An Experiment on the Simulation of Deterministic Turing Machines by Large Language Models

Guilherme Lima[1,2], **Cleyton Slaviero**[3], **Edward H. Haeusler**[2]

[1]PUC-Behring Institute for AI – PUC-Rio
Rio de Janeiro, Brazil

[2]Department of Informatics – PUC-Rio
Rio de Janeiro, Brazil

[3]Faculdade de Ciência e Tecnologia
Universidade Federal de Mato Grosso – Campus Várzea Grande
Cuiabá, Brazil

glima@puc-rio.br, cleyton.slaviero@ufmt.br, hermann@inf.puc-rio.br

Abstract. *Recent studies have shown that the use of chain-of-thought (CoT) under suitable assumptions enables large language models (LLMs) and transformers in general to achieve Turing completeness. In this paper, we put this claim to the test. We evaluate empirically the feasibility of using state-of-art LLMs with zero-shot CoT prompting to simulate arbitrary deterministic Turing machines. We tested four flagship models (DeepSeek V4 Pro, Gemini 2.5 Pro, GPT 5.4 and 5.2) over 116 randomly generated Turing machines with varying configurations. As expected, accuracy drops as the number of transitions of the simulated machine grows, but some models/modes of operation perform notably better than others, which corroborates the results of a related benchmark.*

1. Introduction

Large language models (LLMs) have been shown to exhibit remarkable performance in complex reasoning tasks, often far surpassing those of traditional machine learning. The use of chain-of-thought (CoT), a technique in which the model is instructed to generate a series of intermediate steps, has been demonstrated to greatly improve LLM's reasoning abilities. In particular, CoT make it possible for LLMs and transformers in general to simulate complex computational tasks and, under suitable assumptions, achieve Turing completeness [Pérez et al. 2021, Li et al. 2024, Merrill and Sabharwal 2024, Jiang et al. 2026]. These results have motivated a growing body of research on the algorithmic capabilities and limitations of LLMs [Malfa et al. 2024, Zhou et al. 2024, Hua et al. 2026].

While LLMs may be theoretically capable of simulating arbitrary computations, their execution (inference) is inherently stochastic and therefore prone to errors. Practical limitations such as the maximum size of context windows can also hinder the feasibility of such simulations, even in a CoT setting. Other important constraints are the time it takes for the LLM to actually perform the simulation and the monetary cost of doing so, especially considering the steep prices of flagship models.

In this paper, we investigate the feasibility of simulating arbitrary computational tasks using off-the-shelf LLMs with zero-shot chain-of-thought prompting. By "off-the-shelf LLMs" we mean recent models from popular providers and by "zero-shot" we mean

without using examples. The ultimate test of computational ability is *Turing completeness*, or the capacity of simulating an arbitrary Turing machine. In a sense, what we are evaluating in this paper is the extent to which using a natural language prompt as the executable specification of a computational task is feasible in practice. The "computational task" in this case is running a deterministic Turing machine over a given input with the LLM playing the role of the universal machine.

To evaluate the ability of LLMs to simulate deterministic Turing machines, we proceed as follows. First, we use a system prompt to establish the format to be used to represent Turing machines and the steps required to compute a transition. This is done in an abstract manner, without referring to any specific machine. Then, in a human prompt, we present a specific machine to be simulated together with its input data. From this point on, we enter a loop in which we ask the LLM to execute one transition of the simulated machine and check the model's output at each iteration. Eventually, either the LLM produces an invalid transition or it reaches a state in which the simulated machine is supposed to halt. In either case, we stop the evaluation and record the total number of correct transitions.

We used the previous approach to evaluate four recent LLMs (DeepSeek's V4 Pro, Google's Gemini 2.5 Pro, OpenAI's GPT 5.4 and 5.2) over 116 randomly generated Turing machines with varying configurations. These machines and the Python framework used to simulate their execution via LLMs, called UMLLM, are made available under an open-source license (Apache 2.0).[1] Gemini 2.5 Pro performed notably better than the other models, which corroborates the results observed in a related benchmark [Wu et al. 2025]. As expected, performance deteriorates significantly as the number of transitions grows. For example, from 60 transitions onward, the accuracy of Gemini 2.5 Pro drops below 25%, i.e., it predicts a correct transition only about 25% of the time.

The rest of the paper is organized as follows. Section 2 presents background on Turing machines and discusses related work. Section 3 presents our approach to simulate the execution of Turing machines using LLMs with zero-shot CoT prompting. Section 4 presents the experimental evaluation, including the dataset, metrics, setup, and results. Section 5 presents our conclusions and discusses some future work.

2. Background

The *Turing machine* is a mathematical model of computation introduced by Alan M. Turing [Turing 1936]. Formally, a Turing machine is a quadruple $M = (K, \Sigma, \delta, s, h)$ where K is a finite set of states, Σ is a finite set of symbols, δ is the transition function, and s and h are the initial and halting states.[2] Intuitively, the machine consists of a current state, an infinite one dimensional tape of symbols, and a read-write head. Function δ is the "program" of the machine. It specifies, for each combination of current state $q \in K$ and current symbol $\sigma \in \Sigma$, a triple $\delta(q, \sigma) = (p, \rho, D)$, where p is the next state, ρ is the symbol to be overwritten on σ, and D is either "L" (left) or "R" (right).

