence of the memory architecture. It confirms that such agents inherit the full computational power of universal machines, but also all of their theoretical limitations, most notably the undecidability of fundamental properties like halting and safety.