A Turing machine M starts in the initial state s with some finite input string of symbols written on its tape and the head at some initial position. It then makes a transition according to δ, changing its state, writing a symbol, and moving its head left or right. This

[1] https://github.com/PUC-Behring-AI/umllm

[2] This is a slight variation of the definition found in [Papadimitriou 1994].

process repeats until the machine reaches state h, at which point it halts. The content of the tape at this moment is the machine's output. If the machine halts on a given input x, we can define the output of the machine M on x, denoted $M(x)$. It is possible that M never halts on input x, in which case its output for x undefined, i.e., $M(x) = \uparrow$.

The description above assumes that δ is a total function, which means that at each step during the computation the machine has a single uniquely defined next step. This type of machine is called *deterministic*, as opposed to non-deterministic ones in which there may be more than one appropriate next step. In this paper, we deal only with deterministic Turing machines.

One fundamental fact about Turing machines, established by Turing in its seminal paper, is the existence of universal Turing machines. A *universal Turing machine* U is a Turing machine that interprets its input as the description of another Turing machine M concatenated with the description of an input x to that machine. U's function is then to simulate the behavior of M when presented with input x, i.e., $U(M; x) = M(x)$. The ability to simulate arbitrary Turing machines is precisely what we are trying to achieve here using LLMs. But before simulating the execution of a Turing machine using an LLM or any other computational process, we need to agree on a format for describing arbitrary machines and their inputs. We introduce the particular format we adopt in this paper next.

2.1. Describing and Simulating Arbitrary Turing Machines

To encode the description of a Turing machine we use the symbols "Q", "S", "0", "1", "L", "R", and "␣". The idea is to use strings of the form Q0, Q1, Q10, Q11, ... to represent the states of the machine and strings of the form S0, S1, S10, S11, ... to represent its (non-blank) symbols. (For readability, we sometimes use subscripts when writing such strings, e.g., $\mathtt{Q}_{1011}$ instead of Q1011 and $\mathtt{S}_{101}$ instead of S101.) The symbols L and R are used to specify the direction of movement (left or right) of the head when encoding the description of the transition function of the machine. Finally, symbol ␣ represents the blank symbol used, among other things, to delimit the written portion of the machine's tape.

We describe the transition function of the simulated Turing machine by giving its transition table. For example, here is part of the transition table of a Turing machine INVERT, which "inverts" all bits to the right of the current head position:

$$\mathtt{Q}_0\mathtt{S}_0\mathtt{Q}_0\mathtt{S}_1\mathtt{R} \tag{1}$$
$$\mathtt{Q}_0\mathtt{S}_1\mathtt{Q}_0\mathtt{S}_0\mathtt{R} \tag{2}$$
$$\mathtt{Q}_0\mathtt{␣}\mathtt{Q}_1\mathtt{␣}\mathtt{R} \tag{3}$$

This transition table defines three transitions prescribing that:

1. When in state $\mathtt{Q}_0$, reading symbol $\mathtt{S}_0$, the machine shall stay in state $\mathtt{Q}_0$, write $\mathtt{S}_1$ over the current head position, and then move the head to the right.
2. When in state $\mathtt{Q}_0$ reading $\mathtt{S}_1$, the machine shall stay in $\mathtt{Q}_0$, write $\mathtt{S}_0$, and move right.
3. When in state $\mathtt{Q}_0$ reading ␣, the machine shall change to $\mathtt{Q}_1$, write ␣, and move right.

To describe the content the tape, the current state, and the position of head of the simulated machine, we use strings of the form

$$\text{␣}\ s_i \dots s_j\ \mathtt{Q}_k\ s_m \dots s_n\ \text{␣} \tag{4}$$

where each s is either a blank (␣) or some non-blank symbol of the machine. We call such strings *work tapes* and we require that they be delimited by blanks on the extremities, a requirement that must be observed by the simulation process.

A work tape such as (4) above indicates three things: (i) the machine's current state, namely, $\mathtt{Q}_k$; (ii) the written portions of the machine's tape to the left and right of the head, namely, "␣ $s_i \ldots s_j$" and "$s_{m+1} \ldots s_n$ ␣"; and (iii) the symbol being currently read by the head, namely, s_m. A transition of the machine operates over its current work tape and produces a new work tape.

Besides giving the transition table and initial work tape, the last thing missing to completely describe a Turing machine to be simulated is specifying its final state. We do this by indicating the particular $\mathtt{Q}_f$ which is to be used to represent the final state. The machine is assumed to halt whenever $\mathtt{Q}_f$ occurs in the work tape.

```
# transition table
Q0S0 Q0S1R
Q0S1 Q0S0R
Q0_  Q1_R

# halting state
Q1

# initial work tape
_Q0S1S1S0S0S1S1S0S0_
```

Figure 1. The INVERT Turing machine in UMLLM format.

Figure 1 gives a complete specification of Turing machine INVERT together with some sample input in the format we just described. This is exactly the format adopted by UMLLM, the framework for simulating Turing machines via LLMs presented in Section 3. We call this format the UMLLM format. It consists of three blocks separated by empty lines. First, comes the transition table, which is given as described above with any uninteresting transitions omitted if desired. (If such transitions are ever reached during simulation, they are deemed invalid.) The second block contains the halting state and the third block contains the initial work tape. ASCII whitespace within blocks and lines beginning with "#" are ignored by the parser.

To simulate the execution of the machine given in Figure 1, we start with the initial work tape and proceed to update it one transition at a time, following the rules prescribed by the transition table. The diagram in Figure 2 displays the work tapes produced by successive transitions of machine INVERT when initialized as in Figure 1. Each transition is indicated by an arrow labeled with the position of the transition in the transition table (first, second, or third). After nine transitions the machine halts and the content of the work tape is exactly the content of the initial work tape with each bit inverted (i.e., $\mathtt{S}_0$ replaced by $\mathtt{S}_1$ and $\mathtt{S}_1$ replaced by $\mathtt{S}_0$).



Figure 2. Execution trace of INVERT.

In Section 3, we describe an approach to instruct an LLM to execute a sequence of transitions such as that of Figure 2 for a given UMLLM specification. Before that, however, let's review related work on the simulation of computational processes by LLMs.

2.2. Related Work

There are two kinds of work related to this paper. First, there are the theoretical results on the computation ability of LLMs, or more generally, the transformer architecture. For example, it has been established that transformers can represent arbitrary computations of any finite-state automaton [Liu et al. 2023] and, under suitable assumptions, even simulate arbitrary Turing machines [Pérez et al. 2021, Jiang et al. 2026]. It has also been argued that chain-of-thought (CoT), a technique in which the model auto-repressively generates a sequence of intermediate results before giving the final answer, is essential in enabling transformers to perform the serial computations required for solving complex reasoning tasks [Li et al. 2024].

The second kind of work, more closely related to this paper, are those on the experimental evaluation of the computational ability of LLMs. One such work is [Wu et al. 2025] which proposes TMBench, a benchmark for assessing the capacity of LLMs in simulating programs expressed in a Turing-complete formalism. Although the benchmark is called TMBench, with the "TM" standing for Turing machine, the formalism it adopts is not that of Turing machines. Instead it uses what the authors call *m*-tag systems which are a version of the classical formalism introduced by E. Post [Post 1943]. A further difference between [Wu et al. 2025] and our work, beyond the computation formalism used, is the prompting technique. While they use one-shot non-CoT prompting (one example, no chain-of-thought instruction), we use zero-shot CoT prompting. The experimental results of [Wu et al. 2025] are thus not directly comparable to ours but, as we will see in Section 4, the best performing model in both benchmarks was Gemini 2.5 Pro.

Besides TMBench, we could find no other benchmark that uses Turing machines or related formalisms for evaluating the computational capacity of LLMs. There are, of course, many benchmarks for assessing the ability of LLMs in solving tasks that require some form of computation (cf. [Wang et al. 2026]).

3. Method

To simulate the execution of a deterministic Turing machine by an LLM, we use single a *system prompt* describing:

- The format used to encode the simulated machine (see Section 2.1).
- The data-structures to be used during the simulation. These consist of ten tapes: the Work tape of the simulated machine, a read-only Machine tape containing the transition table, and auxiliary tapes holding intermediate data.
- The seven steps required to compute a single transition of the simulated machine. These steps use the aforementioned tapes to construct and apply a single string substitution operation on the Work tape.
- The format to be used by the LLM to present the results (i.e., the resulting Work tape) and the commands which it should expect to receive in subsequent interactions (either "continue" or "stop").

The actual system prompt we use is shown in Figure 3. In the last paragraph of the prompt, we ask the LLM to explain what it is doing during the execution of each of the seven steps that compute the transition. This is the chain-of-thought instruction. Although we are not interested in these explanations, it has been observed that the mere fact of producing them improves the accuracy of the model in tasks of this kind.

Let's simulate a single transition of a deterministic Turing machine encoded in an alphabet with the symbols `Q`, `S`, `0`, `1`, `L`, `R`, and `_`.

Certain strings of these symbols are interpreted as follows:
* `Qx`, `Qy` represent states of the simulated machine, where `x`, `y` are sequences of `0`s and `1`s;
* `Su`, `Sv`, `Sw` represent symbols of the simulated machine, where `u`, `v`, `w` are sequences of `0`s and `1`s;
* `R` indicates a movement to the right;
* `L` indicates a movement to the left;
* `_` represents the blank symbol - sometimes `Su`, `Sv`, `Sw` will be assumed to possibly denote blanks.

To execute a single transition of the simulated Turing machine, the following tapes are used: Machine, Work, State, Symbol, LeftSymbol, NextState, NextSymbol, NextMove, Subst1, Subst2

At all times, there must be blanks delimiting the written region of any of the tapes. That is, if a blank in one of the extremities of a tape is replaced by a non-blank symbol, then a new blank must be added beside the newly written symbol: to its left if the symbol is on the left extremity of the tape or to its right if the symbol is on the right extremity of the tape.

Initially, only the tapes Machine and Work contain symbols. The other tapes are assumed to be initially empty.

Follow strictly the instructions below without inferring any additional steps.

* **Step 1** - Determine State: Find in the Work tape a substring of the form `Qx` and copy it to the State tape.

* **Step 2** - Determine Symbol: Let `Sv` be the symbol or blank immediately to the right of `Qx` in the Work tape. Copy `Sv` to the Symbol tape.

* **Step 3** - Determine LeftSymbol: Let `Su` be the symbol or blank immediately to the left of `Qx` in the Work tape. Copy `Su` to the LeftSymbol tape.

* **Step 4** - Determine NextState, NextSymbol, and NextMove: The Machine tape is read-only and contains a finite sequence of strings of the form `QxSvQySwM` where `Qx` and `Qy` stand for states, `Sv` and `Sw` stand for symbols of the simulated machine or blanks, and `M` is either `L` or `R`. Search in the Machine tape for a substring of the form `QxSvQySwM` where `State = Qx` and `Symbol = Sv`, that is, such that `Qx` and `Sv` coincide with the contents of the tapes State and Symbol, respectively. Then copy `Qy` to the NextState tape, `Sw` to the NextSymbol tape, and `M` to the NextMove tape. Note that `Sv` and `Sw` might be blank.

* **Step 5** - Determine Subst1: Let `State = Qx`, `Symbol = Sv`, and `LeftSymbol = Su`. If `NextMove = R`, write `QxSv` into the Subst1 tape. Otherwise, if `NextMove = L`, write `SuQxSv` into the Subst1 tape.

* **Step 6** - Determine Subst2: Let `NextState = Qy` and `NextSymbol = Sw`. If `NextMove = R`, write `SwQy` into `Subst2`. Otherwise, if `NextMove = L` and `LeftSymbol = Su`, write `QySuSw` into the Subst2 tape.

* **Step 7** - Perform the substitution: Replace the contents of Subst1 by the contents of Subst2 in the Work tape.

You must execute each transition of the simulated Turing machine interactively. You shall be provided with the contents of the Machine tape and the initial contents of the Work tape. You shall then execute the above steps 1 to 7 in order, explaining what is being done during each step. After performing the substitution (step 7), you must stop and display the full contents of the Work tape on a new line surrounded by the tags <work> and </work>. You shall then wait for my command. Either `continue`, to execute another transition, or `stop`, to end the execution.

Figure 3. System prompt.

The system prompt is followed by a *human prompt* (or user prompt) that describes the actual Turing machine to be simulated together with its input. Figure 4 shows the human prompt corresponding to the machine INVERT described in Section 2.1.

Note that neither the system prompt nor the human prompt mention the existence of initial or final states. This is because the goal here is to use the LLM solely to compute transitions of the simulated machine, i.e., constructing and applying a single substitution to the current work tape, as described in the system prompt. Every time the LLM replies with a new work tape, we process it to see if it is correct (i.e., if it corresponds to the expected one) and then check to see if it contains the halting state. If so, we stop the execution. Otherwise, we ask the LLM to perform another transition.

Figure 5 displays the response generated by OpenAI's GPT-5.4 after receiving the

Consider the initial configuration in which the Machine and Work tapes contain the following:

* Machine: Q0S0Q0S1R Q0S1Q0S0R Q0_Q1_R
* Work: _Q0S1S1S0S0S1S1S0S0_

Execute one transition of the simulated Turing machine (steps 1 to 7), display the contents of the Work tape after the transition, and wait for my command.

Figure 4. Human prompt of INVERT.

system and human prompts shown above. Note that the model adopts the terminology and notation introduced in the system prompt to explain each step it preformed. The only part of the response the matters to us, however, is the string between the tags <work> and </work>, which contains the new work tape, obtained after executing a single transition of the simulated machine. Incidentally, the work tape shown here (highlighted in red) is correct. It matches exactly the work tape obtained after the first transition (see Figure 2).

Step 1 - Determine State: In the Work tape `_Q0S1S1S0S0S1S1S0S0_`, the substring of the form `Qx` is `Q0`. Copy `Q0` to the State tape.

Step 2 - Determine Symbol: The symbol immediately to the right of `Q0` in the Work tape is `S1`. Copy `S1` to the Symbol tape.

Step 3 - Determine LeftSymbol: Immediately to the left of `Q0` in the Work tape there is a blank `_`. Copy `_` to the LeftSymbol tape.

Step 4 - Determine NextState, NextSymbol, and NextMove: Search the Machine tape `Q0S0Q0S1R Q0S1Q0S0R Q0_Q1_R` for a rule matching:
- State = `Q0`
- Symbol = `S1`

The matching rule is `Q0S1Q0S0R`. So copy:
- `Q0` to the NextState tape
- `S0` to the NextSymbol tape
- `R` to the NextMove tape

Step 5 - Determine Subst1: We have:
- State = `Q0`
- Symbol = `S1`
- LeftSymbol = `_`
- NextMove = `R`

Since `NextMove = R`, write `Q0S1` into the Subst1 tape.

Step 6 - Determine Subst2: We have:
- NextState = `Q0`
- NextSymbol = `S0`
- NextMove = `R`

Since `NextMove = R`, write `S0Q0` into the Subst2 tape.

Step 7 - Perform the substitution: Replace `Q0S1` by `S0Q0` in the Work tape.

Work tape before substitution:
`_Q0S1S1S0S0S1S1S0S0_`
Work tape after substitution: `_S0Q0S1S0S0S1S1S0S0_`

<work>_S0Q0S1S0S0S1S1S0S0_</work>

Waiting for your command: `continue` or `stop`.

Figure 5. GPT-5.4's response after receiving the system prompt and the human prompt of INVERT.

3.1. Implementation

We implemented the approach described above as a Python library called UMLLM, which is made available under the Apache 2.0 license.[3] UMLLM is a framework for the execution of Turing machines. It comes with a traditional Turing machine executor written in Python which implements the same algorithm described in the system prompt (Figure 3). It also comes with CLI and web interfaces that can be used to execute machines in the UMLLM format interactively.

The interesting part of UMLLM is, of course, its support for LLMs. It provides LLM-backed executors which can be used run Turing machines using the approach described previously. To interface with the LLMs, UMLLM uses the LangChain[4] library,

[3] https://github.com/PUC-Behring-AI/umllm
[4] https://www.langchain.com/

more specifically, its chat model API which provides a uniform message-based interface for invoking language models from different providers.

One important point about LangChain's chat model API is that the core interface for invoking LLMs is stateless. That is, one sends a series of messages to the LLM and receives back one or more messages comprising the model's reply with no data carrying over to the next invocation, at least not implicitly. This means that to implement the "continue" command described in the system prompt, we have to send to the LLM the full message history. This history contains the initial system and human prompts, the 1st model reply, the 1st "continue" command (which is another human prompt), the 2nd model reply, the 2nd "continue" command, and so on. That said, UMLLM can be configured to *trim* the history to a desired maximum length. For example, if we set this length to 0, at each LLM invocation, instead of sending the full history plus a "continue" command, UMLLM will send only the system and user prompts with the initial work tape updated to reflect the current configuration of the simulated machine. We compare both modes of operation, full history (untrimmed) versus no history (trimmed), in the evaluation.

4. Evaluation

For the experimental evaluation, we generated a dataset with 116 random Turing machines in the UMLLM format. The varieties of machines in the dataset are described in Table 1. Every machine has 2, 4, or 8 states (parameter **#Q**) and an alphabet with exactly two non-blank symbols ($\mathtt{S}_0$ and $\mathtt{S}_1$). The written portion of their initial work tape contains exactly 8, 16, 32, or 64 symbols (parameter **#W**) and each machine is guaranteed to halt after a specific number of transitions, either 8, 16, 32, 64, or 128 (parameter **#T**). The dataset contains two random machines for every combination of these parameters with the exception of $\#\mathbf{Q} = 2$ and $\#\mathbf{W} = 8$ with $\#\mathbf{T} = 64$ or $\#\mathbf{T} = 128$, for which we could find no examples. Hence the number 116 ($= 2 \times 3 \times 4 \times 5 - 4$). Figure 6 illustrates one of the machines in the dataset.

Table 1. The tested Turing machine variations.

	variations	description
#Q	2, 4, 8	number of states
#W	8, 16, 32, 64	length of the initial work tape
#T	8, 16, 32, 64, 128	number of transitions until halting

```
# transition table
Q0S0Q0S1LQ0S1Q0_LQ0_Q11S0RQ1S0Q0S0RQ1S1Q0S0RQ1_Q1S0LQ10S0Q11S0LQ10S1Q10S1LQ10_Q1
↪  S0RQ11S0Q11_LQ11S1Q11S1LQ11_Q0S0L

# halting state
Q0

# initial work tape
_S0S0S1S1S0S0S0S0S1S0S1S0S0S0S1S1S0S0S1S1S1S0S1S1S1S0S1S1Q11S0S1S0S1S1S0S1S0
↪  S1S0S0S1S0S1S1S1S0S1S0S0S1S0S0S0S1S1S0S0S0S0S1S0S0S0_
```

Figure 6. One of the Turing machines in the dataset ($\#\mathbf{Q} = 4$, $\#\mathbf{W} = 64$, $\#\mathbf{T} = 32$).

We tested four different LLMs via the official APIs of their providers: DeepSeek's V4 Pro (DSV4), Google's Gemini 2.5 Pro (GEM2.5), and OpenAI's GPT 5.4 (GPT5.4) and 5.2 (GPT5.2). For more information, including prices, see Table 2. When calling the models, the only API parameters we set explicitly were *temperature* and *seed*, both set to zero. All other parameters were left with their default values. (We wanted to test each model in its vanilla configuration, which is the one seen by the average user.)

Table 2. The tested LLMs (prices in USD, 2026-06-08).

provider	model	abbrev.	snapshot	$/1M in / out tokens
DeepSeek	deepseek-v4-pro[5]	DSV4	2026-04-24	$0.435 / $00.87
Google	gemini-2.5-pro[6]	GEM2.5	2025-06	$2.50 / $15.00
OpenAI	gpt-5.4[7]	GPT5.4	2026-03-05	$2.50 / $15.00
	gpt-5.2[8]	GPT5.2	2025-12-11	$1.75 / $14.00

We run each model over every machine in the dataset twice. Once in *full-history mode*, in which on each interaction we would send the full message history, and once in *no-history (trimmed) mode*, in which on each interaction we would send exactly two messages, one with the system prompt and one with the human prompt containing the updated work tape. This gives a total of $2 \times (4 \times 116) = 928$ runs. The total and average (per run) token consumption and USD expenditure values for each model-mode is shown in Table 3. We use the suffix (tr) to distinguish trimmed runs from untrimmed runs. Note the higher input token consumption figures for untrimmed GEM2.5 and GPT5.2. These indicate that the system was able to maintain longer "conversations" (runs) with these models/modes. We will come back to this point later.

Table 3. Token consumption and expenditure (USD).

	DSV4	(tr)	GEM2.5	(tr)	GPT5.2	(tr)	GPT5.4	(tr)
in tokens	5,103K	1,448K	53,882K	2,035K	18,120K	654K	9,448K	1,278K
out tokens	2,172K	5,416K	4,265K	4,404K	670K	362K	665K	565K
$	$4.11	$5.34	$198.67	$71.14	$41.09	$6.21	$33.59	$11.66
avg. in tokens	44K	12K	465K	18K	156K	6K	81K	11K
avg. out tokens	19K	47K	37K	38K	6K	3K	6K	5K
avg. $	$0.04	$0.05	$1.71	$0.61	$0.35	$0.05	$0.29	$0.10

4.1. Results

Figure 7 shows the progress of all runs grouped by model/mode. We define the *progress* P of a run r as the number of correct transitions performed by the LLM divided by the total number of transitions required for the machine to halt:

$$P(r) := \frac{N_{\text{correct}}(r)}{N_{\text{total}}(r)}$$

Keep in mind that we stop the run as soon as the LLM produces an incorrect transition. So, $P(r) = 0$ means that the LLM failed on the first step, $P(r) = 0.5$ means it failed

[5]https://huggingface.co/deepseek-ai/DeepSeek-V4-Pro
[6]https://ai.google.dev/gemini-api/docs/models/gemini-2.5-pro
[7]https://developers.openai.com/api/docs/models/gpt-5.4
[8]https://developers.openai.com/api/docs/models/gpt-5.2

exactly midway during the execution, and $P(r) = 1$ means it executed every transition successfully until the end. As we can see in Figure 7, the values of progress of untrimmed GEM2.5 runs are significantly higher than those of the other models.

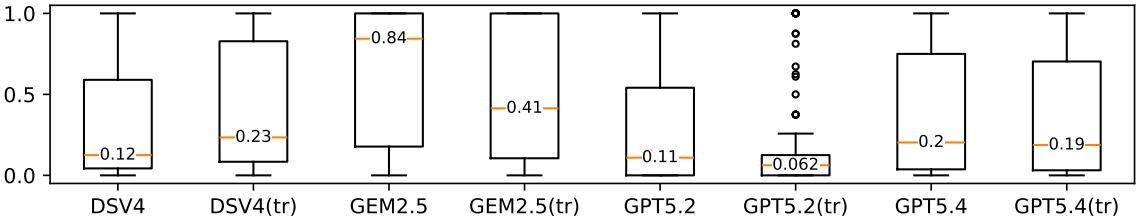


Figure 7. Run progress (for each run, # correct transitions / # total transitions).

A breakdown of run progress across the machine parameters #**Q**, #**W**, #**T** is shown in Figure 8. One thing we can immediately observe is that, in column #**T**, as the number of transitions required for the machine to halt grows, the overall progress values drop significantly across all models. A similar but less dramatic drop can be observed when the length of the initial work tape (#**W**) increases.

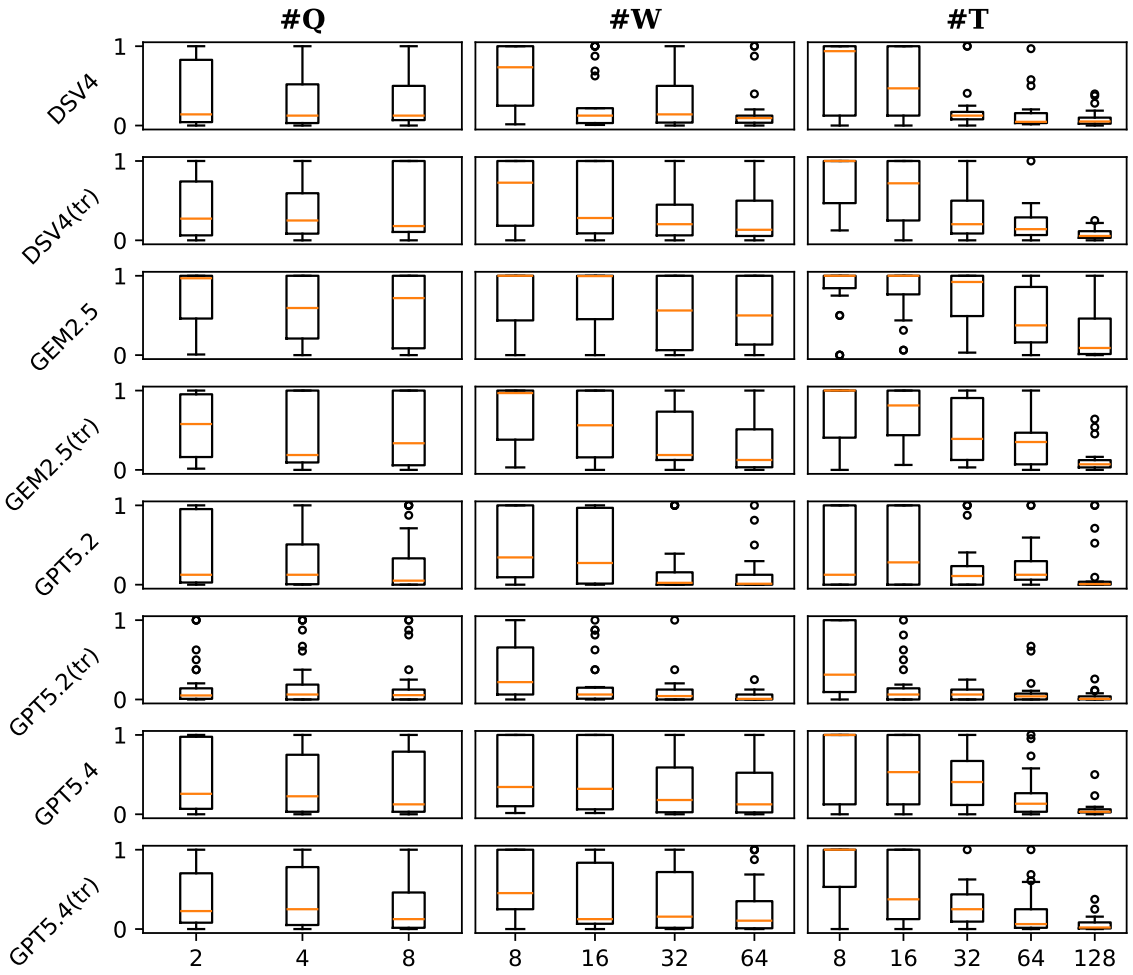


Figure 8. Run progress (detailed).

Another metric we used to evaluate the LLMs performance was transition-wise

accuracy, which quantifies the proportion of correct transitions at a each step of the execution [Wu et al. 2025]. The *(transition-wise) accuracy* at transition i of a given set R of runs is defined as

$$A_R(i) := \frac{\left|\{r \in R \mid N_{\text{correct}}(r) \geq i\}\right|}{\left|\{r \in R \mid N_{\text{total}}(r) \geq i\}\right|}$$

For example, $A_R(30) = 0.5$ means that at step 30 the proportion of correct transitions considering the runs with at least 30 transitions in R is 50%.

Figure 9 shows the transition-wide accuracy of all runs grouped by model/mode. The line plots (left) show the accuracy values for each transition i in the 0–128 range, while the box-plots (right) show aggregated accuracy values for the whole range. As we can see in the line plots, accuracy drops significantly after 20 transitions, with untrimmed GEM2.5 consistently obtaining the best accuracy values[5] and, somewhat surprisingly, untrimmed GPT5.2 performing better than the remaining models/modes in longer runs. In fact, only untrimmed GEM2.5 and GPT5.2 were able to complete runs with 128 transitions, which explains the input token consumption figures of Table 3.

As we can see in the box-plots, the overall accuracy of untrimmed runs is higher than that of their trimmed counterparts. Our hypothesis to explain this behavior is the following: the contents of earlier messages present in the history of untrimmed runs, in particular, earlier chain-of-thought excerpts, influence the model and somehow increase its chance of getting the current transition right.

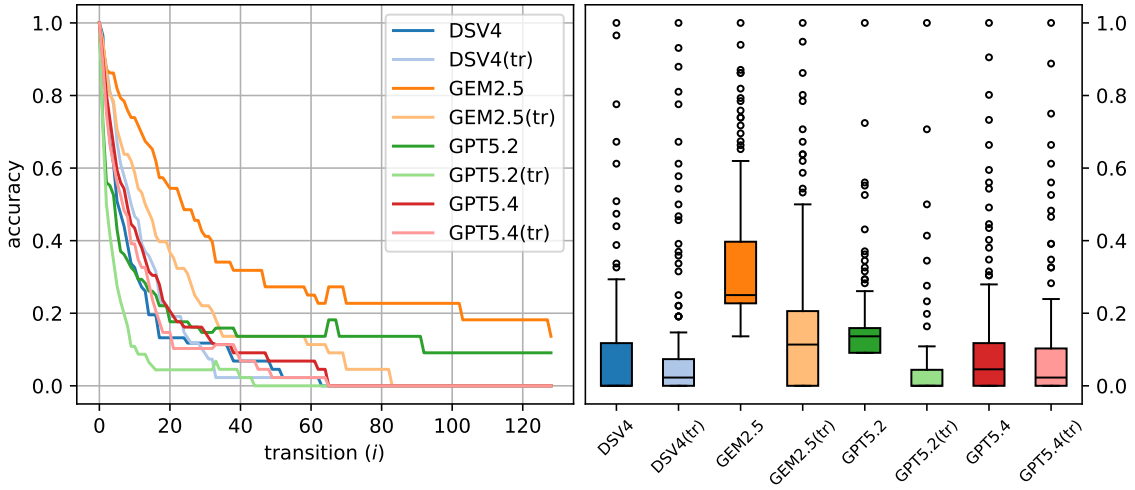


Figure 9. Transition-wise accuracy.

5. Conclusion

In this paper, we presented an empirical evaluation of the simulation of deterministic Turing machines by LLMs. We proposed an approach for accomplishing such simulations using zero-shot CoT prompting and evaluated it empirically by running four flagship LLMs over 116 randomly generated Turing machines. The fact that some runs were able to achieve the halting state after 128 transitions (the maximum number of transitions in the

[5] A similar behavior regarding GEM2.5 was observed in the evaluation done in [Wu et al. 2025], which used LLMs for simulating Post systems [Post 1943].

dataset) is a testament to the capacity of current LLMs and, to a certain degree, demonstrates the feasibility of our approach. That said, in general, accuracy drops sharply across all models after 20 transitions and the simulation is by no means efficient or cheap.

In future work, we intend evaluate further models and improve some known shortcomings of our approach. For example, the current CoT instruction is too simple. We could try to generate more accurate CoT by being more explicit in its instruction or use examples (few-shot) to increase the chances of eliciting a correct response. There is also room for improvement in the way history is being handled. We could use more advanced compression or summarization techniques to reduce the amount of data that is sent to the LLM on each iteration, which would reduce the overall latency and costs.

References

Hua, Z., Zhang, W., Lyu, C., Gu, Y., Gao, S., Liu, K., Lin, D., and Chen, K. (2026). The imitation game: Turing machine imitator is length generalizable reasoner. In *Proc. ICLR 2026*.

Jiang, H., Hahn, M., Zetzsche, G., and Lin, A. W. (2026). Softmax transformers are Turing-complete. In *Proc. ICLR 2026*.

Li, Z., Liu, H., Zhou, D., and Ma, T. (2024). Chain of thought empowers transformers to solve inherently serial problems. In *Proc. ICLR 2024*.

Liu, B., Ash, J. T., Goel, S., Krishnamurthy, A., and Zhang, C. (2023). Transformers learn shortcuts to automata. In *Proc. ICLR 2023*.

Malfa, E. L., Weinhuber, C., Torre, O., Lin, F., Marro, S., Cohn, A., Shadbolt, N., and Wooldridge, M. (2024). Code simulation challenges for large language models. https://arxiv.org/abs/2401.09074.

Merrill, W. and Sabharwal, A. (2024). The expressive power of transformers with chain of thought. In *Proc. ICLR 2024*.

Papadimitriou, C. H. (1994). *Computational Complexity*. Addison-Wesley, USA.

Post, E. L. (1943). Formal reductions of the general combinatorial decision problem. *Am. J. Math.*, 65(2):197–215.

Pérez, J., Barceló, P., and Marinkovic, J. (2021). Attention is Turing complete. *J. Mach. Learn. Res.*, 22(1).

Turing, A. M. (1936). On computable numbers, with an application to the Entscheidungsproblem. *Proc. London Math. Soc.*, 2(42):230–265.

Wang, P.-Y., Liu, T.-S., Wang, C., Li, Z., Wang, Y., Yan, S., Jia, C., Liu, X.-H., Chen, X., Xu, J., and Yu, Y. (2026). A survey on large language models for mathematical reasoning. *ACM Comput. Surv.*, 58(8).

Wu, H., Han, Z., Zhou, J. T., Huang, H., and Zhang, C. (2025). Computational reasoning of large language models. https://arxiv.org/abs/2504.20771.

Zhou, H., Bradley, A., Littwin, E., Razin, N., Saremi, O., Susskind, J. M., Bengio, S., and Nakkiran, P. (2024). What algorithms can transformers learn? A study in length generalization. In *Proc. ICLR 2024*.